Co-funded by the
Erasmus+ Programme
of the European Union

Accelerating Distributed Storage in Heterogeneous Settings

Waleed Reda

Doctoral Thesis in Information and Communication Technology
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2022
and
Institute of Information and Communication Technologies,
Electronics and Applied Mathematics
Université catholique de Louvain
Louvain-la-Neuve, Belgium 2022

KTH School of Electrical Engineering
TRITA-EECS-AVL-2022:32 and Computer Science
ISBN 978-91-8040-230-9 SE-164 40 Kista
UCLouvain Collection Number: 893|2022 SWEDEN

Academic dissertation which, with due permission of the KTH Royal Institute of Technology, is submitted for public defence for the Degree of Doctor of Technology on Monday, May 30 2022 at 15:00 CET in Sal C, Electrum, Kungliga Tekniska Högskolan, Kistagången 16, Kista.

© Waleed Reda, May 2022

Printed by: Universitetsservice US-AB

Abstract

Heterogeneity in cloud environments is a fact of life—from workload skews and network path changes, to the diversity of server hardware components, these are all factors that impact the performance of distributed storage. In this dissertation, we identify that heterogeneity can in fact be one of the primary causes of service degradation for storage systems. We then tackle this challenge by building next-generation distributed storage systems that can operate amidst heterogeneity while providing fast and predictable response times. First, we study skews in cloud workloads and propose scheduling strategies for key-value stores that seek to optimize latency. We then conduct a measurements study in one of the largest cloud provider networks to quantify variations in network latencies, and possible implications for storage services. Next, with fast non-volatile RAM (NVRAM) now becoming commercially available, we look into how storage systems can deal with the increasing diversity of storage technologies. We design and evaluate a distributed file system that can manage data across NVRAM and other types of storage, while providing low latency and high scalability. Lastly, we build a framework that transforms commodity Remote Direct Memory Access (RDMA) NICs into Turing machines—capable of performing arbitrary computations. This provides yet another compute resource on server machines, and we show how we can leverage it to accelerate common storage tasks as well as real storage applications.

Keywords

Distributed storage, File systems, Hardware offload, Persistent memory

Sammanfattning

Heterogenitet i molnmiljöer är ett livsfaktum—allt från skevheter i arbetsbelastningen och nätverksvägsförändringar till diversitet av serverhårdvarukomponenter, dessa är faktorer som påverkar prestanda för distribuerad lagring. I denna avhandling identifierar vi att heterogenitet faktiskt kan vara en av de främsta orsakerna till tjänste-degradering för lagringssystem. Vi tacklar sedan denna utmaning genom att bygga nästa generations distribuerade lagringssystem som kan fungera mitt i heterogenitet medan de ger snabba och förutsägbara svarstider. Först studerar vi skevheter i molnarbetsbelastningar och föreslår schemaläggningsstrategier för nyckelvärdelagring som försöker optimera latens. Vi utför sedan en mätstudie i ett av de största molnleverantörsnätverken för att kvantifiera variationer i nätverkslatenser och möjliga implikationer för lagringstjänster. Därefter, i och med att snabbt non-volatile RAM (NVRAM) nu blir kommersiellt tillgängligt, undersöker vi hur lagringssystem kan hantera den ökande diversiteten av lagringsteknik. Vi designar och utvärderar ett distribuerat filsystem som kan hantera data tvärs över NVRAM och andra typer av lagring, medan det ger låg latens och hög skalbarhet. Slutligen bygger vi ett ramverk som transformerar lättillgänglig Remote Direct Memory Access (RDMA) NICs till Turingmaskiner — som kan utföra godtyckliga beräkningar. Detta ger ännu en beräkningsresurs på servrar, och vi visar hur vi kan utnyttja den för att accelerera gemensamma lagringsuppgifter såväl som verkliga lagringsapplikationer.

Nyckelord

Distribuerad lagring, Dilsystem, Maskinvaruavlastning, Beständigt minne

Résumé

L'hétérogénéité dans les environnements cloud est une réalité : des écarts de charge de travail et des changements de chemins réseaux à la diversité des composants matériels dans les serveurs, tous ces facteurs ont un impact sur les performances du stockage distribué. Dans cette thèse, nous identifions que l'hétérogénéité peut en fait être l'une des principales causes de dégradation de service pour les systèmes de stockage. Nous relevons ensuite ce défi en créant des systèmes de stockage distribués de nouvelle génération capables de fonctionner dans un environnement hétérogène tout en offrant des temps de réponse rapides et prévisibles. Tout d'abord, nous étudions les asymétries dans les charges de travail cloud et proposons des stratégies de planification pour les base de données clé-valeurs qui cherchent à optimiser la latence. Nous menons ensuite une étude quantitative dans l'un des plus grands réseaux de fournisseurs de cloud pour caractériser les variations des latences du réseau et les implications possibles pour les services de stockage. Ensuite, alors que la RAM non volatile rapide (NVRAM) est désormais disponible dans le commerce, nous examinons comment les systèmes de stockage peuvent gérer la diversité croissante des technologies de stockage. Nous concevons et évaluons un système de fichiers distribué qui peut gérer les données sur la NVRAM et d'autres types de stockage, tout en offrant une faible latence et une évolutivité élevée. Enfin, nous construisons un cadre qui transforme les cartes réseau d'accès direct à la mémoire à distance (RDMA) de base en machines de Turing - capables d'effectuer des calculs arbitraires. Cela fournit encore une autre ressource de calcul sur les machines serveur, et nous montrons comment nous pouvons en tirer parti pour accélérer les tâches de stockage courantes ainsi que les applications de stockage réelles.

Mots clés

Stockage distribué, Systèmes de fichiers, Déchargement matériel, Mémoire persistante

Acknowledgments

During my PhD journey, I have had the privilege to work and learn from many talented individuals, whom I likely will not be able to give enough credit. First, I would like to thank my advisor Dejan Kostić for making this PhD possible. Dejan has been a tremendous source of guidance throughout my journey, and has given me enough room to flourish as an independent researcher. It is no exaggeration to say that he has gone over and above his duty as an advisor, and has had my back on several occasions (even at this own expense!) Dejan, I cannot thank you enough for all your support.

I would like to thank my co-author and member of my thesis committee, Marco Canini, for helping me get bootstrapped during the first years of my PhD and for guiding me towards my first publication. Marco has also done a great job in expanding my academic network, which made it easier to find collaborators. I would also like to thank my other committee members Marios Kogias, Charles Pecheur, and Etienne Riviere for all their helpful comments and feedback on my thesis. Many thanks to Peter Van Roy for helping organize my private defense.

I would like to extend my thanks to my co-author Simon Peter. Simon has hosted me as an intern for over 6 months at UT Austin. During my time there, I delved into an almost completely new research area and gained valuable experience designing and implementing next-generation distributed file systems. He has also helped brainstorm several ideas, which spurred additional papers. Our collaboration was fruitful and resulted in an OSDI, SOSP, and NSDI paper.

I would like to thank my friends and colleagues. First and foremost, big thank you to Kirill Bogdanov, whom I have had the pleasure of collaborating with over several years. Kirill has been a constant source of motivation as well as technical knowledge, and someone that I could share the PhD grind with on a daily-basis. He also happens to be an unmatched whiskey expert. I would also like to thank Georgios Katsikas, Voravit Tanyingyong, Alireza Farshin, Tom Barbette, Alexandros Milolidakis, and Hamid Ghasemirahni. It was a pleasure getting to know all of you over the past few years and sharing the same corridor.

I would like to thank Vladimir Vlassov for helping me understand the complex rules and regulations of my dual-degree PhD program. I would also like to give a shout out to Vanessa Maons for being an awesome human being, and helping with all the paperwork at UCLouvain.

I would like to also thank all the mentors I have had over the years. Thanks to my M.Sc. advisor Sameh El-Ansary for guiding me during my first foray into the world of academia, and helping me land a PhD position despite my lackluster undergraduate GPA (2.4 for those interested). Thanks also go to my highschool teacher Ahmed Al-Sioufy who helped build my initial interest in mathematics, which ultimately pushed me towards pursuing a career in STEM.

Credit also goes to the European Commission (EACEA) for supporting me via the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) fellowship during the first three years of my PhD.

Lastly, I would like to credit my family and, most importantly, the person who was the main reason I pursued this PhD. Father, what can I say. Thank you for everything. I will carry on..

Stockholm, May 2022
Waleed Reda

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The last few decades have witnessed massive growth in online cloud services and, consequently, the demands on the underlying storage infrastructure have also increased. To handle these shifts in demand and provide quick response times to the users of these services, storage systems have to provide low and predictable IO latencies and scale to a higher number of machines while being highly available. Failure to do so can lead to a dramatic reduction in end-user satisfaction and ultimately impact revenue. In this thesis, we explore the causes of unpredictable response times and performance fluctuations in cloud environments and argue that, heterogeneity—for user workloads, network state, and server hardware— represents both challenges and opportunities for distributed storage systems. This dissertation will aim to answer the question: *How to design next-generation distributed storage systems that provide fast and predictable response times in heterogeneous settings?* This will be done via an exploration of optimizations that make these systems more aware of the workload and network characteristics as well as the underlying hardware.

1.1 Tackling Heterogeneity

As mentioned, in cloud environments, heterogeneity can present itself across different layers. We outline the different types of heterogeneity based on recent trends and describe the associated challenges/opportunities in Fig. 1.1. We discuss briefly below how the different components of this thesis address these items.

We take a two-pronged approach to tackling heterogeneity. First, we focus on overcoming the challenges of workload and network heterogeneities by adapting the scheduling heuristics of storage systems to the environment. Second, we take

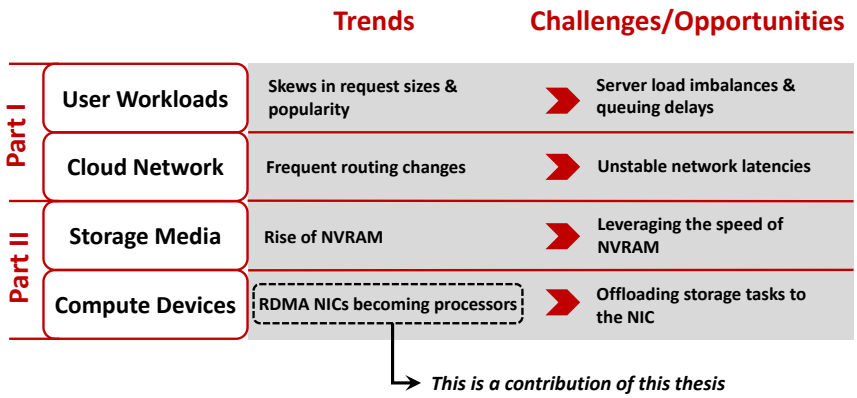


Figure 1.1: Current trends in cloud environments and the challenges and opportunities they present for distributed storage systems. Heterogeneity is a common theme among these trends.

advantage of the opportunities provided by new server hardware technologies by redesigning storage software to exploit their capabilities.

In part I of this thesis, we focus on adapting storage storage system scheduling policies. Our first project—entitled ***Rein***— examines recent workload trends for modern web applications and quantifies the level of workload heterogeneity. We find that "request sizes" are inherently skewed — where certain requests require orders of magnitude more resources to be serviced by the underlying storage infrastructure. We study these skews along with the challenges they pose for request completion times. We then propose several request-aware scheduling strategies that distributed key-value stores can leverage to tame tail latencies.

To adapt our scheduling to network conditions, we then perform a study on latency variations *across* datacenters within a large cloud provider network — namely, Amazon *AWS*. We show that latencies between datacenters in different geographic regions can vary unexpectedly and in small timescales (order of tens of seconds) due to routing changes. This behavior can lead to performance fluctuations for distributed storage systems deployed across multiple regions. We then provide insights on how we can use this information to make better scheduling decisions.

In part II, we shift our focus to redesigning storage system software to leverage state-of-the-art hardware. In the ***Assise*** project, we find that with the increasing diversity of storage devices available — now including Harddisk Drives (HDDs), Solid-State Drives (SSDs), and the recently introduced non-volatile random-access memory (NVRAM) — there are a slew of challenges to be addressed in order

to provide predictable response times. Firstly, NVRAM provides higher IO performance than SSDs and HDDs, which exposes bottlenecks in traditional client-server file system models. Secondly, with the increased diversity of storage technologies, file systems should be able to manage data across these devices in a transparent manner to the user. To address these challenges, we propose a new distributed file system model that pushes (meta)data management from remote servers to client machines in order to increase locality and leverage the speed of NVRAM. Our model also captures how data can be managed across different device types efficiently.

However, with Assise's new model, we find that the CPU burden is increased on client machines as they now need to manage (meta)data. We explore how this may be ameliorated by offloading these tasks to Network Interface Cards (NICs). In ***RedN***, we unlock the computational power of commodity Remote Direct Memory Access (RDMA) NICs by turning them into universal Turing machines. In doing so, we effectively transform RDMA NICs into general-purpose CPUs. We leverage this to offload common storage tasks to the NIC, allowing us to accelerate storage performance while saving CPU cycles. We then discuss how RedN can be integrated with Assise to reduce its CPU footprint on client machines, allowing for better scalability.

In the next sections, we expand on the two main parts of the thesis.

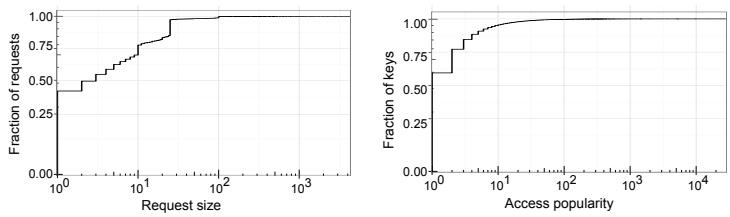
1.2 Adapting to Workload & Network Trends

Large-scale web services face the challenge of providing low and predictable latencies for user requests. These latencies can be inflated due to skews in both user workloads as well path changes in the cloud provider network. We discuss each of these elements below and how storage system scheduling policies can be adapted to reduce latency inflation.

Workload skews. One of the sources of performance fluctuations is skewed workloads—which are commonly seen in practice [1, 2, 3, 4, 5]. This is due to the predominance of power law distributions in social systems, which also manifests in workload patterns. For example, a music streaming service might see its most popular songs receiving orders of magnitude more traffic than all other songs combined. To examine workload heterogeneity, we analyze a trace gathered from the production cluster of a popular music streaming service—SoundCloud. This trace records *get* requests sent to the backend systems that ask for a value corresponding to a provided key; multiple keys can be requested within a single *get* request. We analyze two properties of this workload—key access frequency and the number of keys per request (*i.e.*, the request size). Figure 1.2b shows

the distribution of key access frequency. We can clearly observe a heavy-tailed distribution—where the majority of the keys are accessed only once, whereas a few are accessed up to 1000 times. Figure 1.2a shows the distribution of request sizes. Similarly, we also observe a heavy-tailed distribution— ∼40% of requests involve more than a single key; the average size is 8.6 keys and the maximum is as high as ∼2000 keys.

We find that these workload skews can cause hotspots and queueing delays in the underlying storage systems—resulting in inflated latencies and possibly causing service disruptions. In our first project—Rein [6]—we set out to deal with these issues by developing scheduling heuristics that leverage information about request characteristics to execute them in a latency-aware manner. Next, we look into performance hot spots caused by path changes.



(a) Distribution of request sizes.

(b) Distribution of key popularity.

Figure 1.2: Multiget requests from a production trace exhibit significant variations in size (a) and key popularity (b).

Path changes in Cloud Networks. For services hosted on the cloud, operators also need to consider the network characteristics of the cloud provider's network. Despite the common belief that routing changes are infrequent in such networks, our study [7] in one of the largest cloud provider networks, has shown that this is not universally true. We find that network latencies between different geographic regions change frequently and unpredictably, as a result of traffic engineering. This is typically done by the cloud provider to allow their network resources to be used more efficiently. However, given the resulting latency fluctuations, this goal may not align with the needs of the customer.

To show these latency changes, we unpack a simple experiment conducted on Amazon *AWS*. In this experiment, we established three TCP connections between two machines located in two different Amazon *AWS* regions — namely Oregon and Virginia. For each of these connections, we periodically generate ping messages and plot their latencies in Fig. 1.3. We observe that each connection experiences different latencies, despite sharing the same machines. Moreover, the latencies

on these connections can change suddenly in short time-scales (order of 10s of seconds). Such effects can negatively impact the latencies storage systems deployed across multiple regions. To reduce these negative effects, one can exploit information about the network latency to further augment Rein's scheduling policies and provide more latency-aware scheduling.

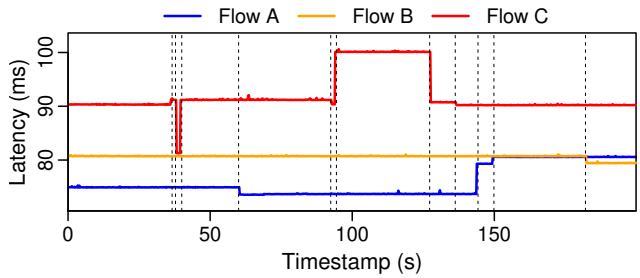


Figure 1.3: TCP RTT measured between two Amazon VMs deployed in Oregon & Virginia. The black dotted vertical lines highlight moments when RTT latency changed and indicate potential Traffic Engineering activity.

1.3 Leveraging State-of-the-Art Hardware

The increasing diversity of server hardware offer several distinct opportunities for storage systems. We argue that, to leverage the capabilities of newer hardware technologies, storage systems need to be redesigned to provide better integration between software and hardware. We look below into these new trends.

Rise of NVRAM. Storage hardware also plays a key role in the performance of distributed storage systems. With the advent of non-volatile random-access memory (NVRAM), we now have an even more diverse selection of storage devices to choose from. NVRAM provides DRAM-like performance with higher capacity and lower cost, while allowing data to persist. Table 1.1 shows measured IO latency, bandwidth, and cost of modern server memory and storage technologies. As we can see, NVRAM's IO latencies is up to two orders of magnitude better than SSDs but its price is at least 10× higher. For distributed storage, remote NVRAM can be accessed quickly using RDMA (NVM+RDMA in Table 1.1) given that it bypasses both the server CPU and network stack. As such, NVM+RDMA can provide lower latencies compared to even accessing SSDs locally.

To take advantage of the high IO performance of NVRAM, we propose a new distributed file system model in Assise. This model pushes (meta)data management

from remote servers to client machines in order to improve locality and leverage the higher NVRAM speeds. Assise also allows us to manage data across different storage devices in a manner that is transparent to the user, forgoing the complexity of setting up different file system software for each device type. This new model, however, is not a panacea. It places an increased burden on client CPU, prompting us to look for ways to offload Assise's storage tasks. We explore this next.

Memory	R/W Latency	Seq. R/W GB/s	$/GB
NVRAM	175 / 94 ns	32 / 11.2	3.83 [8]
NVRAM+RDMA	3 / 8 μs	3.8	-
SSD	10 μs	2.4 / 2.0	0.32 [9]

Table 1.1: Memory & storage price/performance (October 2020).

NICs as processors. RDMA has become ubiquitous [10] due to providing low latency access to remote server memory, while allowing the user to bypass server CPU. This allows users to offload work from the CPU, saving precious cycles of an increasingly scarce resource. To cope with higher packet rates, RDMA NIC (RNIC) processing power has effectively doubled with each subsequent generation, as can be seen in Table 1.2.

Despite this, RNICs cannot perform more complex offloads since we are limited by the RDMA API— which only allows simple remote memory reads and writes. In RedN [11], we make the discovery that despite these limitations, RNICs are, in fact, Turing complete. We then develop ways to combine simple RDMA operations to implement more advanced constructs (*e.g.*, loops and conditional).

In doing so, RedN effectively turns RNICs into Turing machines—capable of performing arbitrary computations— and provides yet another compute resource. In leveraging this resource, we show how we can offload several common tasks used in storage systems (*e.g.*, hash table lookups) —allowing us to reduce latency, provide failure resiliency/performance isolation, and save CPU cycles.

RNIC	**PUs**	**Throughput**
ConnectX-3 (2014)	2	15M verbs/s
ConnectX-5 (2016)	8	63M verbs/s
ConnectX-6 (2017)	16	112M verbs/s

Table 1.2: Number of Processing Units (PUs) and performance of ConnectX NICs.

1.4 Thesis Statement

This thesis explores various types of heterogeneity in distributed systems — namely heterogeneity in user workloads, network infrastructure, and server hardware — and provides insights/blueprints for designing storage systems under such settings.

The focus of this thesis is to provide both fast and predictable response times by: (i) adapting storage system scheduling policies to workload and network state and, (ii) by redesigning their software components to leverage state-of-the-art storage and network hardware.

1.5 Contributions and Outline

This section outlines the thesis' structure. Chapter 2 provides the necessary background and motivation. After this, the main works of this thesis are presented in two parts, each addressing a component of the thesis statement. Part I (Ch. 3 & 4) focuses on providing new scheduling algorithms for storage systems that can adapt to workload and network heterogeneities. Part II (Ch. 5 & 6) focuses on providing better integration between storage software and state-of-the-art hardware technologies. The contributions of the individual chapters are as follows:

- **Chapter 3**: I provide an analysis of skews in production workloads and study their impact on storage systems. Based on this, I devise several scheduling heuristics for reducing latency by exploiting information about request characteristics.
- **Chapter 4**: I perform a measurements study on the latency variations across different regions in one of the largest cloud networks, showing that these variations occur very frequently. I discuss implications for storage systems and how to leverage this information to improve scheduling decisions.
- **Chapter 5**: I provide a design blueprint and implementation of a distributed file system that manages NVRAM with different types of storage devices, providing a multi-layer caching hierarchy. For low latency access to remote storage (*e.g.*, for replication), RDMA is built into the system and combined with an efficient concurrency control mechanism to provide high scalability.
- **Chapter 6**: I design and implement a framework that converts commodity RDMA NICs into Turing machines—capable of performing arbitrary computations, all without requiring any hardware modifications. I demonstrate the effectiveness of this framework in fully offloading common storage tasks (*e.g.*, hash lookups) and accelerating real applications like memcached.

After covering the main works, we discuss required reflections in Chapter 7. The thesis finally concludes in Chapter 8.

1.6 Previously Published Material

The work in this thesis has appeared in previous publications:

- Chapter 3 revises a previous publication [6]: **Waleed Reda**, M. Canini, L. Suresh, D. Kostić, and S. Braithwaite, "Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling," in *12th European Conference on Computer Systems (EuroSys).* ACM, 2017.

- Chapter 4 is adapted from [7]: **Waleed Reda**, K. Bogdanov, A. Milolidakis, H. Ghasemirahni, M. Chiesa, G. Q. Maguire Jr, and D. Kostić, "Path Persistence in the Cloud: A Study of the Effects of Inter-Region Traffic Engineering in a Large Cloud Provider's Network," *ACM SIGCOMM Computer Communication Review*, 2020.

- Chapter 5 revises a previous publication [12]: T. E. Anderson, M. Canini, J. Kim, D. Kostić, Y. Kwon, S. Peter, **Waleed Reda***, H. N. Schuh, and E. Witchel, "Assise: Performance and Availability via Client-local NVM in a Distributed File System," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.
  *** Lead student author**

- Chapter 6 is adapted from [11]: **Waleed Reda**, M. Canini, D. Kostić, and S. Peter, "RDMA is Turing complete, we just did not know it yet!" in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

1.7 Other Publications

This is a list of other publications that are not part of this thesis:

- J. Kim, I. Jang, **Waleed Reda**, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel. "LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism." in *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, 2021.

- K. Bogdanov, **Waleed Reda**, G. Q. Maguire Jr, D. Kostić, and M. Canini. "Fast and Accurate Load Balancing for Geo-Distributed Storage Systems." in *ACM Symposium on Cloud Computing (SoCC)*, 2018.

Chapter 2

Background

Over the next few sections, we will cover the background for this thesis. We start with a description of the typical cloud and datacenter environments and the different sources of heterogeneities that face services deployed in the cloud (Section 2.1). We then provide background on state-of-the-art storage and network hardware in sections 2.2 and 2.3, respectively. This is followed by a small survey on programmable network hardware, including FPGAs, System-on-Chip NICs, amongst others (Section 2.4). Next, we discuss the importance of having predictable performance in the cloud (Section 2.5). Lastly, we cover related works in the distributed storage area, and highlight their limitations (Section 2.6).

2.1 Cloud Environment

Cloud services have become increasingly popular as the computational and storage requirements have increased beyond what can be provided by the individual business's infrastructure. Cloud providers manage one or more datacenters, which are large server farms. In doing so, Cloud providers benefit from economies of scale allowing them to reduce capital and operational costs. Cloud providers can then rent their resources to private or corporate customers that use these services — which we denote as Service Operators. Cloud providers can rent out their resources as virtual machines and/or bare-metal machines with dedicated hardware.

Cloud infrastructure encompasses many elements, including server hardware, software, and the network, all of which are managed by the cloud provider. The cloud environment encompasses the infrastructure as well as the ways users interact with the cloud — including their workload characteristics.

Heterogeneity exists across these different elements (workload/network/hard-

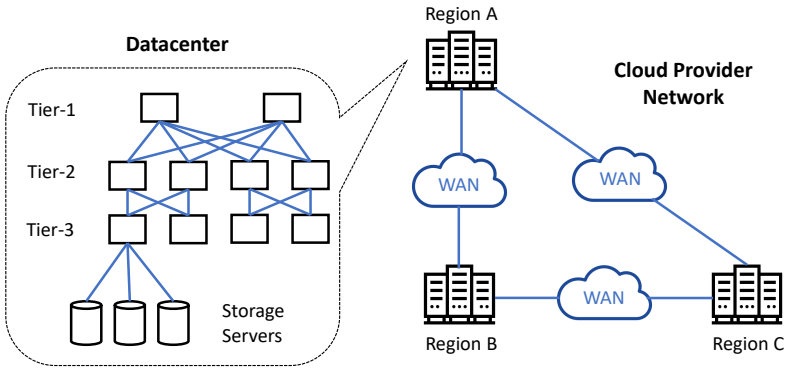


Figure 2.1: Cloud provider network architecture. Regions can be composed of multiple datacenters which are interconnected either via a public or private Wide Area Network (WAN).

ware) and service operators need to take them into account as they can affect the Quality of Service (QoS) of their applications. We next describe these different sources of heterogeneity in detail.

Workloads. Research has shown [13] that, in popular Online Social Networks (*e.g.*, Flickr and Youtube), user interactions usually follow a power-law distribution where a small cluster of *super-users* are much more popular in the network. For example, on Youtube, a small subset of channels can get orders of magnitude more subscribers than the remaining channels. As such, in practice, user workloads in cloud services are typically skewed [1, 2, 3, 4, 5]. For distributed storage systems, these skews can manifest in the data object popularity (*i.e.*, how frequently certain objects are accessed), IO granularities, request sizes, *etc.*. This phenomena can introduce performance hotspots and impact the QoS.

Network. The three largest cloud providers worldwide (e.g., Amazon *AWS*, Microsoft Azure, Google Cloud Platform) deploy their networks using a similar hierarchical structure. This architecture is depicted in Figure 2.1. Within each cloud provider, a set of typically more than 10 *regions* are interconnected by a globally-deployed, singly administered, Wide Area Network (WAN). All of the regions are subdivided into availability zones, each consisting of at least one datacenter network. The datacenter is connected by a Local Area Network (LAN) which commonly uses a Fat-tree topology [14]. Operators can deploy their services

within the same region or across different regions. In the latter case, high-frequency oscillations in inter-region latencies on the WAN may impact performance — as was observed in Microsoft's network [15].

Storage. Cloud providers provide different types of storage technologies on their instances such as Solid-State Drives (SSDs) and Harddisk Drives (HDDs). The diversity of storage options have recently increased with the adoption of Non-Volatile Random Access Memory (or NVRAM). NVRAM provides much higher access speeds than traditional storage, which calls into question existing designs of distributed storage systems. Moreover, given this increased diversity, storage systems need a way to manage data across different types of storage.

Compute. Similar to storage, cloud servers can also be equipped with different models of CPUs and GPUs, each with their own processing capabilities. Moreover, the recent surge in programmable network devices provides yet another compute resource. Application logic can now be offloaded to network hardware as such System-on-Chip NICs [16], which come with programmable micro-CPUs. We discuss programmable network devices in more detail in section 2.4.

2.2 Non-Volatile Random Access Memory

Non-Volatile Random Access Memory (NVRAM) is now commercially available. Phase-shift memory [17] RAM like Intel's Optane DC persistent memory module (PMM) [18] were available as early as 2019. Other NVRAM technologies such as TAS-MRAM [19] and STT-MRAM [20] are under development.

NVRAM is comprised of non-volatile DIMMs that are attached to a CPU's memory bus, similar to DRAM DIMMs. NVRAM provides many benefits compared to DRAM as well as traditional storage media. We highlight these benefits below.

Byte-addressability. NVRAM like Intel's Optane DC PMM allows data to be loaded or stored at 256-byte granularity. This means that small IO can be performed efficiently compared to traditional storage devices like SSDs and HDDs which access data at much higher granularities (4 KB or higher). Writing 256 bytes to NVRAM takes just 94 ns [12], only 15% slower than DRAM and orders of magnitude faster than traditional storage media.

Low-cost memory. At the time of this writing, a 128GB Optane DC DIMM costs approximately 3.8 dollars per gigabyte, which makes it 40% cheaper than a

same-sized DRAM DIMM [12]. This means that NVRAM can be used as a cheap substitute for DRAM, which is helping drive their adoption at scale [21, 22, 23].

Non-volatile. As opposed to DRAM, NVRAM is non-volatile. It persists data even during shutdowns and crashes. This makes it suitable for NVRAM to be utilized as a storage medium.

NVRAM can be used like traditional memory and appear as ranges of physical memory addresses. Data can be directly accessed on the local machine via a load/store interface. For distributed storage, direct access to remote NVRAM can be accomplished using a technology called Remote Direct Memory Access (RDMA) [24]. We cover the basics of RDMA in the next section.

2.3 Remote Direct Memory Access

Distributed applications typically need to move data across nodes. This can often be on the critical path of these applications. As such, implementing this in an efficient manner is critical. With NVRAM in the picture, this becomes ever so important, since they provide low IO latencies. As such, to take advantage of their speed, network and software overheads need to be minimized. Remote Direct Memory Access (RDMA) allows direct access to remote machine memory, including NVRAM. This offers many advantages, as it allows us to bypass the OS kernel, does not involve server CPU, and eliminates unnecessary copies to/from network buffers. We discuss these advantages in-depth below.

Kernel bypass. RDMA can be used by applications directly in userspace, which allows it to bypass the operating system kernel entirely. This reduces context switch overheads and improves performance.

No CPU involvement. Applications can directly read and write remote memory using RDMA without any server CPU involvement. In regular socket-based transfers, the CPU needs to move data to/from network buffers, therefore incurring additional cycles .

Zero-copy. RDMA can be used to directly read or write data remotely without copying data to/from network buffers. This removes unnecessary memory transfers, improving performance and efficiency.

Due to the aforementioned benefits, RDMA is able to speed up access to remote memory. This makes it a good fit for latency-sensitive applications that store their data on DRAM or NVRAM.

2.3.1 How does RDMA work?

The RDMA interface specifies a number of data movement verbs (READ, WRITE, SEND, RECV, etc.) that are *posted* as *work requests* (WRs) into *work queues* (WQs) in host memory. The mechanism for executing these verbs is summarized in Figure 2.2. The RNIC starts execution of a sequence of WRs in a WQ once the offload developer triggers a *doorbell* (❶) —a special register in RNIC memory that informs the RNIC that a WQ has been updated and should be executed. After receiving the doorbell, the NIC would then fetch the headers of the WR first (❷), which contain the WR's metadata. This is done via a Direct Memory Access (DMA) operation. After fetching the WR headers, the NIC then retrieves the WR itself (❸). After fetching the WR, the NIC can then proceed to execute it (❹), and later receive an acknowledgement after the WR completes (❺). These steps describe the general workflow for executing an RDMA operation, however, some of them can be skipped entirely. For instance, ConnectX NICs [25] can allow the user to send the WR in the same operation as the doorbell. This may reduce latency, since it removes the need for the NIC to fetch the WR header and body via additional DMA operations. However, this comes at the cost of additional CPU cycles, since the CPU is actively involved in ringing the doorbell. Similarly, acknowledgements can be disabled if the application does not need to be notified of completions. This can help reduce NIC overheads, since it no longer needs to send a completion notification via a DMA.

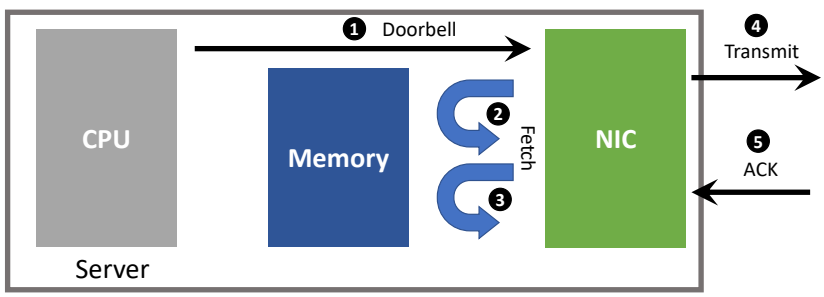


Figure 2.2: The RDMA mechanism for executing operations can be broken down into five main steps. The CPU first starts the execution via a doorbell operation (❶), which causes the WR to be fetched (❷) (❸). Lastly, the WR is executed (❹) and an acknowledgement is received (❺).

2.4 Programmable Network Hardware

Recent advances in programmable network hardware provide an opportunity for distributed storage systems. It allows system developers to co-design applications with the network. The increase in network programmability comes from various solutions, including (from most to least programmable) System-on-chip NICs [16], FPGA-basd NICs [26] and RDMA NICs [25]. In this section, we provide background on the different types of programmable NICs.

2.4.1 System-on-Chip NICs

System-on-Chip (SoC) NICs (e.g. Mellanox BlueField [16]) are network adapters that provide extended processing capabilities using a micro-CPU with dedicated memory. This CPU is typically connected to an internal switch inside the NIC via PCIe. SoC NICs are highly programmable and allow users to run even full-fledged operating systems on the NIC's cores. However, this comes with both an increased cost and a performance overhead. The latter is primarily due to packets needing to traverse the PCIe interconnect to be handled by the NIC's CPU. In addition, these CPUs are typically "wimpy" providing much lower clock rates than server CPUs (800 Mhz for Mellanox's first-generation BlueField [16]).

2.4.2 FPGA-based NICs

Field Programmable Gate Arrays (FPGAs) are programmable logic devices that are based on a matrix of re-configurable logic blocks. Programs running on FPGAs are typically written using hardware programming such as Verilog and VHDL which require special skillsets to use and incur longer development time. To simplify the task of writing programs, vendors and the open-source community have introduced high-level synthesis compilers [27] — which convert C-like language to code optimized for FPGAs. However, these compilers come with their own set of challenges and can result in significant resource bloat [28].

FPGAs have been used to provide NICs with increased programmability and augment their capabilities [29, 30]. Similar to SoC-based approaches, these solutions are also more expensive than commodity devices.

2.4.3 RDMA NICs

Remote Direct Memory Access (RDMA) [24] has become ubiquitous [10]. Mellanox ConnectX NICs [25] have pioneered ubiquitous RDMA support and Intel has added RDMA support to their 800 series of Ethernet network adapters [31]. RDMA focuses on the offload of simple message passing (via SEND/RECV

verbs) and remote memory access (via READ/WRITE verbs) [24]. Both primitives are widely used in networked applications. However, RDMA is not designed for more complex offloads that are also common in networked applications. For example, remote data structure traversal and hash table access are not normally deemed realizable with RDMA [32]. As such, it does not offer the same level of programmability as System-on-chip and FPGA-based NICs. This led to many RDMA-based systems requiring multiple network round-trips or to reintroduce involvement of the server's CPU to execute such requests [33, 34, 35, 36, 37, 38].

2.5 Performance Predictability

We now discuss the importance of performance predictability for distributed storage systems. Inside the cloud, there are several types of applications that can be deployed, including but not limited to, AI & machine learning, big data analytics, and latency-critical applications that need to have bounded latencies. In this thesis, we focus on the latter type of applications.

Cloud providers often provide guarantees that are stipulated in a legal contract and can be expressed as a latency bound for a certain percentage of the requests. If these terms are not satisfied, the cloud provider may need to reimburse the service operator according to the terms of their contract. To provide predictable response times, distributed storage systems need to have control over the tail of their latency distribution.

Distributed storage systems typically follow the datacenter scale-out design [39], where they are deployed across different servers. This allows them to partition their dataset and use the collective resources of these machines. As such, clients that are accessing data, for example to render a web-page for an online service, may need to access multiple servers as seen in Figure 2.3. This communication pattern — where an initial request is broken down into sub-requests — is called scatter-gather and is very common in datacenter workloads [40]. In this case, the responses are collected from several servers and sent back to the client.

Tail at scale problem. Applications using the scatter-gather pattern of communication suffer from a problem known as the *tail at scale*. There is inherent variability in the response times of storage servers, which can be caused by the different sources of heterogeneity in the cloud environment (as described in section 2.1). In many cases, client requests are not considered complete until all the remaining sub-requests have also been completed. As such, the response time is therefore equivalent to the slowest request to finish. The higher the number of sub-requests, the higher the probability of experiencing such slowdowns. For example,

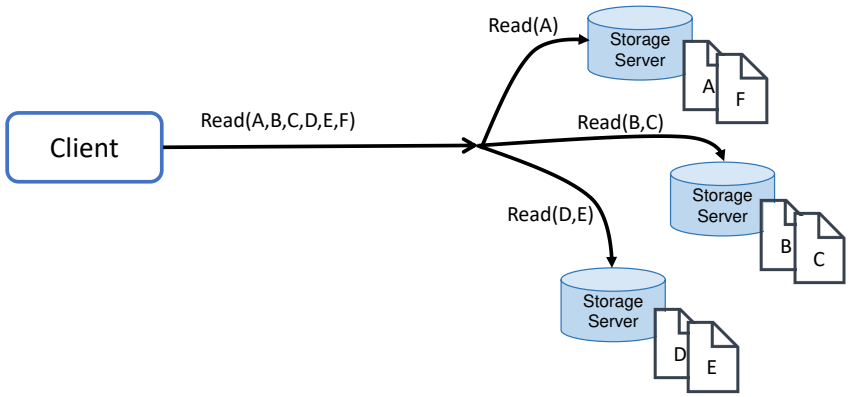


Figure 2.3: Client requests for online services may require sending 10s to 1000s of subrequests to different servers and collecting their responses. This communication pattern is quite common and is known as scatter-gather.

a request that generates 1000 sub-requests, will likely have the same end-to-end latency as the 99.9^{th} slowest sub-request.

Service Level Agreements (SLAs) are contracts between cloud providers and the end-user. These legal documents define all the responsibilities and level of service expected from the cloud provider. They lay out metrics by which the service is being assessed, along with any penalties should the agreed-upon service levels not be achieved.

The metrics used within an SLA are referred to as Service Level Objectives (SLOs). One commonly used SLO is setting a latency-bound for a certain percentage of requests. Other metrics include the service's availability or up-time which can be computed by measuring the percentage of time a service was operational over a given period.

2.6 The Need for Fast Distributed Storage

We outline why cloud applications have a high demand for high-performance distributed storage, current storage and networking hardware trends that enable it, and why other approaches to distributed storage fall short of providing it.

2.6.1 Lack of workload & network awareness

Several studies [39, 41] have shown that latency distributions in Web-scale systems exhibit long-tail behaviors where the 99.9^{th} percentile latency can be more than one order of magnitude higher than the median latency. As mentioned earlier, workload skews can cause performance hotspots and inflate latencies [6, 42]. There have been many proposals for achieving latency reduction and lowering the impact of skewed performance, which we can broadly categorize in three categories. The first includes work that propose system techniques such as duplicating or reissuing requests, predicting stragglers, or trading off completeness for latency [39, 41, 43]. The second consists of works that allocate resources according to well-established policies (e.g., some notion of fairness) to guarantee certain SLOs [44, 45]. In some cases, these works use admission control to refuse requests that would violate SLOs. The third class of works approaches the latency reduction problem of from the load-balancing perspective [42, 46, 47].

However, none of these works exploit information about the workload/network but mainly aim to treat the symptoms caused by their skews. We argue that workload and network latency information can be incorporated by distributed storage systems to provide performance gains.

2.6.2 Weak support for NVRAM

Distributed applications have diverse workloads, with IO granularities large and small [48], different sharding patterns, and consistency requirements. All demand high availability and scalability. Supporting these properties simultaneously has been the focus of decades of distributed storage research [49, 50, 51, 52, 53, 54, 55]. Before NVRAM, trade-offs had to be made. For example, by favoring large transfers ahead of small IO, or steady-state performance ahead of crash consistency and fast recovery, leading to the common idiom of remote-storage file system design. We argue that with the arrival of fast NVRAM, these trade-offs need to be re-evaluated.

Block stores. These include systems such as Amazon's EBS [56] and S3 [57] which use a multi-layer storage hierarchy to provide cheap access to vast amounts of data [48]. However, block stores have a minimum IO granularity (16KB for EBS) and IO smaller than the block size suffers performance degradation from write amplification [48, 58].

Traditional distributed file systems. A notable example of such systems is Ceph [49] which uses consistent hashing over data and metadata to provide scalable file service for cloud applications. However, remote access for data harms

performance. While such systems can support small IO more efficiently than block stores, their promise of higher throughput via parallel access to disaggregated storage is surpassed by the up to 8× higher bandwidth of local NVM.

To combat network overheads, several disaggregated file systems have been built [50, 51] or retrofitted [55, 59, 60] to use RDMA. For example, Orion [51] leverages local NVM for data storage, but it still disaggregates metadata management. However, local NVM has *both lower latency and higher bandwidth* than the (RDMA) network and these file systems still incur network overhead for each access. The high network latency and limited bandwidth increases file system operation latency, reduces throughput, and limits scalability.

2.6.3 In-Network offloads are still limited

Remote Direct Memory Access (RDMA) NICs provide a way for accessing remote memory efficiently with high throughput and low latency. This benefit stems from their ability to bypass the OS kernel, avoid complex network stacks, and access memory without the involvement of the remote CPU. However, despite these benefits, their APIs are rather limited, allowing only for reads and writes to remote memory using either *one-sided* or *two-sided* verbs. Remote Procedure Calls (RPCs) act as a building block for networked services but without support for more complex offloads, their performance can significantly degrade if they are not properly integrated with RDMA. Typically, RDMA-based systems [33, 34, 35, 36, 37, 38, 61] require multiple round-trips or direct involvement of the server's CPU to execute an RPC. This incurs additional latency overheads and can introduce scalability challenges for services that handle thousands of clients.

To workaround these issues, several proposals have been made to augment RDMA NICs [29, 62, 63]. System-on-Chip NICs (e.g. Mellanox BlueField [16]) use a micro-CPU that allows "slow-path" offloads which, while flexible, incur additional latencies due to CPU and PCIe overheads. On the other hand, FPGA-augmented NICs offer a great deal of flexibility but are notoriously difficult to program.

All the aforementioned solutions suffer from at least one big limitation that prevent in-network offload solutions from being widely adopted.

Part I

Adapting to Workload & Network Trends

Chapter 3

Taming Latency in KV Stores via Multiget Scheduling

For modern cloud services, service level objectives are becoming increasingly stricter. These objectives are often defined in terms of a percentile of the latency distribution (such as the 99.9^{th}-ile [64]). Thanks to scale-out designs and simple APIs, key-value storage systems have emerged as a fundamental building block for creating services capable of delivering microseconds of latency and high throughput.

Moreover, for interactive Web services, requests with a high fan-out that parallelize read operations across many different machines are becoming a common pattern for reducing latency [1, 2, 4, 5]. These requests, which batch together access to several data elements, typically generate tens to thousands of operations performed at backend servers, each hosting a partition of a large dataset [65]. This makes reducing latency at the tail even more imperative because the larger a fan-out request is, the more likely it is to be affected by the inherent variability in the latency distribution, where the 99^{th}-ile latency can be more than an order of magnitude higher than the median [39, 41].

To support this request pattern, many popular storage systems, including wide-column stores (e.g., Cassandra [66]) and document stores (e.g., Elasticsearch [67]), offer a `multiget` API. A multiget request allows multiple values at the specified keys to be retrieved in one request. Batching multiple read operations together is typically far more efficient than making several individual single-key requests.

Multiget requests provide opportunities to reduce latency via scheduling. These requests follow the "all-or-nothing" principle, where a multiget only finishes once all of its operations are completed and the last operation's response is received. In

practice, multiget requests in real-world workloads present variability in attributes like size, processing time, and fan-out as well as load imbalances and accessed keys (§3.1). Such variability means that the response time of different requests is likely affected by different bottleneck operations across different servers. Our insight is that it is possible to improve latency by scheduling operations while considering what bottleneck affects the overall request completion time, and "slacking" non-bottleneck operations for as long as they cause no extra delay to the request. Slacking certain operations can allow other operations with tighter bottlenecks to be serviced earlier and thus decrease aggregate response times as well as latencies at the tail.

Scheduling operations of multiget requests across cluster nodes of a storage system is challenging. Even when a priori knowledge of all operations to be scheduled and their service times is assumed, the problem can be expressed as a concurrent open-shop scheduling problem, which is known to be NP-complete [68]. Moreover, no single scheduling algorithm provides optimal ordering for reducing aggregate response times. For heavy-tailed distributions of request sizes, policies such as Shortest Job First (SJF) help to reduce average makespans (or execution times), but for light-tailed distributions. First Come First Served (FCFS) can become the optimal scheduling strategy [69]. It has also been proven that there is no non-learning policy that can optimize scheduling across a wide variety of workloads [70].

In practice, the problem is even more challenging because operation completion times are not known in advance due to several factors, including variable service rates, server load imbalances, skewed access patterns, etc. In addition, these operations are meant to have very low latency and are typically serviced in parallel across a number of different servers. As such, an efficient scheduling algorithm has to operate on multiple uncoordinated servers with minimal overhead. Thus, this problem is distinct from scheduling batch jobs in big data systems (*e.g.*, [71]) in which a centralized scheduler can make coordinated decisions.

To address these challenges, we introduce Rein (§3.2), a multiget scheduler that leverages variability in the structure of multiget requests to reduce the median as well as tail latency. We first devise a method for predicting bottleneck operations within requests, and, based on these predictions, we employ a combination of two policies to schedule requests in an efficient manner (§3.2.2). The first policy – Shortest Bottleneck First (SBF) – prioritizes requests with smaller bottlenecks, which reduces head-of-line blocking and improves average makespans. The second policy – Slack-Driven Scheduling (SDS) – allows us to use resources efficiently by delaying non-bottleneck operations without impacting the overall request completion time. The priorities computed by the aforementioned policies are assigned at the client-side and then enforced at the backend nodes responsible

for servicing the requests. We leverage both policies to schedule requests in a manner that is optimized for both median and higher percentiles of the latency distribution. We then provide a blueprint for how to enforce our scheduling order in highly concurrent systems by using a novel scheduling approach called multi-level queues (§3.2.3), which combines the advantages of classic scheduling algorithms while yielding an efficient implementation. Our technique achieves a low-overhead implementation by adopting non-blocking FIFO queues that significantly reduce contention among threads.

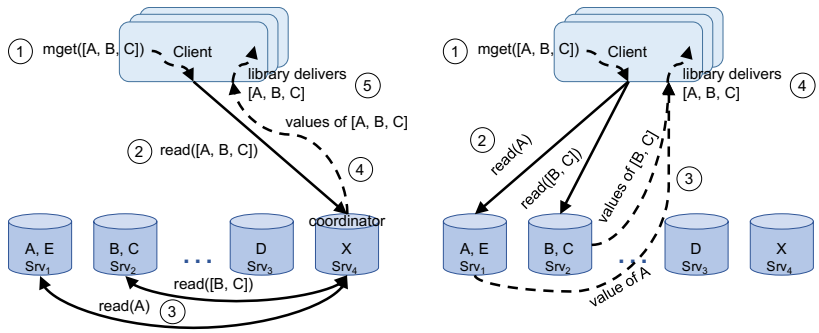
We demonstrate the feasibility and benefits of our approach by instantiating our prototype implementation (§3.3) within Cassandra, a popular distributed database. Through empirical evaluation (§3.4.1) and simulations (§3.4.3) using both production and synthetic workloads, we show that Rein can reduce the median latency and 95^{th} percentile by 1.5 times and the 99^{th} percentile latency by 1.9 times compared with multiget-agnostic, FIFO scheduling. We also compare our approach with other state-of-the-art techniques and find that Rein consistently outperforms the others under varying system conditions. We expect that our results will inspire the design of new multiget-aware scheduling algorithms achieving even greater improvements. Our Rein implementation is available as open source.*

3.1 Background

Before we describe how Rein operates, we first describe the `multiget` abstraction. Multiget requests are a common idiom employed in modern cloud services to use storage systems efficiently. Many popular systems, including Redis [72], Memcached [73], Cassandra [66], MongoDB [74] and Elasticsearch [67], offer support for multiget requests. Their usage is simple. Developers write client-side code that batches access to multiple values as a single request. For example, `mget(A, B, C)` requests values of keys A, B, C. Their execution dependes on the system, as we illustrate in Figure 3.1. For systems like Cassandra (Figure 3.1a), the client-side library sends the multiget request to one node of the cluster (referred to as the coordinator, e.g., Srv_4), which is responsible for reading all values across cluster nodes and combining them into a single response to be sent back to the client. In other systems like Memcached (Figure 3.1b), the client-side library itself could obtain the response by parallelizing data fetches across cluster nodes, each of which receives only (batched) read operations for keys that fall in the data partition it serves.

Regardless of the implementation, as data is typically partitioned (and replicated) across cluster nodes, the likelihood that a multiget request would require fetching values distributed among nodes increases with cluster size and request

* Code at https://github.com/wreda/cassandra-rein.



(a) Multiget via coordinator.

(b) Multiget in client library.

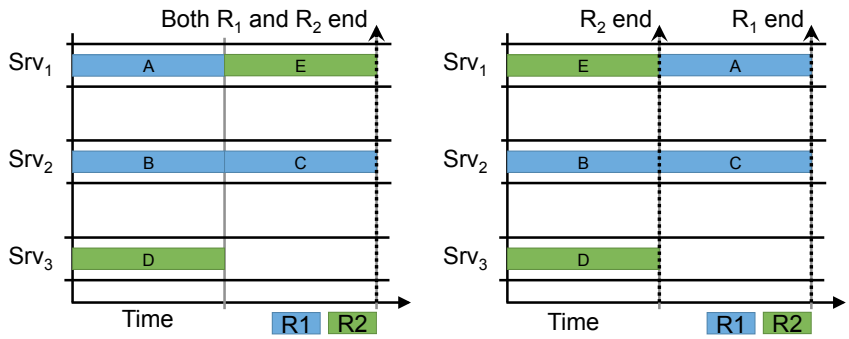
Figure 3.1: Common styles of multiget requests in partitioned key-value stores.

size (i.e., number of requested keys). The resulting workloads consist of fan-out operations across nodes. Moreover, these workloads are likely to exhibit significant variations as key-value workloads for cloud applications are often skewed [75, 76], multiget requests vary in size and processing time [4], and individual servers are exposed to performance variability (e.g., due to resource contention) [39, 42].

3.1.1 Benefits of Multiget Scheduling

Variations in the structure of multiget requests give rise to possible performance improvements through inter-multiget scheduling. We illustrate these with a simple example — shown in Fig. 3.2. Assume that two multiget requests, R_1 and R_2, arrive at the same time: one request is for three keys, A, B, C and the second is for two keys, D, E. Given how the data are partitioned, request R_1 is broken down into two sets of operations (called *opset*s) to read the values of A and B, C from servers Srv_1 and Srv_2, respectively.

What will the response times of these requests be? For presentation sake, assume that every server serves each operation with a service rate, μ, of one operation per unit of time (*e.g.*, 1 op/ms). This means that serving the B, C opset will last two units of time (we omit network latencies for now). We call the opset with the longest response time the *bottleneck opset* (which is B, C in this example). A multiget response time depends on its bottleneck's completion time. Thus, assuming no queuing, R_1 will complete in two units of time whereas R_2 will complete in one unit of time thanks to parallelization across two servers. However, when R_1 and R_2 execute concurrently, operations might happen in the order A, E. With this schedule, both requests complete in two units of time as



(a) Multiget-oblivious schedule.

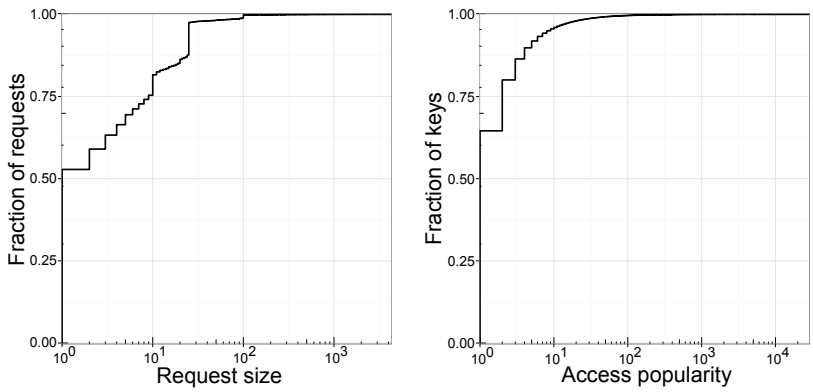
(b) Multiget-aware schedule.

Figure 3.2: Performance of requests using a multiget-oblivious (a) and a multiget-aware (b) schedule. Multiget-aware scheduling reduces average response time.

shown in Figure 3.2a. This is because servers are oblivious to the structure of multiget requests.

Would processing requests in a multiget-aware fashion lead to benefits? Because a multiget request is complete only once all its operations complete, there is some *slack* for accessing A — in particular, to cause no extra delay to R_1, the access to A can be postponed for as long as it completes with a delay of up to two units of time. Given this information about the global deadline of a multiget request, a server could serve operations to meet their deadlines while minimizing the delay of other operations. In our example, server Srv_1 can perform the read operation for E before the one for A. With this optimal schedule, the completion time of R_2 is just one unit of time and the average response time minimized to 1.5 as shown in Figure 3.2b.

Underpinning these performance improvements are the variations in multigets' attributes like size, processing time, and fan-out as well as the load imbalances across cluster nodes and other common skews of key-value store workloads, such as heavy-tailed key access frequency. In principle, the higher the variance of these factors, the greater the opportunity for slack-driven latency reductions, in particular at the tail of the latency distribution. Using a trace from a production cluster at SoundCloud, we next highlight the variations in request sizes and workloads and then quantify the performance improvements that these variations may yield in practice.



(a) Distribution of request sizes.

(b) Distribution of key popularity.

Figure 3.3: Multiget requests from a production trace exhibit significant variations in size (a) and key popularity (b).

3.1.2 Analysis of Multiget in Production

We focus on understanding the characteristics of multiget requests by analyzing a 30-minute trace from a production cluster at SoundCloud gathered during its operation. We highlight that the structure of multiget requests exhibits significant variations in size and key popularity. While this single workload does not generalize to all systems and environments, we note that similar properties were observed in other environments [1, 2, 3, 4, 5]. These insights motivate and inform Rein's design.

Figure 3.3a plots the distribution of multiget request sizes. The request sizes show a heavy-tailed distribution: ∼40% of requests involve more than a single key; the average size is 8.6 keys and the maximum is as high as ∼2000 keys.

Figure 3.3b plots the distribution of key access frequency. This distribution is also heavy-tailed. Most keys are accessed just once in the collected trace, whereas a few keys are accessed up to 1000 times.

The values being accessed in this trace are of fixed size. This allows us to validate that there exists a positive correlation between a multiget size and its response time. However, as we were limited in the intrusiveness of our instrumentation, we are not able to report on the variation in performance from the production cluster. To quantify the variation in performance across multiget requests and to provide figures in the context of the results of this work, we use the trace to benchmark our Cassandra testbed composed of 16 AWS EC2 m3.xlarge nodes (our setup is detailed in §3.4). We generate our workload at 75% utilization

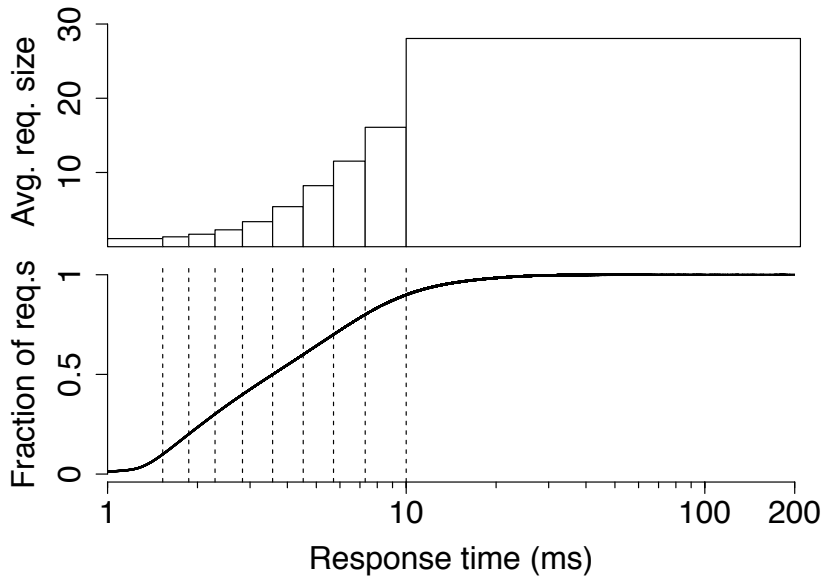


Figure 3.4: The distribution of multiget response times (bottom). The average multiget size of the requests binned by response time for each of the 10 percentile intervals of the response time distribution (top).

of the system capacity and measure the response time of multiget requests.

Figure 3.4 shows the distribution of response times. We also analyze and plot the average multiget size of requests binned by their response times across 10 percentile intervals of the response time distribution. We observe that there exists a positive correlation between the response time of a multiget and its corresponding size (the Pearson's correlation coefficient is 0.403). In other words, the larger the multiget size, the more likely it will incur a higher response time.

The main takeaway from our analysis is that multiget requests vary widely in size, key access pattern, and response time. Combined these variations manifest as uneven load on the serving nodes. These variations bring several challenges to achieve performance predictability but at the same time they create opportunities for performance improvements through scheduling read operations, as we discuss in the next subsection.

3.1.3 Quantifying Latency Reduction Opportunities

We illustrate the potential benefits of inter-multiget scheduling by analyzing our production traces to quantify latency reductions at different percentiles. This analysis focuses on upper and lower bounds estimated on a simplified model of the system (detailed in §3.4.3), though our evaluations are based on production and synthetic workloads on a real system prototype. We present simple estimates for this analysis because the problem of choosing the optimal request allocation can be shown to converge into a more complex version of the online knapsack problem, which is a difficult problem [77].

Recall that we denote by slack the possible delay of an operation without affecting the completion time of its containing request. As an upper bound, we assume that every operation that can be slacked could ideally produce a latency reduction equal to the slack for that operation. We compute it by subtracting the response time of an operation from the response time of its request. To estimate a lower bound, our model assumes that every server has a queue of operations where operations are enqueued if the server is not idle. We then consider every operation, x, that can be slacked and check whether the server processing that operation could have processed the subsequent operation, y, in its operation queue such that the service time of y is no larger than the slack time of x. If that is the case, we measure the resulting latency reduction if it is greater than zero.

Figure 3.5 plots the range of average per-operation latency reduction (area between upper and lower bounds) for increasing system utilizations. It can be seen that latency reduction opportunities increase with utilization levels and that at 90% utilization is roughly double that at 10% utilization.

In summary, these results demonstrate that there exist potentially good opportunities to reduce request latencies via slacking operations.

3.1.4 Related Work

There is a large body of work in the literature on the problem of reducing tail latencies for distributed storage systems. Load-balancing techniques [42, 47] have been employed to make replica server selections in replicated settings. Other systems techniques such as speculative reissues and duplication of requests [41, 43, 78] have also been used to reduce latency at the tail. However, all these approaches offer optimizations that work on the granularity of requests (or operations). Other approaches that perform adaptations at longer time-scales have proposed selective replication of data [79, 80], tuning the placement of data according to keys accessed frequently [81, 82], configuring priorities and rate limits across multiple systems stages (e.g., network and storage) [83], and batching requests to adapt to variation in storage-layer performance [84]. None of these works addresses the

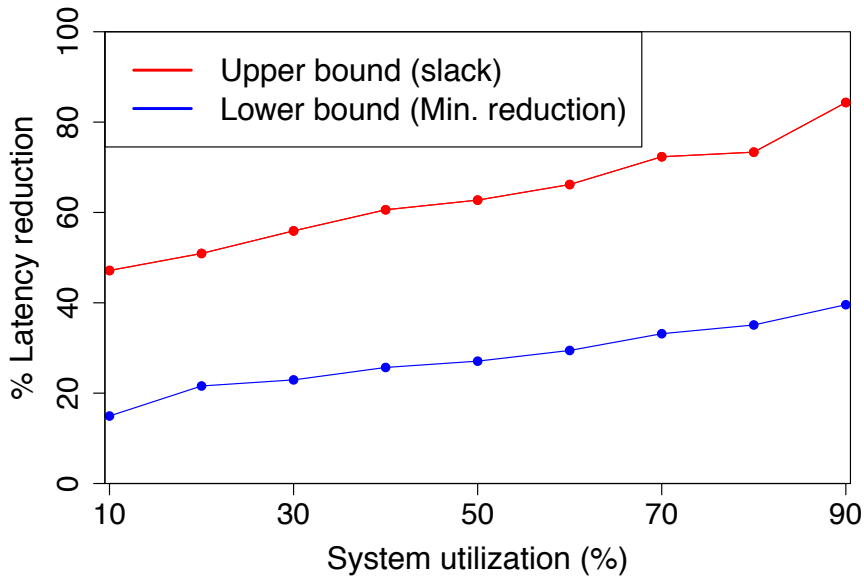


Figure 3.5: Upper and lower bounds for latency reductions at different system utilization levels.

added dimension of multiget workloads and how to leverage knowledge about their structure to optimize scheduling at the backends.

Adaptive parallelization techniques [85, 86] have also been used for latency reduction in adaptive server systems. Haque *et al.* [85] uses dynamic multi-threading to reduce tail latencies by keeping track of the progress of request execution times and increasing the level of parallelism the longer the request stays in the system. To determine the level of parallelism, they profiled the workload and hardware resources offline and computed a policy. The requests then decide online on their level of parallelism based on the computed policy, the system load level, and their own progress. Li *et al.* [86] generalized this approach and aimed to reduce the number of requests that miss a user-defined target latency. They serialized large requests in the system to reduce the impact of queuing delay on the smaller requests which is a form of work-stealing akin to our Shortest Bottleneck First (SBF) policy. Both the aforementioned works, however, focus on optimizing execution per server. They do not deal with the problem of scheduling requests across different servers. In addition, they do not explicitly target key-value stores, which offer their own set of challenges.

There is also work in the literature that has some commonalities with Rein but is otherwise applied to an entirely different domain — namely, network flow scheduling. Baraat [69] is a decentralized co-flow scheduler that primarily utilizes FIFO scheduling, but avoids head-of-line blocking by dynamically modifying the multiplexing level in the network. Varys [87] – a network scheduling system for data-intensive frameworks – utilizes a combination of Smallest-Effective-Bottleneck-First (SEBF) to minimize flow completion times and Minimum-Allocation-for-Desired-Duration (MADD) to decide the rate allocation for each flow in a way that slows down all flows to match the longest flow. Aalo [88], similarly to Varys, operates on the co-flow scheduling problem but does not assume a priori knowledge of co-flow sizes. To work around this lack of clairvoyance, its scheduling algorithm involves using multiple queues with different weights and assigns flows dynamically from higher priority to lower priority queues based on the amount of data that each flow has accrued. While these approaches have some commonalities with Rein, they tackle a different set of problems with their own list of challenges. To the best of our knowledge, no other work has considered the benefits of task-aware scheduling for multiget workloads within the context of distributed key-value stores.

3.2 Rein Scheduling

Rein aims to reduce latency of key-value stores via inter-multiget scheduling by exploiting variations in the attributes of multigets, such as size, processing time,

and fan-out. Specifically, the scheduler's goals are as follows:
• Achieve best-effort minimization of median multiget response times.
• Provide a more predictable performance by reducing high-percentile latencies.

Our objective is not to determine an optimal schedule, given the hardness of the underlying scheduling problem. Rather, we seek to design heuristics that can improve response times under realistic settings. For this reason, we resolve to employ a novel heuristic solution that fits within the desired constraints that we fix in the design space as we elaborate below.

The design space of solutions for minimizing latency of storage systems is large, even if only scheduling-based solutions are considered. We make the following decisions to guide us towards a practical solution:
• The scheduler should operate online, with minimal overhead, and assuming no prior knowledge of requests.
• There should be no coordination between clients and servers, nor centralized components; rather, clients may only pass information to servers in the form of meta-data assigned to individual operations.

3.2.1 Solution Overview

We address the design goals via two main components: (i) server-side, multiget-aware scheduling based on (ii) client-side priority assignment. Figure 3.6 visualizes the architecture used in our approach.

As seen in the figure, the process begins when a multiget request is issued at the requester endpoint. This endpoint is either the client that issues the request via the client-side library or the coordinator node that is tasked with processing the request submitted by a client of the cluster.

The multiget request is first subdivided into a collection of *opsets*, wherein each opset is comprised of all operations for each distinct data partition. Thus, operations are split according to the same strategy used for selecting servers to serve operations (typically some form of hashing). Empty opsets are pruned out. Note that, in replicated storage systems, this split maps to the number of replica groups; that is, it maps to the set of servers responsible for a data partition.

Next, the requester endpoint assigns each opset with a certain priority using a priority assignment strategy (detailed later). This priority is then inserted as meta-data in every operation of a given opset; all operations of an opset share the same priority.

The scheduling of operations does not occur until they are processed on the server side. After priority assignment, each operation is sent to a server responsible for the correct data partition. At a high level, the server then serves requests according to their assigned priority, reaping the benefits of scheduling operations

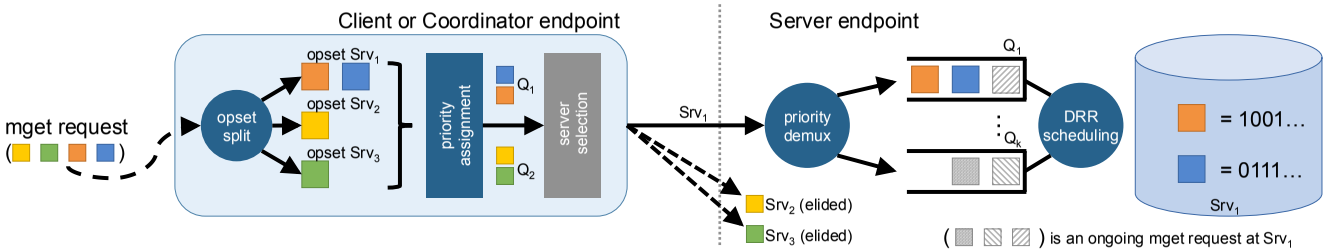


Figure 3.6: Overview of Rein scheduling for a multiget request arriving at the system. On the requester endpoint, the request ( ) is split into multiple opsets, each grouping the operations towards the same data partition. Priority assignment accounts for the cost of each opset and marks each operation with corresponding priority as meta-data. For example, operations are assigned the highest priority (level 1) as they are estimated to be the bottleneck and sent to Srv_1. On the server endpoint, each operation is enqueued into a queue based on its priority. Multiple queues are serviced using Deficit Round Robin (DRR), which efficiently approximates processing operations in highest-priority-first order.

in a multiget-aware fashion. However, to perform this scheduling efficiently in practice, our design makes use of multiple queues. As we discuss below, this design approximates the desired highest-priority-first order while preventing starvation and enabling an implementation based on non-blocking FIFO queues.

While this design appears conceptually simple, the devil is in the detail. The difficulty stems from the fact that, by design, each client (or coordinator) in our solution needs to make independent decisions based on only the structure of each mutiget request. Thus, there are two major challenges that we need to solve: (i) what strategy to use for priority assignment? and (ii) how do we efficiently perform the operation scheduling?

3.2.2 Design Details

We now discuss the components of our solution that address these challenges.

3.2.2.1 Multiget-Aware Scheduling

Multiget-aware scheduling exploits the fact that a multiget's response time depends on the slowest of its operations to complete (i.e., the bottleneck operation). Our first approach was to consider having a simple priority queue on the server side and assigning priorities on the client side. Priorities are assigned based on a notion of the cost of an operation. The cost aims to reflect the amount of work necessary to process an operation. The cost of an opset refers to the sum of the costs of all its operations. The cost is also used to estimate the bottleneck of a request, as we explain below.

Bottleneck estimation. Estimating the bottleneck of a request (that is, the response time of the last operation) is a difficult problem because this time can be influenced by many factors, including server performance fluctuations (e.g., garbage collection, performance interference, or other background activities), resource contention, skewed access patterns, and caching. Knowing the requested value size is therefore not necessarily a good predictor of the response time. In queuing theory, our system can be modeled by a fork-join queue, which factors in the splitting (or forking) of requests amongst K servers and and then being joined into a single response after all operations return. The problem with these models, however, is that they fail to provide exact solutions for more than 2 servers [89]. As such, due to the difficulties of exact-analyses in these models, most studies focus on approximation techniques [90]. To make matters worse, based on our consultations with queuing theory experts, it is challenging for this model to accurately account for the complexities of our system given the multi-threaded nature of server nodes and the fact that multigets do not have a constant K and their opset sizes vary.

One might consider tracking several run-time metrics, such as the distribution of response times, the likelihood of finding a key in cache, etc., and performing an estimation using these data. We found that making an accurate estimation based on the distribution of previous response times resulted in sub-optimal scheduling decisions due to staleness of the information. In addition, given the need for sub-millisecond decision making, performing these predictions in an online fashion can be challenging. Relying on a simple heuristic can therefore be advantageous.

We choose a simple method that we found to yield reasonably good results in practice. For a given request, we assume that the bottleneck opset is the opset with the highest number of operations in it. Multiple opsets can be treated as the bottleneck if they are the same size. In other words, the cost scales linearly with the opset size, which follows what has been observed for the sizes of I/O queries in distributed execution environments [71, 91, 92]. Implicitly, we assume that all operations experience the same response time, which is of course not true in practice. Based on our experiments, we find that there is a weak correlation between the size of the data being requested and the service time if all data is stored in memory. However, this does not hold true if the data is only partially cached. Our evaluation for instance presents the impact that caching can have on our performance improvement. It is part of our on-going work to determine computationally cheap and effective bottleneck estimation techniques.

Initial scheduling algorithms. Based on this understanding of the problem, we started to investigate classic scheduling algorithms by considering the following insights:

- Prioritizing operations with shorter execution times can reduce head-of-line blocking and improve latencies.
- Deprioritizing operations that can be slacked can allow bottleneck operations to be processed earlier, which results in lower aggregate response times.

Using these two insights, we considered two natural scheduling schemes: *Shortest Bottleneck First* (SBF) and *Slack-Driven Scheduling* (SDS).

Shortest Bottleneck First assigns to every operation of a multiget request a priority that corresponds to the cost of the bottleneck opset of that multiget request. The intuition is that requests with shorter bottlenecks should be given precedence to minimize the average request makespan and reduce head-of-line blocking. This strategy is similar to Shortest Job First (SJF) scheduling as well as Shortest Remaining Processing Time (SRPT); however, in our case, given that request completion times are determined by the last operation to finish, we use the cost of the bottleneck opset instead of the cost of individual operations. SBF

favors smaller requests by scheduling them ahead of larger multigets, penalizing them in the process. For workloads with heavy-tailed multiget sizes, which we use throughout this chapter, we find that this strategy can be used to reduce *both* the median as well as higher percentile latencies (*e.g.*, 95^{th} and 99^{th}). Like SJF, SBF and SRPT can be prone to starvation under certain workload distributions [93, 94]. To counter this, we also incorporate a technique to boost priority of an operation after a certain duration has passed since the operation entered the servicing server's queue.

Slack-Driven Scheduling assigns the priority for every operation x of a non-bottleneck opset, O, as the cost of x plus the slack of x divided by the number of operations in O, that is, $(cost(x)+slack(x))/size(O)$. A lower value corresponds to a higher priority. This deprioritizes operations based on how long they are allowed to be slacked without becoming the bottleneck themselves. In doing so, this policy aims to use server capacity more wisely by prioritizing servicing opsets in proportion to how they are bottlenecking their multiget request. As opposed to SBF, SDS does not penalize one request in favor of another. This is because the operations that are delayed are chosen such that they do not affect the overall response time of their parent multiget request.

While these approaches gave good results in simulation, we recognize that, in high-throughput systems (such as key-value stores), canonical priority queue implementations can act as significant performance bottlenecks due to lock contention in multi-producer/multi-consumer settings [95]. This is a well-known problem in the literature; several solutions have been proposed [96, 97]. We address this problem with a novel technique, which we term *multi-level queues*.

3.2.3 Multi-Level Queues to the Rescue

A way to avoid the inefficiencies of priority queues is to utilize multiple FIFO queues. This allows us to escape the lock contention problem since there exist implementations of lock-free FIFO queues that allow concurrent access without incurring synchronization overheads.

We now introduce multi-level queues, a way to use K FIFO queues to approximate a single priority queue by assigning to each of the K-th FIFO queue a different dequeue rate, $w_{Q_i}, i \in [1, K]$. In this design, objects with higher priorities are allocated to queues with higher rates (and vice versa). This provides us with a way to approximate the behavior of a single priority queue in a contention-free manner. As a trade-off, we do not have complete control over the scheduling order. The fact that our bottleneck estimation is just approximate makes this less of an issue for us.

3.2.3.1 High-Level Idea

Our aim is to use the multi-level queue data structure to realize our two scheduling policies — namely, SBF and SDS. To do so, we need to answer the following questions:
(i) How can operations be assigned to queues in a way that realizes the aforementioned scheduling policies?
(ii) How do we configure the number of queues and their respective rates?

We describe some intuitive ways of answering (i) and then tackle (ii) in the sensitivity analysis later in this section. To realize SBF and SDS, we want to conceptually map these policies to multi-level queues while preserving the main insights that govern the policies.

For SBF, we find we can simply assign all the operations of a multiget request to a queue based on the request's cost. In other words, operations belonging to costlier requests get assigned to queues with lower rates, while operations for smaller multigets are assigned to the faster queues. In doing so, we allow smaller requests to bypass larger multiget requests and get serviced with higher priority. This allows us to approximate the SBF schedule.

To realize SDS, we can assign the bottleneck opset to the fastest queue. The non-bottleneck opsets are then "slacked" by assigning them to queues with lower rates based on the ratio of their own cost to that of the bottleneck. In other words, if an opset cost is half of its bottleneck, then, ideally, it should be assigned to a queue that is twice as slow. More generally, however, it should be assigned to the queue that minimizes the difference between the cost and rate ratios. We explain this in greater detail below.

Based on this high-level description, we see another opportunity for *combining both* approaches into one scheme. We achieve this by, first, assigning the bottleneck opset based on its cost (and not simply to the fastest queue as described for SDS). In doing so, we allow opsets with shorter bottlenecks to finish first, which satisfies the SBF policy. The non-bottleneck opsets are then assigned to queues with equal or lower rates according to the aforementioned ratios. This allows us to attain the benefits of SBF by deprioritizing requests with longer bottlenecks while at the same time reducing response time variability between opsets within a request, thereby realizing SDS and increasing the efficiency of resource utilization.

3.2.3.2 Multi-Level Queue Design

Our multi-level queue scheduler (depicted in Figure 3.7) consists of a set of K queues $\mathbb{Q} = \{Q_1, Q_2, \ldots, Q_K\}$. Each queue is assigned different dequeue rates, with the highest assigned w_{Q_1} and successive queues assigned progressively lower rates. The scheduler uses Deficit Round Robin (DRR) to dequeue operations based

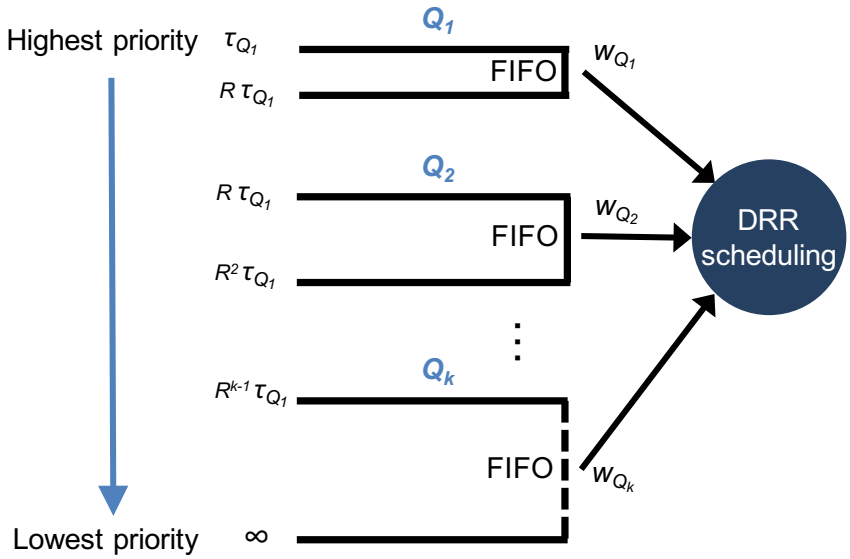


Figure 3.7: A multi-level queue scheme showing K FIFO queues. Consecutive queues have incrementally decreasing rates and exponentially larger bin sizes. Our scheduler uses DRR to dequeue operations from the queues.

on the assigned dequeue rates.

Each queue is assigned an interval of opset costs so that each opset can be mapped to a queue based on its cost. For the queue Q_i, the interval is the pair of thresholds $\tau_{Q_i}, \tau_{Q_{i+1}}$. For $i = 1$, $\tau_{Q_i} = 1$, the minimum cost. For $i > 1$, successive thresholds are calculated as $\tau_{Q_{i+1}} = \tau_{Q_i} * R$ where R is the range factor. In other words, the queue intervals increase exponentially. We opt to use exponentially ranged thresholds since fine-grained prioritization can result in suboptimal makespans if the opset completion times are unknown [88]. Despite the fact that we know the size of each opset, our cost function is merely an estimate and the actual execution times can differ across servers due to variation of performance. Thus, we do not to differentiate between opsets in a fine-grained manner based on their sizes.

Opset queue assignments. We now detail how we assign priority to each opset (technically, the operations within it). Once a multiget request is split into its opsets, we calculate the cost of the bottleneck opset, B. Opset B is then assigned to Q_i such that $\tau_{Q_i} \leq cost(B) < \tau_{Q_{i+1}}$. In other words, bottlenecks with higher costs are assigned to lower priority queues, which preserves the SJF properties of SBF. At the same time, since $w_{Q_i} > 0, \forall i \in [1, K]$, our requests do not suffer from

starvation as they are guaranteed a non-zero service rate.

We use a different method to assign non-bottleneck opsets to queues. Algorithm 1 gives the pseudo-code for the procedure. For a non-bottleneck opset, op, we calculate the ratio between its cost and that of the bottleneck opset. We then loop over all the queues and choose the queue that minimizes the absolute difference between the cost ratio and corresponding dequeue rate ratios. In other words, we try to find

$$Q_{min} = \underset{q \in \mathbb{Q}}{\text{argmin}} \left\| \frac{\text{cost}(op)}{\text{cost}(B)} - \frac{w_q}{w_B} \right\|$$

where w_B is the rate of the queue to which B is assigned.

The intuition behind this is that these opsets are likely to complete before the bottleneck. As such, we opt to assign them to a lower priority queue to slack them in a manner that is proportionate to the ratio between their cost and the bottleneck's cost. This assignment captures the properties of our second priority policy – SDS – which is mainly designed to synchronize the completion times of the different opsets (thereby reducing response time variability).

Sensitivity analysis. As with any system with tunable parameters, a primary concern is to determine the reduction in performance if these parameters are not configured optimally. To address this concern, we resort to trace-driven simulations to conduct a sensitivity analysis of our main parameters. We used the SoundCloud trace to generate our workload and the same settings as in Section 3.4.3. The primary goal of this analysis is to assess how changing the number of queues (K) and the range factor (R) – our two main configurable parameters – can affect the attained latency reductions. In both experiments, we set the initial range to one and the default values of K and R to four and three respectively. We observed minimal changes for the median and 95^{th} percentiles, so we exclude their plots for brevity. In Figure 3.8, we plot the normalized performance degradation (%) experienced compared to the optimal settings for different values of K and R. We see that varying the number of queues can impact the 99^{th} percentile latency by up to 25%, whereas varying the range factor changes it by up to 15%. The biggest performance drop occurs at the setting with the low number of queues and smallest range where this can be expected (the benefits of the multi-level queue are effectively being removed). More importantly, however, the performance drop in the 1-hop neighborhood of the optimal setting is a more manageable 8%. As such, even in unfavorable settings, Rein is able to realize reasonable performance gains with only minor hits at the higher percentiles.

Based on these results, it is important to note that, while having more queues can provide increased control over scheduling, it can result in queue load imbalances if the average number of operations entering each queue is not carefully accounted

Algorithm 1 Opset queue assignment algorithm.

Data: opsets, queues

R = range factor `// bottleneck opset`

bn = argmax $cost(op)$ τ_{next} = 1 q_{bn} = $queues[0]$ **for** q *in queues* **do**
    `// Find the queue for the bottleneck opset`
    if *cost(bn)* $\geq \tau_{next}$ *and cost(bn)* $< \tau_{next} * R$ **then**
        q_{bn} = q break
    $\tau_{next} = \tau_{next} * R$

`// Assign bottleneck opset to queue`

tagPriority(B, q_B) `// Find and assign queues to non-bottleneck opsets`

for *op in opsets* **do**
    if op == bn **then**
        continue
    costRatio = cost(op)/cost(bn) q_{op} = null minError = arbitrarily high value (e.g. 10^9) **for** q *in queues* **do**
        rateRatio = $w_q/w_{q_{bn}}$ error = $abs(costRatio - rateRatio)$ `// Find the queue with min. diff between cost and rate ratios`
        if $error < minError$ *and* $w_{q_{bn}} \geq w_q$ **then**
            minError = error q_{op} = q
    tagPriority(op, q_{op})

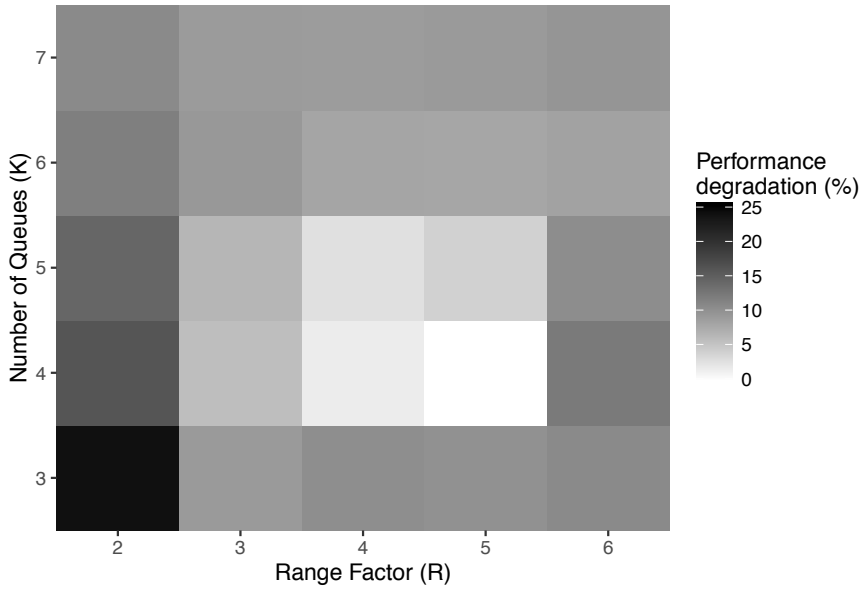


Figure 3.8: Sensitivity analysis of the range factor (R) and the number of queues (K), showing that the performance drop is about 8% in the 1-hop neighborhood of the optimal setting.

for. We explored different combinations of these parameters through simulations and our best performing configuration was $K = 4$ with queue rates increasing in increments of 1. We also found the optimal value for both R and r_{Q_1} to be 1 and 5 respectively. However, we believe that these parameters are dependent on the workload and, as such, need to be tuned according to the target workload to reach optimal results. We leave developing adaptive algorithms for tuning these parameters for future work.

3.3 Implementation

Rein is implemented in Cassandra, a widely used distributed database offering a key-value interface and multiget operations. We used version 2.2.4 of Cassandra. For every request sent to Cassandra, each node in the cluster can act as a coordinator, server or both based on the utilized internal routing mechanism. Cassandra itself is based on a Staged Event-Driven Architecture (SEDA) [98]. Each stage performs different functions and has its own thread pool as well as queue for all the outstanding operations. The stages communicate among each other via a

messaging service.

We considered two different thread pools that are relevant to Rein's implementation; these are the native-transport and the read thread pools. The former is responsible for reading incoming requests from the TCP socket buffer. The latter handles the assignment of threads for coordinating the actual read operation on the target server. Rein's scheduling algorithms are implemented in the read thread pool queue. Note that Rein cannot be implemented in the native-transport queues as this is where requests are being parsed. As such, there is no notion of priorities at that stage and the requests can only be handled in FIFO order.

Rein's priority assignment takes place at the coordinator. Since these nodes are responsible for dispatching operations to the servers handling the operations' partitions, we can count the number of operations going to each partition and assign our priorities accordingly. As per Cassandra's code path that handles reads, these operations can either happen locally (if the data is present on the coordinator node itself) or are sent to other nodes. If data is replicated, Cassandra uses a snitch module to load balance the requests across the different replicas. However, since our priorities are assigned based on the number of operations headed to a certain partition (i.e., the opset size) and not a replica, our approach is logically separated from the load-balancing scheme employed by Cassandra.

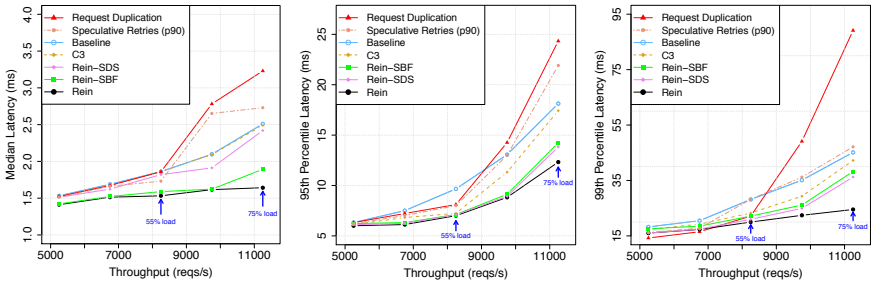
3.4 Evaluation

We evaluate the effectiveness of our approach compared to a baseline and other latency reduction techniques. We also leverage simulations for running higher-scale experiments and to test our system for a wider variety of configurations and workloads. Our main results are as follows:

(i) We evaluate the performance of Rein in a real system under both realistic and synthetic workloads. We find that Rein substantially outperforms all other approaches that we ran against it; e.g., at the 99^{th}-ile at 75% utilization, Rein reduces latency of multiget requests by about a factor of two. Moreover, our results show that request duplication (both straightforward and speculative) is ineffective in reducing the tail latency because it simply doubles the load offered to the system.

(ii) We assess the overheads of using single priority queues and establish the need to use multi-level queues.

(iii) We test Rein under higher-scales and under a wide variety of workload conditions and corner cases, observing that its performance is in several cases close to an idealized clairvoyant oracle strategy.



(a) Median latency. (b) 95^{th} percentile. (c) 99^{th} percentile.

Figure 3.9: Latency attained by the different variants of Rein compared to other latency reduction techniques. The x-axis represents the offered load. We see that Rein's approach achieves the highest gains in the median as well as high percentile latencies.

3.4.1 Effectiveness of Rein

We first evaluate Rein in a realistic testbed using production workloads. For these experiments, we use 16 m3.xlarge AWS EC2 instances. To generate our workloads, we used a modified version of the Yahoo! Cloud Serving Benchmark (YCSB) [76] – a general-purpose cloud systems benchmarking tool – and configured it to run on a separate node. The instances each have a total of 15 GB of memory, 2x40 GB SSD, and 4 vCPUs. We insert data items (also called rows) into Cassandra with value sizes generated following the distribution of Facebook's Memcached deployment [75]. Our experiments focus on evaluating the effectiveness of Rein at different operational conditions. In all runs, we set the replication factor of our partitions to three and the concurrency level, or how many requests can a node serve simultaneously, to eight. The consistency level for all requests is set to one. We also disable the automatic paging feature in Cassandra to make sure our queries are not sent in a sequential manner.

Realistic workloads. We first evaluate how Rein performs under different system load levels. We use YCSB to generate multiget requests based on our trace (described in Section 3.1.2). For this scenario, we first insert a dataset composed of 100,000 rows, such that the entire dataset can fit into memory. As such, most of the reads are satisfied through the operating system's page cache (since Cassandra delegates memory management to the operating system). Our first set of experiments uses 500 closed-loop YCSB threads. We first test the cluster and find the maximum attainable throughput, which is ∼15,000 requests/sec. Note that

these are multiget requests, which fetch 8.6 keys on average. As such, our cluster is actually serving around 129,000 read operations/s for this workload. We then cap YCSB's sending rate for different experiments to evaluate the performance under different system utilization levels.

We compare six different latency reduction techniques against the baseline, which uses Cassandra's default settings. Our results are presented in Figure 3.9.

We evaluate the effectiveness of request duplication, which involves pre-emptively sending an additional request for each query. This allows the client to receive the fastest returning response. Although such a technique has long been used to reduce latency at the tail, it is not universally applicable across a wide-variety of settings. In fact, in our experiments, once we go higher than 55% system utilization, the tail latencies are greatly inflated and become twice as high as the baseline at 75% utilization. This is because sending an extra request essentially doubles the demand on the servers, which degrades performance if the cluster is already saturated, confirming the results in [43].

We also evaluate speculative retries, which is another popular latency-reduction technique that is a more general case of request duplication. In this case, the coordinator triggers another request once a response is delayed past a pre-defined timeout. We have tested many different timeout values and found that setting it to the 90^{th} percentile of the response time provides the most favorable results. We can see that, given its less aggressive nature, speculative retries provides modest benefits at the tail (between 15% and 25% reduction) but suffers the same problems as request duplication once our saturation level exceeds the midpoint (albeit with lower performance degradation than straightforward duplication).

In addition to these techniques, we also tested a modified version of C3 [42] – an optimized dynamic load-balancer – to see how it fairs against Rein. C3 is composed of a replica ranking system (which chooses which server should answer a request) as well as a distributed rate-limiter. We removed the rate-limiter, as we found that its Akka-based [99] implementation was causing performance bottlenecks in our experimental settings. As seen in the results, C3 manages to provide tangible benefits at the tail (upwards of 25%) while not suffering any performance problems at the higher utilization levels. However, we can also observe that there are little to no improvements in the median latency, which we believe is a consequence of the fact that C3 was optimized primarily for the tail.

Lastly, we experiment with 3 different variations of Rein. Rein-SDS and Rein-SBF use the pure forms of Slack-Driven Scheduling and Shortest Bottleneck First, respectively. We first quantify the gains of each of the policies to discern whether a combination of both is indeed superior. As we can see, both approaches provide increasingly higher gains as we move towards higher utilization levels. This is because as the load increases, the system experiences higher burstiness,

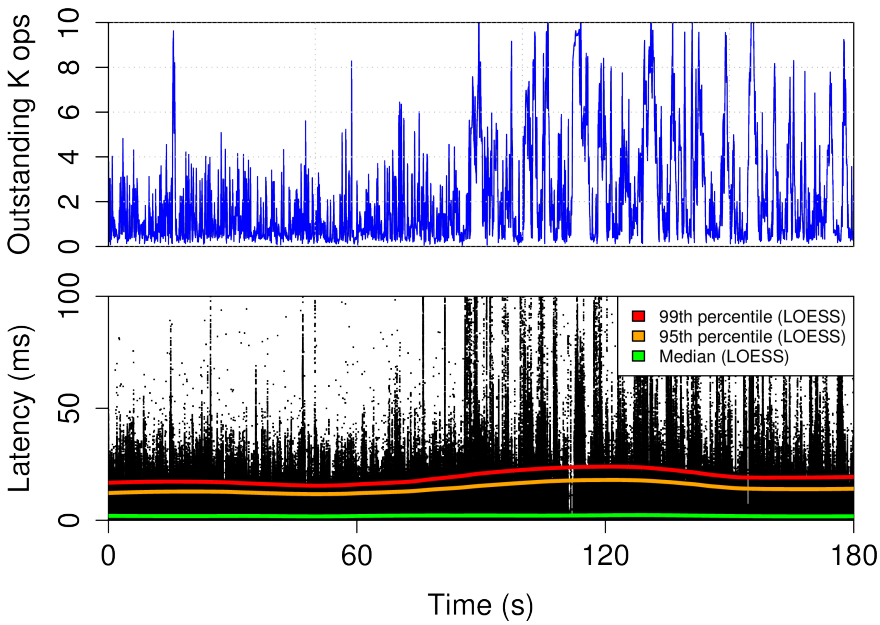
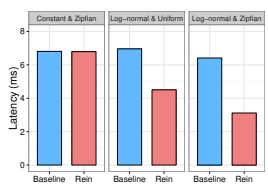
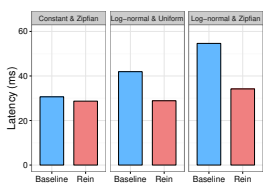


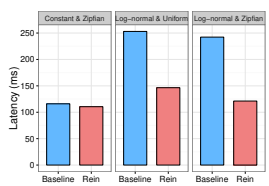
Figure 3.10: Time series showing the multiget latencies as well as the number of outstanding operations during a three-minute run. We also plot the moving median, 95^{th}, and 99^{th} percentiles, which we smoothen using LOESS regression. The median, 95^{th}, and 99^{th} percentiles are averaging at around 1.9, 14.3, and 19.2 ms respectively. Points exceeding 100 ms latency are not shown for readability reasons.



(a) Median latency.



(b) 95^{th} percentile.



(c) 99^{th} percentile.

Figure 3.11: Latency comparison of Rein versus the baseline using different synthetic workloads at 75% utilization.

which temporarily causes requests to queue. In turn, this provides our scheduling heuristics with greater opportunities to re-order the execution of these operations and improve tail latency. We can see that both variations can provide upwards of 50% reduction at the 95^{th} and 99^{th} percentiles at 75% utilization. In addition, Rein-SBF, which is optimized for reducing average makespans, is able to cause a sizable reduction in the median latencies, reaching up to 30% at higher load levels. Despite the fact that SBF relies on prioritizing requests with shorter bottlenecks at the cost of delaying larger requests, it is still effective at reducing higher percentile latencies due to the heavy-tailed nature of our multiget sizes. In essence, delaying very large requests has no negative impact on the 99^{th} percentile but can, counterintuitively, even improve it. In summary, both variations provide much higher gains at different configurations than the other techniques we compare against. We can also see that Rein is able to outperform the two policies, which shows that a combination of both policies is in fact better for both the median and tail latencies.

To investigate the level of burstiness of our workload generation and to develop a deeper understanding of response time characteristics, we plot in Figure 3.10 the time series of the number of outstanding operations (from the point of view of YCSB) sampled at 100 ms intervals as well as the multiget latencies during a three-minute interval of Rein at steady state. We also plot the moving median, 95^{th}, and 99^{th} percentiles for the multiget latencies, which are smoothened using local regression (LOESS) [100] with a span parameter of 0.3. We can see that the number of outstanding read operations can vary by as much as 10,000 operations during the experiment with a standard deviation of 2.38, which is mostly due to the heavy-tailed nature of multiget sizes in our workload. We also observe latency spikes, which increase in frequency and magnitude during periods of higher loads (*e.g.*, after the 80s mark). However, despite this, both the 95^{th} and 99^{th} percentiles remain relatively stable throughout the experiment.

Synthetic workloads. We also evaluate Rein against a wide variety of workloads to assess its generality and to ensure that our solution is not tailor-fitted to our trace. To formulate these workloads, we focus on two key features: multiget size distribution and the access patterns.

For multiget distributions, we want to observe the behavior of Rein against both short- and heavy-tailed workloads. For the former, we simply use a constant multiget size of 50, which corresponds to the fan-out factor of requests reported in Bing's cluster [1]. We use a constant multiget size to assess the performance of Rein in a challenging scenario in terms of latency reduction opportunities due to the uniformity of multiget size. Secondly, we generate a heavy-tailed multiget size distribution by fitting a log-normal curve to the distribution of multiget sizes reported inside Facebook's memcached production clusters [4]. The values of μ

and σ of the resulting distribution are set at 2.5 and 1.25, respectively.

For the key popularity distribution, we configure YCSB to use Zipfian access patterns to emulate skewed reads (where certain keys are orders of magnitude popular than others). In addition, we also use uniform access patterns in which keys are equally likely to be requested in a multiget.

We then combine the above workload characteristics to generate three distinct workloads.* The three workloads have multiget size and access pattern distributions as follows: *Constant & Uniform*, *Log-normal & Uniform*, and *Log-normal & Zipfian*. We evaluate Rein performance with these workloads at a throughput of 900, 1,700, and 1,900 requests/s, respectively. This equates to around 75% system utilization for each of the specified workloads.

In Figure 3.11, we can see the performance of Rein compared to the baseline for varying multiget sizes and key access patterns. In the scenario with constant multiget size and Zipfian access pattern, head-of-line blocking is minimized due the uniformity of multiget sizes. As such, Rein achieves modest performance gains of 6% and 5% at the 95^{th} and 99^{th} percentile, respectively. In contrast, having heavy-tailed, log-normal distributed multiget sizes allows Rein to attain gains with the SBF aspect of our scheduling scheme. Since the SBF policy is designed to reduce head-of-line blocking, it exploits non-uniform request sizes to attain performance gains. When using a uniform access pattern, we observe a 35% reduction at the median and up to 30% and 42% at the 95^{th} and 99^{th} percentiles, respectively. Finally, when using a workload with log-normal distributed multiget sizes and Zipfian access pattern, Rein achieves higher performance gains with a 1.6x reduction at the 95^{th} percentile and a roughly two-fold reduction at both the median and 99^{th} percentile latencies.

SSD-heavy reads. In addition to memory-heavy settings, we evaluate Rein's performance with SSD-heavy reads in high-load settings. To do this, we inserted a dataset composed of 300 million rows, such that only approximately one-third of the data can fit into the operating system's page cache. In this scenario, most of the reads are going to be performed on the instance's SSD. As seen in Fig. 3.12, Rein improves the 95^{th} and 99^{th} percentiles by up to 35% and 15%, respectively, with only minor benefits at the median. These results, while still a significant improvement, are likely smaller than in the cache-heavy read scenarios, due to Rein's inability to predict whether a requested value exists in-memory or on SSD. Since one-third of the dataset can exist inside the cache, our ability to predict bottlenecks can be compromised since our scheduler determines bottleneck opsets solely based on the number of operations inside them. We leave how to improve

* We omit the workload with constant multiget size and uniform access patterns since it does not offer opportunities for scheduling-based optimizations.

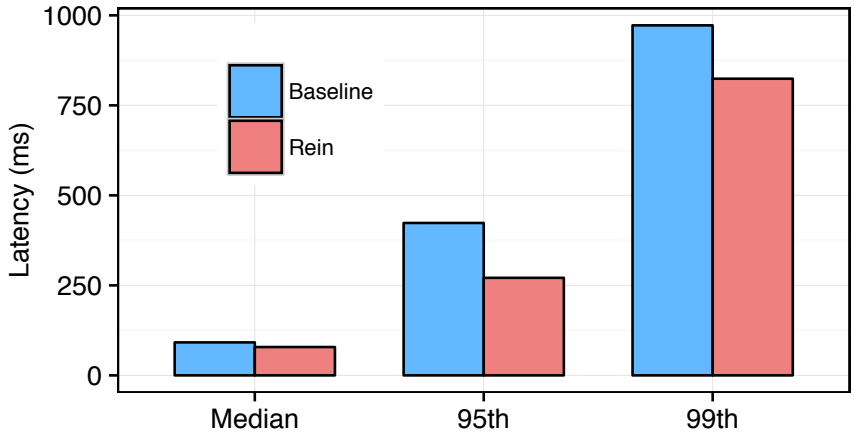


Figure 3.12: Latency comparison of Rein versus the baseline for SSD-heavy reads. The dataset size was adjusted such that the nodes can only cache one-third of the stored rows.

our bottleneck prediction given the interactions between multiple caching layers for future work.

3.4.2 Performance of Single-Priority Queues

Next, we evaluate the performance of different implementations of priority queues by measuring their throughput under high-load settings. Namely, we use Java's Priority Blocking Queue as well as a concurrent Priority SkipList Queue that is based on Lotan and Shavit's design [101]. We compare them with our multi-level queue (configured with $N = 5$). In addition, we also include a comparison to a concurrent FIFO queue – namely, Java's Concurrent Linked Queue, which is based on Michael and Scott's algorithms [102] – and use it as a baseline.

We measure the throughput of these data structures (in a similar manner to [103]) by using 32 threads that either enqueue or dequeue elements from the target queue with a 50% probability. The priorities of inserted items are integers chosen uniformly at random between 1 and 5. Each thread generates a total of 100,000 operations.

We run these experiments on a single machine with 128 GB of RAM and an Intel Xeon E5-2640v3 with 16 physical cores at 2.60 GHz with hyper-threading enabled. We repeat each experiment 50 times with different random seeds.

Figure 3.13 shows the results. Priority Blocking Queue and Priority SkipList Queue obtain on average around 110,000 and 130,000 ops/s, respectively. On

the other hand, Concurrent Linked Queue achieves much higher throughput as it does not incur the overheads of maintaining a priority order for its elements. In comparison, multi-level queue provides much higher throughput, at roughly 2,000,000 ops/s, which is one order of magnitude higher than the other priority queue implementations and a five-fold improvement over the non-blocking FIFO queue.

The reason multi-level queue outperforms traditional priority queue implementations is that it substantially reduces lock-contention, thanks to the fact that it does not need to maintain a fine-grained priority order. Multi-level queue also has a significant performance advantage over FIFO queue despite being based on the same data structures and incurring the overhead of performing scheduling rounds for Deficit Round Robin (DRR). The reason is that, while the Concurrent Linked Queue is indeed lock-free by relying on compare-and-swap (CAS), it does not eliminate the contention between multiple producers trying to enqueue elements concurrently. Multi-level queue reduces this type of contention due to the fact that the threads' activity is spread over multiple queues — five in this case. The relatively higher variance of multi-level queue is primarily due to the workload generated, which differs across experiments. The more balanced the workload on the target queues, the lower the contention. However, even at its minimum, the performance of multi-level queue is still higher than that of FIFO queue. This shows that in multi-consumer/multi-producer settings, multi-level queue is likely a better choice than lock-free priority queues and even concurrent FIFO queues. This finding supports our adoption of multi-level queue for realizing Rein in highly concurrent systems such as key-value stores.

3.4.3 Simulations

We now turn to simulations to assess the performance of our techniques at larger scales. We built our simulator in Python using a discrete-event simulation framework called SimPy. We evaluate the performance of Rein's different multi-level queue policies and compare them to an oracle (idealized) approach as well as the baseline policy (*i.e.*, FIFO). For the purposes of load-balancing requests across servers, we use a straw-man approach, where replicas are chosen for each operation using round-robin.

We simulate a system with 128 clients and 128 servers at a concurrency level of eight (*i.e.*, they can service up to eight operations in parallel), each operating with a replication factor of three at an average service rate of 3,750 requests/s. We set our one-way network latency to 50 μs. Similarly to the real experiments, we drive our workload using the SoundCloud trace. We also generate the value sizes for the requests using the same distribution that we used in our previous experiments. We

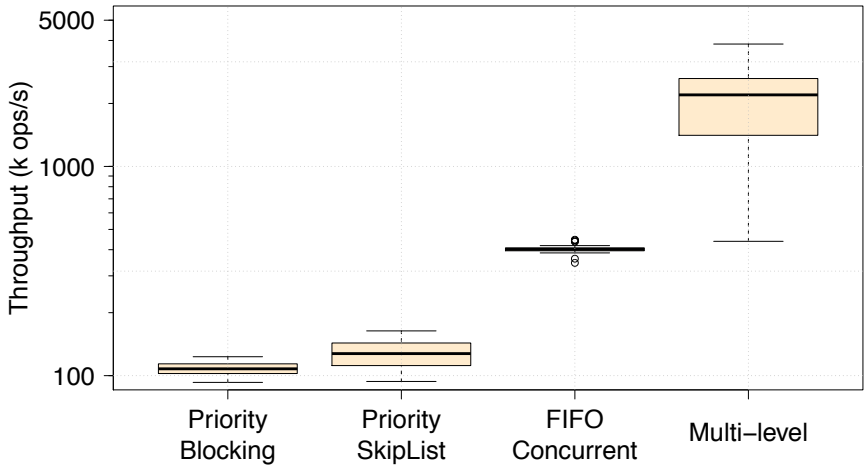


Figure 3.13: Throughput benchmark comparing different queue implementations.

then generate request inter-arrival times using a Poisson process in which the mean rate is set to match 75% of system capacity. The experiments are then repeated six times with different random seeds.

In addition to using the different variations of Rein, we also use a clairvoyant oracle. For this method, the client has complete knowledge of the system's state. It knows the instantaneous load on all servers (*i.e.*, queue sizes) as well as their service rates. In addition, it can also observe all the in-flight requests that are sent from each client in the system and their target servers. Using this information, the oracle can construct a more idealized cost estimation that not only considers the opset sizes for a request but also the present as well as the future load on the target servers; the latter of which is calculated by counting all the in-flight requests. This allows the client to have a more accurate prediction of the slack as well as the bottleneck, which results in better scheduling decisions. Using an oracle allows us to assess the performance gap that we experience by adopting a completely decentralized scheduling protocol in Rein.

Figure 3.14 shows the read latencies at the median, and 95^{th} and 99^{th} percentiles averaged across the six runs. The standard deviation is not shown as it is largely negligible. As shown, Rein outperforms the baseline scenario across all percentiles and improves the latencies by up to a factor of two at the median, and 95^{th} and 99^{th} percentiles. In addition, Rein is within 7% of the performance attained by the oracle, which has the advantage of having global state information that allows it to make better scheduling decisions. Despite Rein's lack

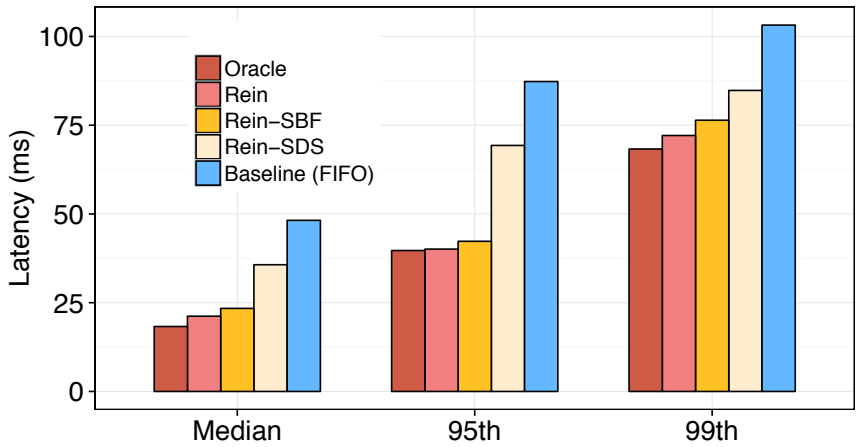


Figure 3.14: Latency comparison of Rein's multi-level queue policies versus the oracle and FIFO baseline.

of coordination, we are still within reach of the performance of a fully coordinated system.

3.5 Discussion

How generic is Rein? Rein uses a system model that is commonly employed in distributed key-value stores. Its scheduling policies do not leverage any system-specific parameters and are logically separated from the load-balancing algorithms. As such, the same scheduling heuristics can be applied to a number of key-value stores (*e.g.*, Memcached, MongoDB, Redis) without any customization.

What is the overhead of using Rein? Our scheduling heuristics are simple. At the client-side, for every request, we count the number of operations in each opset and calculate the maximum operation count (to determine the bottleneck). No running statistics or other types of complicated mathematical procedures are performed. To put this into context, Cassandra's dynamic snitch requires considerably more complex operations to be performed for ranking the replicas, as it maintains exponentially decaying reservoirs of latencies toward the different backends. As such, we consider our approach to be minimalistic in this regard.

Why is Rein effective despite being static? Even though Rein does not account for system state, its lack of dynamism does not detract from its viability. For larger multigets – which generate 100s to 1000s of operations and for which our scheduling decisions matter the most – their response times are mostly dominated

by the service times of their operations. As such, despite the various sources of variability (*e.g.*, variable waiting times, service rate variations, network latency spikes, *etc.*), our statically-derived cost estimate remains valid. However, as part of our ongoing work, we are modifying the SDS approach to incorporate feedback information from the servers pertaining to the sizes of the different priority queues (in the form of running averages). Preliminary results have shown reasonable gains from applying this approach.

Does Rein work at different consistency levels? In the evaluation, we maintained a consistency level of *one* for all our read requests (that is, each operation requires a response from only one replica, *e.g.*, the server chosen by Cassandra's snitch). This setting is quite commonplace in large Web services today [65, 104]. It remains to be seen how Rein can perform at higher consistency levels; however, it is important to note that increasing the consistency level also increases the number of operations being sent to the replicas, which would provide more opportunities for our scheduling policies to exploit.

Can Rein be applied to low-latency key-value stores? Ultra-fast key-value stores [105, 106] have been gaining traction recently. They usually employ shared-nothing architectures, where each core is assigned to a partition and there exists no single-point of contention in the system. Doing this effectively eliminates context-switching overhead and reduces operation latency to the order of microseconds. However, these ultra-low services times can create challenges since the kernel could become a bottleneck [107]. The implications for Rein is that most of the queuing in the system could end up happening at the kernel level. This would put our system at a disadvantage since Rein operates strictly on the application layer. However, kernel-bypass techniques – which are also increasing in popularity [105, 107] – can be leveraged to effectively merge kernel-level with application-level queues, allowing us to realize the full benefits of Rein.

3.6 Conclusion

In this work, we propose scheduling techniques that take advantage of the structure of multiget requests in real-world data store workloads to reduce aggregate median and tail latencies. Under heavy loads, our scheduling algorithms reduced the median, 95^{th}, and 99^{th} percentile latencies by factors of 1.5, 1.5, and 1.9, respectively. Rein demonstrates that distributed scheduling can provide significant benefits in the context of key-value stores without requiring coordination. While our evaluation focused on Cassandra, because our solution is non-intrusive and relatively straightforward to implement, it should be easy to apply to other systems.

Chapter 4

Path Persistence in Large Cloud Provider Networks

As cloud storage becomes more ubiquitous, cloud provider networks are now playing an essential role in guaranteeing Quality-of-Service (QoS); however, little is known about how traffic is routed across such networks. Operators have long relied on Traffic Engineering (TE) tools to optimize the flow of traffic within their networks, with varying degrees of success. At the beginning of the 2000s, shortest-path routing protocols, such as OSPF [108] and RIP, were prevalent despite being ill-suited for TE operation [109]. Since these protocols lack fine-grained routing control, TE optimization was infrequent and suboptimal. TE optimization became prevalent with the advent of MPLS technology [110] and later with the introduction of the Software-Defined Networking [111] paradigm. Since then, Microsoft and Google have reported performing TE optimizations of their wide area networks with a 5-minute period [15, 112].

Are today's TE optimizations hindering network traffic performance? Despite the crucial role played by TE tools, the impact of frequent TE optimization on the network traffic's performance has not been adequately explored. In this work, we take a first step in this direction by looking at the impact of TE on cloud application's traffic.

Cloud providers perform TE to efficiently utilize their network resources. However, this goal may not align with the needs of tenants' applications and their associated traffic. Fig. 4.1 shows the Round Trip Times (RTTs) of three different TCP connections established between two machines located in two different Amazon *AWS* regions — namely Oregon and Virginia. Each TCP connection uses

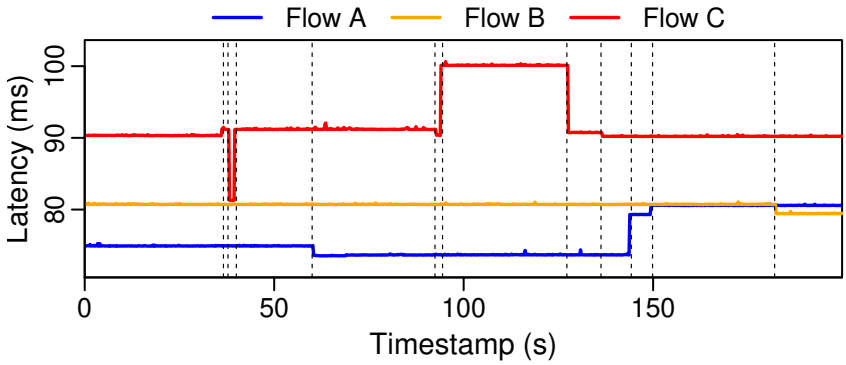


Figure 4.1: TCP RTT measured between two Amazon VMs deployed in Oregon & Virginia. The black dotted vertical lines highlight moments when RTT latency changed and indicate potential TE activity (e.g., path changes).

a distinct TCP source port and periodically generates a ping message to measure the RTT. We observe that each flow experiences different RTT values. Moreover, these RTTs are relatively stable around one value for a period of time before suddenly "jumping" to a different value.

This simple experiment leads to three observations: *(1)* TE is operating at a time scale of seconds, *(2)* traffic flows may experience frequent latency variations, and *(3)* TE may be unfair as some flows consistently experience higher latencies (*e.g.*, flow C) and/or experience more frequent latency changes (*e.g.*, flows A and C) than others.

Highly-variable flow latencies can be cumbersome for application developers. First, high-frequency oscillations in inter-region latencies adds yet another source of uncertainty for application developers — as Microsoft Bing's [113] developers observed [15]. Consequently, guaranteeing latency-related Service Level Agreements (SLAs) for latency-sensitive geo-distributed services [114, 115] can be difficult. Secondly, the performance of congestion control algorithms, both loss-based and delay-based ones, degrade under frequent latency changes, e.g., due to packet re-ordering [116, 117, 118, 119]. Additionally. applications relying on UDP can be negatively affected by packet reordering (e.g., VoIP [120]). Finally, our results show that a fraction of "unlucky" flows can suffer from high RTTs over prolonged periods (hours and days), further exacerbating application performance problems. Our prior work [121] shows that these traffic characteristics have existed consistently for at least a few years (since January 2015) and our continued measurements have shown that they are not transient. This prompted

us to take a deep dive and study this phenomenon more comprehensively.

Measuring TE effects in a large-scale Cloud network. In this work, we study the effects of TE on flow latencies measured among all 16 currently available Amazon *AWS* regions, all of which are connected via Amazon's privately-owned infrastructure [122]. We conducted measurements between up to 105 unique region pairs to identify latency characteristics of traffic flows and thoroughly study the three observations presented above.

Results and implications for TE & the design of applications. Our measurements for this work encompass data gathered over a 2-week span. We have also conducted several other measurement snapshots for *AWS* from 2015 to 2019. All our data indicates that flows frequently jump between paths that can have as high as 20% greater latency than the lowest observed latency path. We corroborated our findings in discussions with cloud networking experts.

Our analysis shows that the changes in latency are caused by frequent routing changes and are potentially problematic for delivering traffic that desires specific performance guarantees. Moreover, we also found that, despite these frequent changes, there is unfairness in the observed flow latencies — where some flows get the lion's share of *low and consistent* latencies while others are penalized over long periods of time. Flow unfairness can be undesirable, especially for latency-sensitive services deployed across several geographic regions [123]. Last but not least, tenants have incentives to monitor the latency observed per source port and then *selfishly* move latency-sensitive traffic to low latency paths. This could, in turn, overload certain paths and trigger a counter-response from the TE mechanisms, a classic problem known as "Selfish Routing" [124]. This work does not aim to solve these problems but rather exposes the current effects of TE. The hope is that quantifying these effects would motivate the community to tackle the challenges of designing TE mechanisms suitable for cloud environments.

4.1 Background

This section provides the necessary background for TE techniques used to optimize the flow of traffic in cloud networks.

Traffic engineering basics. Network operators perform TE to optimize the flow of traffic in their network, e.g., reducing communication latency, ensuring fairness, and providing traffic isolation without increasing packet drops. TE consists of a closed control loop between a *monitoring* component and a route

computation component. This loop spans both the data- and control-planes. The data-plane both forwards packets according to the installed monitoring & forwarding rules and collects traffic statistics. The control-plane fetches the collected traffic statistics and feeds them as input to the route computation engine, and then updates the data-plane with newly computed routes (in the form of per-switch forwarding rules).

Multipath forwarding mechanisms. Modern TE mechanisms make extensive use of multipath forwarding [112] to split an aggregate of traffic flows across different paths. Specifically, for each aggregate of flows between two nodes in a network, the operator specifies the *splitting ratio* of this traffic aggregate that should be routed on each path [125]. The operator also specifies whether the splitting of the traffic aggregate should be performed at the *per-flow* granularity (i.e., packets with a common set of header fields traverse the same path) or at the *per-packet* granularity (i.e., packets are routed using some sort of weighted round-robin scheduler). Network operators have traditionally operated networks using per-flow splitting mechanisms, e.g., based on hash calculations on the packet header (i.e., specific IP and TCP/UDP fields). The reason for using per-flow mechanisms stems from the way TCP congestion control (such as, NewReno and Cubic) *reacts* to the presence of packet reordering by assuming network congestion, which may not always be the case. Specifically, when the receiver of a TCP flow receives packets out of order, it sends duplicate acknowledgments (dupACKs) for the first missing packet in the received TCP stream. When the sender receives three dupACKs, the sender assumes that the network is congested and reduces its congestion window, and consequently its sending rate. Packet reordering may arise when a routing (re-)configuration takes place and a flow is moved from a high latency path to a low latency path. This can happen, for example, when network operators (or automated TE tools) update any hash-based traffic splitting ratios in the network, which is an operation known to cause flow rehashing [125, 126].

Evolution of TE in cloud networks. Traditional TE tools and mechanisms (e.g., OSPF [108] and RIP [127]) are ill-suited for supporting the ever-growing inter-region traffic of large cloud networks. Such tools are tailored to shortest path routing and only support uniform splitting ratios, thus they lack the fine-grained routing control needed to effectively utilize network resources. Consequently, wide-area network operators have long relied on MPLS-based OSPF extensions for improved TE [128], by periodically re-optimizing network routes using preconfigured route computation algorithms (e.g., Constrained Shortest Path First (CSPF)) according to the measured traffic volumes. Recently, the three largest cloud providers (*AWS*, *azure*, and *gcp*) transitioned to SDN-based networks [112,

115, 122], in which network operators have full control of the monitoring and route computation processes using the well-defined interfaces between the control- and data-planes.

4.1.1 Related Work

There has been a great amount of work concerning measurements of Internet routes across a variety of performance dimensions. In this section, we discuss only the most closely related work. Several previous works investigated the impact of TE mechanisms on flow performance [129, 130]. Some of these results showed how the performance of a flow can be affected during reconvergence of routing protocols in the wider Internet, possibly including paths across different domains [131]. In contrast, our work focused on *intra*-domain routing where the TE modification is controlled by a single entity. Others have shown how MPLS-based TE can potentially move flows of traffic to high-latency paths in response to routing reoptimization computation [15]. This is to be expected as TE has to move flows of traffic away from congested shortest paths. However, the authors of this previous work do not discuss whether some source ports experience much worse latencies (i.e., flow unfairness). Additionally, we observed a much higher frequency of path changes than expected, possibly due to SDN control of the network.

The techniques to identify base propagation latency are not novel and the concept of identifying base propagation latency by tracking a minimum latency sample has been previously used in [132, 133, 134].

Another large body of literature focuses on inferring the topology of load-balanced networks. The Multipath Detection Algorithm [135] together with Paris [136] and Tokyo [137] traceroutes improve upon traditional traceroute by avoiding measurement anomalies due to load balancing in the network. DTRACK [138] is a probing tool that predicts routing changes and decides on the number of probes necessary to identify a new network path. In contrast, our work does not aim to inferring the network's topology, rather our goal is to observe the *impact* of TE on flow latency persistence and unfairness across different source ports over time.

Using source port manipulation to send traffic via the highest-performance paths has received some attention in the context of datacenter networks [139] and MPTCP [140]. However, our work differs in several ways. First, we do not claim any novelty in using source ports to route traffic along different paths. Second, MPTCP periodically opens connections on new source ports to discover better paths. We believe MPTCP — and MultiPath protocols in general, could use dynamic port search techniques to *proactively* select the best performing source ports. Finally, these prior works do not investigate the impact of TE on flow latency.

4.2 Measurement Methodology

To understand the extent to which a cloud provider's TE operations affect the latency of traffic in their networks, we performed a study across Amazon *AWS* [122], the largest worldwide cloud provider network in terms of market share [141].

Goals of the study. In Fig. 4.1, when our application established three TCP connections to another region, we noticed that the latencies were not only different among the three connections but also changed over time and in a step-wise manner. Many questions arise from this measurement: *(i)* are path changes the cause of the steps in the RTT trace? if so, *(ii)* how and what is the persistence of each path? *(iii)* what is the difference between the minimum and maximum observed latency, *(iv)* are there source ports that experience long-term better RTTs (and One Way Delay (OWD) latencies) with respect to other ports? *(v)* do we observe the same behaviour across all region pairs and to what extent?

In order to answer these questions, we performed a systematic set of measurements of the *AWS* inter-region network. We note that, while we suspect that Amazon *AWS* uses hash-based traffic splitting mechanisms, our measurements and conclusions do *not* rely on any specific assumptions of how the traffic splitting mechanisms are implemented. Our first goal was to identify whether the step-wise latency changes correspond to actual path changes (possibly due to TE operation) and analyzing the frequency and extent of latency changes both *spatially* — across different flows — and *temporally*.

4.2.1 Detecting TE activity

To detect latency changes that occur as a result of path changes, we introduce a methodology for filtering out congestion noise.

Measuring RTT and OWD flow latencies. Before delving into the intricacies of the algorithm, we first describe our measurement setup. We performed both RTT and OWD measurements using TCP and UDP flows. On all Virtual Machines (VMs) we use `chrony` configured to access the Amazon Time Sync Service [142], which provides high precision time synchronization. As we do not have visibility inside the Amazon *AWS* network or to its routing, we assume flows can use different forward and backward paths for their traffic. As such, from here on, we use the term "path" to simply refer to the joint forward and backward communication paths used by a flow for the RTT measurements and the forward path only for OWD measurements.

Decoupling propagation delays from congestion. In Fig. 4.1, one can easily observe that the latency experienced by a flow consists of a *base propagation* delay (due to traversing a certain geographical distance over a given routing path) and spurious congestion delay. In this figure, we identified several moments (each denoted by a black dashed vertical line) where the minimum latency observed during a certain time window (i.e., the base propagation latency) suddenly changes by at least 0.5 ms.

One way to infer TE activity in the cloud's backbone network is to detect these sudden changes in the base propagation latencies while filtering out the congestion component — which otherwise might be falsely interpreted as path changes resulting in an overestimation of TE activity. This problem is well-known and several techniques have previously been proposed to extract base propagation delay from latency measurements when possible (i.e., when congestion is limited and buffers periodically drain) [132, 133, 134]. Roughly speaking, these techniques are based on computing a rolling minimum of the last k observed latencies. Using this approach, we were unable to achieve a 0% rate of false positive, i.e., avoiding non-existing path changes. The minimum false positive rate that we achieved was 7.69% with a rolling window of $k = 20$. Therefore, we enhanced the rolling minimum technique as we will explain later in this section. It is also worth noting that using traceroutes to detect paths [136] requires cooperation of the cloud provider and can fail to correctly detect paths when routers do not respond with ICMP errors.

We first present our technique to extract base propagation latencies from *AWS* measurements while removing congestion. Then we validate the accuracy of this technique to detect TE activity (i.e., path changes). We want to stress that we do not claim our technique is general but rather we tailored our path change detection to the *AWS* network and its observable congestion profile. We further discuss this aspect in Section 4.5.

Our approach to detect path changes. We use a conservative approach that computes the "mode" in a sliding window (see Fig. 4.2 for an example). For each flow, we first aggregate the measured latencies (top-left part of the figure) across 1 second intervals by computing the minimum latencies across 5 probes*. We then apply our sliding window mode-based function using a window size of 4 seconds. While sliding our window we maintain an estimate of the current base propagation latency. If 3 out of 4 observations in a window are within 0.5 ms of the minimum value in the window, we say that the minimum is *stable*. Two cases are possible: (a) a window has a stable minimum or (b) not. In case (a), i.e., the minimum is stable, if the minimum is also within $\pm$ 0.5ms from the currently estimated base

* Probes are sent at 200 ms intervals

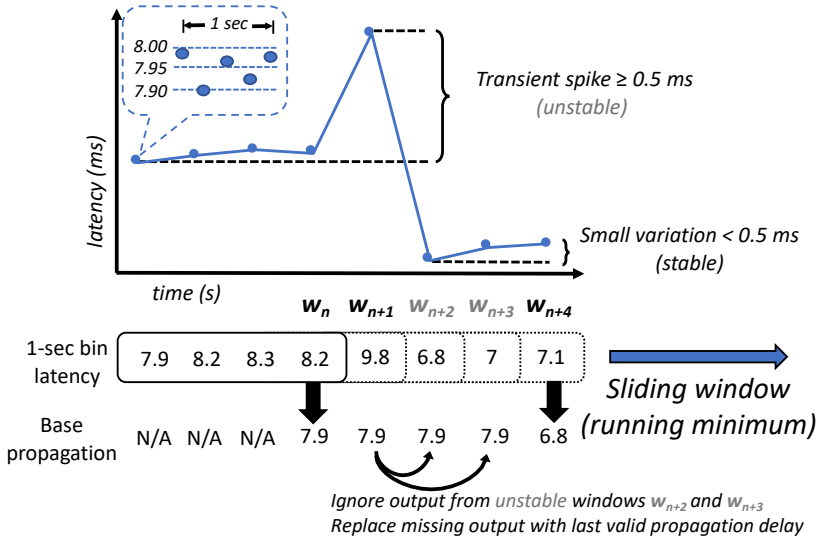


Figure 4.2: An overview of path detection using a mode-based sliding window that computes a running minimum. We depict an example where transient spikes ($\geq$ 0.5 ms) are filtered out, whereas actual base propagation changes (with stable latencies) are eventually detected.

propagation latency, we assume that the estimated base propagation latency is stable and do not modify it. Otherwise (i.e., the new minimum is not within $\pm$ 0.5ms from the estimated propagation latency), we update the propagation latency to the new minimum. In case (b), i.e., the minimum is unstable, hence we ignore the value and keep the estimated base propagation latency unchanged.

In Fig. 4.2, the minimum of 7.9 ms in window w_n is stable since at least 3 latency samples in the window are within 7.9 ± 0.5 ms. Therefore, we update the estimated propagation latency with this stable minimum. Window w_{n+1} is stable and the minimum of 8.2 is within 7.9 ± 0.5 ms. Therefore, we do not update the base propagation estimate. Both windows w_{n+2} and w_{n+3} are unstable because of the three samples 6.8, 8.2, and 9.8ms. For this reason, we keep the previously computed base propagation latency as our estimate. Window w_{n+4} is stable and the minimum of 6.8 ms is not within 7.9 ± 0.5 ms; therefore, we update the new base propagation latency with this new value.

To further remove congestion noise, we cluster our latency classes into 1-millisecond bins, by rounding to the nearest millisecond. We then compute the churn — or number of flows admitted — to all observed clusters. If a cluster has less than 0.1% of the sum of all cluster admissions, we conservatively consider it

as congestion noise and remove it from our measurements. We presume that, in these cases, such events are likely to occur if there is persistent congestion — with minimal latency oscillations which can result in false positives. As such, given that Amazon offers 99.9% availability for most of their inter-region services, we opt to remove clusters receiving less than 0.1% activity. We refer to the sequential execution of the mode sliding window followed by the above filtering technique as the *path change detector*.

Note that if a flow is moved to a path with the *same* latency, we are unable to detect this change, thus our conservative approach has false negatives. In the next subsection, we show that our approach can accurately distinguish path changes from congestion events in the *AWS* network. This eliminates false positives in path change detection.

4.2.2 Accuracy of path change detector

To verify that the latency changes observed by our measurements (e.g., Fig. 4.1) are due to routing changes in the network (and *not* caused by our measurement technique, congestion, cloud interference, etc.), we performed three sets of experiments: (i) we measured RTTs among VMs *within* one availability zone between two different DCs, (ii) identified and correlated packet reordering across the *AWS* WAN with the changes in the extracted base propagation latencies from RTT measurements, and (iii) identified and correlated changes in network paths using traceroute with measured OWD. Additionally, we discussed our findings with cloud networking experts, who had read an early draft of the manuscript for this project [7], and they supported these findings.

Latency measurements within the same region. First, we measured network RTT between different VMs located in the same region but in different availability zones. Our results show that intra-region latencies are stable and do not experience changes such as those seen in Fig. 4.1. We conclude that latency fluctuations are caused by traversing the inter-region cloud backbone network.*

Packet reordering correlation. Next, we deployed a pair of `c5.large` VMs in the Oregon and Virginia *AWS* regions. Using our custom traffic generator (deployed in Oregon and Virginia) we measured RTT by sending UDP packets every 0.5 ms continuously for 2 hours. Each packet had a unique ID and a monotonically increasing sequence number. We note that no packet was dropped during the experiment. By identifying inconsistencies in the sequence numbers at the receiver, we could detect when packets arrived out of order. Next, we

* For brevity, we do not show these results, but all samples can be found in [143].

correlated the instances of out-of-order arrivals with the observed RTT between VMs in Oregon and Virginia. The top graph in Fig. 4.3 shows a 2-hour snapshot of RTTs between Oregon and Virginia. The red vertical lines indicate when a sequence of packets arrived out of order. Note that such reordering events are strongly correlated with *decreases* in measured RTT. In some cases, possibly due to asynchronous network updates, we observe that the base propagation latency may quickly oscillate between two values before stabilizing to one of them. These events may mistakenly give the impression that, in Fig. 4.3, we observed packet reordering events when the base latency increased, as the path latency decreases are indistinguishable.

As discussed in Section 4.1, when a network flow is moved (e.g., due to TE) to a path with lower latency, subsequent packets can overtake the in-flight packets on the higher latency paths; thus, they arrive out of order. In contrast, moving to a path with a higher latency does not affect the order of packet arrivals. Therefore, packet reordering events can only be used to detect path changes to lower latency paths. Furthermore, routing changes that cause RTT decreases of less than 0.5 ms (e.g., at 14:00 on the x-axis) may not be detected by our measurement as packet probes are transmitted exactly every 0.5 ms. We recall that since we do not observe any packet drops, packet reordering can only occur due to path changes or some specific switch's packet schedulers; hence, they are not due to congestion in the network. By observing a high correlation between packet reordering events and latency decreases, we exclude those reordering events due to specific packet schedulers at the switches. Increasing the frequency of probe generation could increase the sensitivity of our measurement.

To evaluate whether our path change detector detects all the path changes associated with packet reordering (and thus changes to lower latency paths), we fed as input the packet trace measurements to our path change detector and plotted the output in the bottom subplot of Fig. 4.3. We plotted a dotted red line when our path detector detects a negative/positive latency path change. We observe that our path detector accurately detect the vast majority of the negative latency path changes corresponding to packet reordering events (some of them may not be detected due to noise in the latency signal). More importantly, out path change detector *never* reports a non-existing latency-decrease path change, thus achieving zero false positive rate. We note that the packet reordering correlation cannot be used to verify the accuracy of latency-increase path change detection.

Traceroute correlation. Finally, we identified and correlate changes in network paths (using traceroute) for a sample TCP flow between VMs in Oregon and Virginia with the measured OWD.

We established a TCP flow between these two regions and sent ping probes at

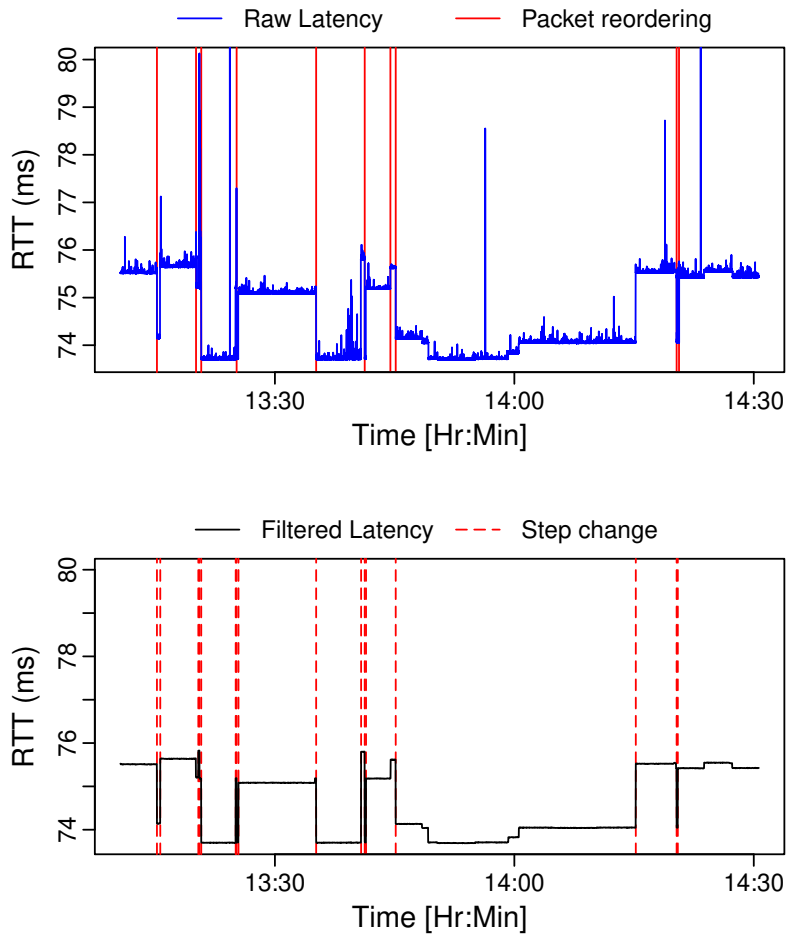


Figure 4.3: Measuring path detection accuracy using reordering events. The top plot shows observed RTTs and packet reordering occurrences whereas the bottom plot shows the filtered RTTs computed via our path change detector as well as the detected path change events.

a rate of 5 per second to measure RTT and OWD. In parallel, we ran traceroute measurements that were obtained using our custom traceroute (based on the principles described in [136]). The traceroute crafted network packets that matched the 5 tuples of the sample TCP flow to guarantee that both packets from the traceroute and the TCP flow would follow the same network path.*

We ran traceroute every 20 seconds and recorded the observed network paths. By analyzing the sequence of measured paths we identified the moments when changes in routing happened. These events were correlated with the OWD latency measured using TCP flows. Fig. 4.4 shows the OWD network latency with routing changes shown as vertical red lines. This figure illustrates that every change in OWD is accompanied by a change in the forward network path (unless the duration of such a change was less than 20 seconds as the traceroute would miss such event). Note that the opposite is not always true; as two distinct network paths may have nearly identical network latency, thus resulting in no network latency change.

As with packet reordering, we show in Fig. 4.4 that our path change detector identifies all the traceroute path change events that incur a latency change of at least 0.5 ms.

4.2.3 Measurements description

We have so far established that it is feasible to rely on changes in the base propagation latency to accurately detect path changes in the *AWS* network. We now discuss in detail the set of long-term measurements performed to answer the measurements goals stated at the beginning of this section.

Our measurements can be broken into two sets: (i) *Macro-scale* measurements that collect all the observable base propagation latencies between 120 unique Amazon *AWS* region pairs and (ii) *Micro-scale* measurements that are used for analyzing in deeper detail the performance of the individual flows (e.g., number of path changes experienced) and compare among flows (e.g., to examine fairness) during the complete duration of the measurement. In the first set of measurements, for each pair of regions, we select two machines in two different regions and create a large number of flows using many different source ports. We randomly pick a set of 512 ports every 30 seconds. The goal is to compute the maximum difference between observed base propagation latencies among all the DC pairs. The second set of measurements targets a limited number of selected region pairs, but generates a greater number of probes per second in order to detect latency changes at a fine-grained temporal resolution. To gather per-flow statistics, we randomly select 512 ports and keep them unchanged for the entire duration of the measurement. We describe the specifics of our experimental configurations below and summarize

* This is the only experiment where we assume the presence of per-flow load balancing in the network.

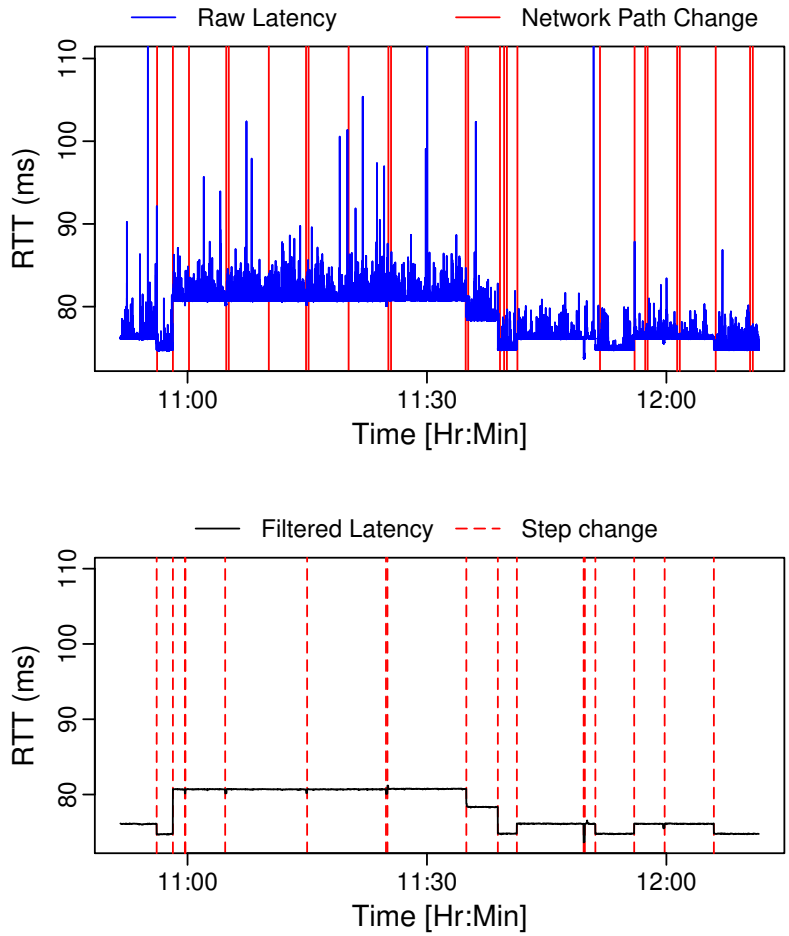


Figure 4.4: Measuring path detection accuracy using inflow traceroutes. Top plot shows path changes captured via traceroutes and bottom plot shows same output as Fig. 4.3.

them in Table 4.1.

Table 4.1: Testbed configuration for measurements.

Config.	Macro-scale	Micro-scale
# of DC Pairs	120	4
# of Flows	512	512
Probing Rate	10 probes every 30s	5 probes/s
Flow Generation	Dynamic (every 30s)	Static
Duration	2 days	1 week
Ping Mechanism	Raw Sockets	TCP Ping
Results	Fig. 5	all except Fig. 5 & 15

Testbed. We use two distinct setups to conduct our macro and micro-scale measurements on *AWS*. Our macro-scale testbed probes paths between 16 regions located in 16 different Amazon *AWS* regions. We create `c5.large` EC2 instances in each region and connect them all-to-all, for a total of 120 unique region pairs. In contrast, our micro-scale measurements are deployed in a more limited setting and focus on only 4 region pairs.

Probing. In our macro-scale experiments, we use raw sockets to perform an exhaustive search of base propagation latencies by emulating TCP connections. This allows us to efficiently probe paths by creating custom TCP packets with different source ports *without* being bound by the maximum number of TCP connections that can be created on a given VM instance. We perform a search by picking 512 random ports every 30 seconds. For our micro-scale measurements, we focus on only four region pairs — selected among the top 10% pairs in terms of greatest differences in the previously observed propagation latencies — and no longer emulate TCP connections, but instead rely on hash-consistent TCP ping, where we maintain 512 static TCP connections between each regions pair. We verified that our probes are never retransmitted twice during our measurements, e.g., due to packet drops.

4.3 Analysis of the Results

We showed in Section 4.2 that we can use our path change detector to extract the base propagation latency of the path through which a flow in being routed as well as detect when the flow is rerouted over a different path. We refer to *flow latency* as the base propagation latency of the path through which the flow is being routed.

We say that a flow *changed path* if the flow latency changes by at least ± 0.5 ms, as described and validated in Section 4.2.

In this section, we evaluate the frequency and magnitude of the flow latency changes observed between *AWS* regions using the measurements obtained from both the macro- and micro-scale experiments described in Section 4.2. These results shed light on the extent to which TE operation is performed within the *AWS* cloud backbone network and the assumptions that can be made by cloud application developers. Our main findings can be summarized as follows: *(i)* between two machines there may exist many different path with diverse latencies (Section 4.3.1), *(ii)* some paths may suddenly become unavailable (Section 4.3.2), *(iii)* half of the observed paths persist for 10 seconds or less (Section 4.3.3), *(iv)* the lowest latency paths have the highest flow churn (Section 4.3.3), and *(v)* some flows may *consistently* experience worse propagation latencies than others (Section 4.3.4). Results (ii-v) are based on the micro-experiments, while result (i) is based on the macro-experiment.

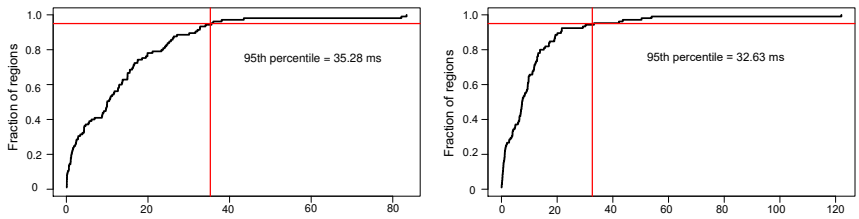
4.3.1 Flow latencies vary greatly across regions

To understand the spectrum of possible base propagation path latencies experienced by a flow, we first measure the highest (max) and lowest (min) flow latency for each of the unique region pairs as described in the macro-scale paragraph in Section 4.2. Fig. 4.5(a) shows a CDF of the distribution of $max - min$ latency differences across all DC pairs. We see that the median of these differences is at 10 ms, but can increase to roughly 35 ms at the 95$^{\text{th}}$ percentile.

While these differences appear to be non-negligible, it is hard to reason about their significance in isolation since the minimum inter-region latencies can be 10s to 100s of milliseconds. In order to put these numbers into perspective, Fig. 4.5 (b) shows a CDF of the *relative* $\frac{max}{min}$ flow latency percentages; where the latency change is computed as the percentage increase from the lowest (min) to the highest (max) flow latency observed for each region pair. We can clearly see a heavy-tailed distribution, with changes of up to 32% at the 95$^{\text{th}}$ percentile. When such *large jumps in latency occur* this can negatively affect services that require consistent inter-region request-response latencies.

4.3.2 Availability of paths

While it is clear that the maximum latency change can be quite profound, this value is computed based on the best and worst flow latencies observed during a period of 2 days. However, it is not immediately clear whether these latency classes are available universally or only during certain time periods. To evaluate this, we took a closer look at four region pairs, namely: “Oregon-Virginia”, “Sydney-Tokyo”,



(a) Latency difference (ms) (b) Relative latency change (%)

Figure 4.5: CDF of the (a) absolute differences ($max - min$) and (b) relative percentages $\left(\frac{max}{min}\right)$ flow latencies across all the *AWS* region pairs.

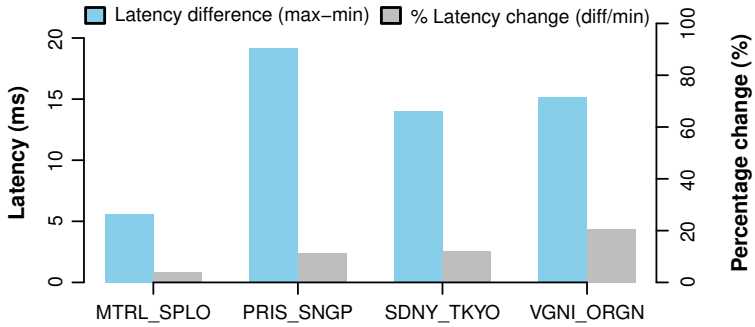
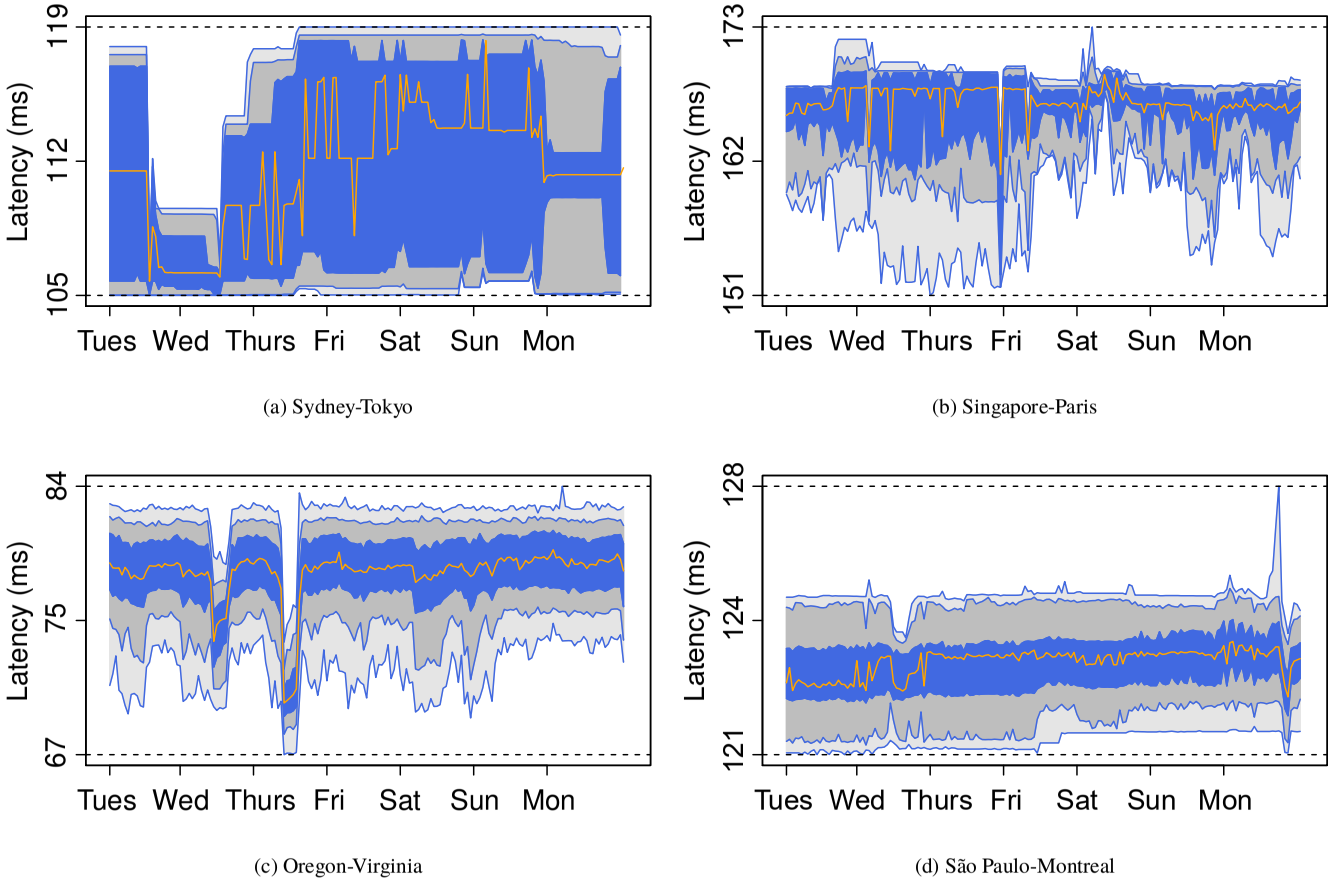


Figure 4.6: Absolute and relative latency changes for four different Amazon *AWS* DC pairs.

"São Paulo-Montreal", and "Singapore-Paris". We chose these pairs to represent different continent combinations (with the exception of "Oregon-Virginia", where both are in the US) from the top 50^{th} percentile of regions pairs that showed the largest absolute latency differences. We plot the latency differences for these regions pairs in Fig. 4.6 where "Oregon-Virginia" has the highest percentage change at 40% and "Sydney-Tokyo" being the lowest at roughly 10%.

To show the availability of paths across time, in Fig. 4.7 we plot the flow latency percentiles of 512 flows observed for each region pair over time. The blue area corresponds to the inter-quartile, the bottom (upper) dark gray area corresponds to the 5^{th} to 25^{th} and the 75'th to 95^{th} interpercentile range, and the bottom (upper) light gray area corresponds to the 0^{th} to 5'th and the 95^{th} to 100^{th} interpercentile range. The plotted latencies are the average base propagation latencies obtained from the path change detector (fed with the raw measurements) binned in 1 hour intervals, and thus do not include congestion-induced spikes. We can clearly see



(a) Sydney-Tokyo

(b) Singapore-Paris

(c) Oregon-Virginia

(d) São Paulo-Montreal

Figure 4.7: Changes in the distribution of latencies across 512 different flows binned in 1 hour intervals.

in all subfigures (with the exception of (a)) that the latencies experienced by the flows can change dramatically with time. The inter-quartile ranges (highlighted in blue) can shift sporadically over time, indicating that certain paths become less prevalent or harder to reach (e.g., due to increased load or failures). Moreover, looking at the minimum observed latency we observe that lowest-latency paths become unavailable over time. For instance, in Fig. 4.7 (c), we saw on Day 3 a new low latency path that lasts for only a few hours and then disappears for the rest of the week. This could be caused by TE operations that reallocate the paths across different region pairs or perhaps because of different types of customers.

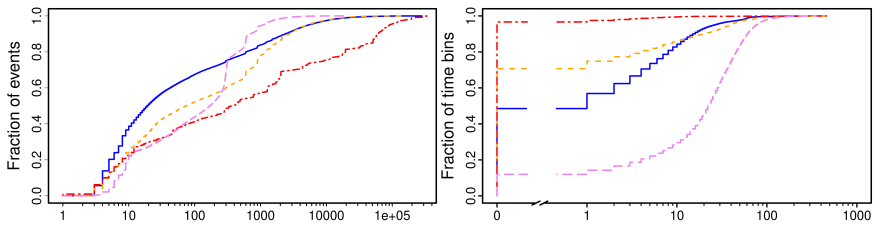
Summary. Flow latencies between a single region pair can change dramatically (by up to 32% at the 95$^{\text{th}}$ percentile). Moreover, some latency classes are unavailable during specific hours of the day, suggesting that specific paths become unavailable.

4.3.3 Flow latency persistence

In the previous section, we explored differences between flow latencies and how flows are distributed across paths varies over time. However, it is desirable to know *how often* these flow latencies change. In fact, frequent routing changes can be a problem both for cloud network operators as well as for a cloud's customers since frequent changes can have a negative impact on TCP flows due to packet re-ordering and inaccurate RTT estimation [116].*

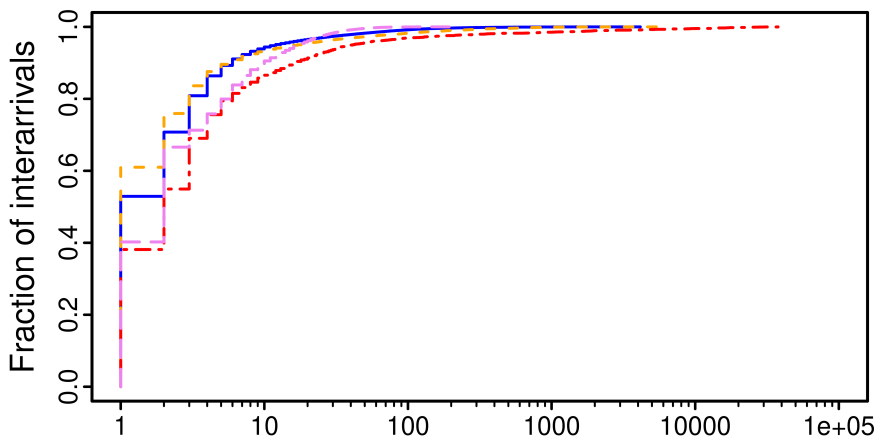
We study these changes by first looking into how long a flow persists in a latency class. Fig. 4.8(a) shows a CDF of flow latency durations for all flows for each region-pair. The y-axis represents the fraction of events in which a flow moved to a new path and persisted on that path for less than x seconds before being rerouted to a different path. Surprisingly, from this plot we find that in "São Paulo-Montreal" roughly 40% of flows moving to a new path continue to use this path less than 10 seconds before moving to another path. This can lead to packet re-ordering for flows longer than 10 seconds — which, to put into context, is more than 75% of the inter-region flows in Facebook's caching clusters [144]. In contrast, for "Sydney-Tokyo" flows change their paths rather infrequently and 50% of those flows moving to a new path continue to use that path for more than 280 seconds. As previously seen in Fig. 4.7, this region-pair rarely exhibits any changes across its flow latency distributions, suggesting that its paths are unlikely to be subject to frequent TE changes. In Fig. 4.9, we break down the flow persistence graph into the forward and reverse paths.

* Evaluating the impact of latency instability on different congestion control mechanisms is outside the scope of this work.



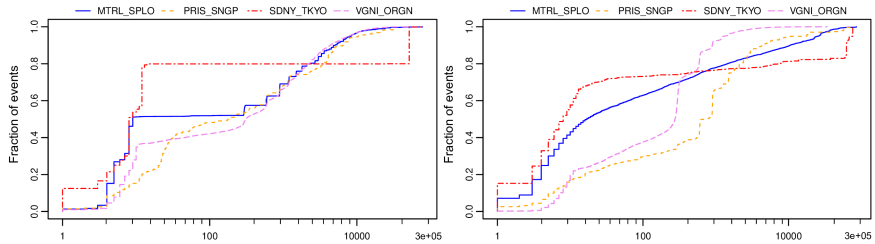
(a) Flow persistence (s)

(b) # of flows changing paths



(c) Flow path change interarrival time (s)

Figure 4.8: CDFs showing characteristics of traffic class changes across all the monitored flows. Subfigure (a) shows every path change event of a flow, (b) shows the # of flows that experienced at least 1 path change in each 20 second time interval bin, and (c) shows the interarrival time between any two path change events observed in the network. Legend: São Paulo-Montreal ———; Paris-Singapore - - - -; Sydney-Tokyo -·-·-; and Oregon-Virginia - - - -.



(a) Forward flow persistence (s)

(b) Reverse flow persistence (s)

Figure 4.9: CDFs showing asymmetry of flow latency class persistence across all 512 monitored flows. Subfigures (a) & (b) report results for the latency classes on the forward and reverse paths, respectively.

Secondly, we examine whether flows change their paths in tandem, i.e., whether flows move to different paths at the same time. Fig. 4.8(b) plots the number of flows (out of 512) that changed their path at least once during a 20-second interval. The y-axis presents the fraction of 20 second time interval bins within which at most x flows change their path in a single bin. We find that, "Singapore-Paris" and "São Paulo-Montreal" exhibit an *all-or-nothing* property where either the majority of flows are affected or none of them are affected. This indicates that TE events, when triggered, can dramatically impact a large population of flows, perhaps due to failures in the network, planned topology augmentation, an unwanted cascading effect — where changes in the traffic matrix caused by the initial TE events trigger successive waves of routing changes — or even asynchronous network updates. In contrast, "Oregon-Virginia" ("Sydney-Tokyo") exhibits many small TE operations during (50% of) all the 20-second intervals. To further investigate these cases, Fig. 4.8(c) shows the inter-arrival time between flow path changes. In this figure, the y-axis shows the fraction of path changes events across all flows such that the time between that path change event and the next path change event (possibly of a different flow) is x. Similar to the previous plot, "Oregon-Virginia" also demonstrates unique characteristics. On average, its flows change paths more frequently — with interarrival times of inter-flow path changes of less than 5 seconds at the median. Moreover, this rate of changes can increase leading to many path changes in 150 ms. This may suggest different subclasses of TE events — with major events (likely in response to significant traffic changes) occurring less frequently but affecting more flows. Minor TE events might be triggered in reaction to smaller congestion events and consequently only need to reroute a small subset of the flows (e.g., by slightly modifying traffic splitting ratios).

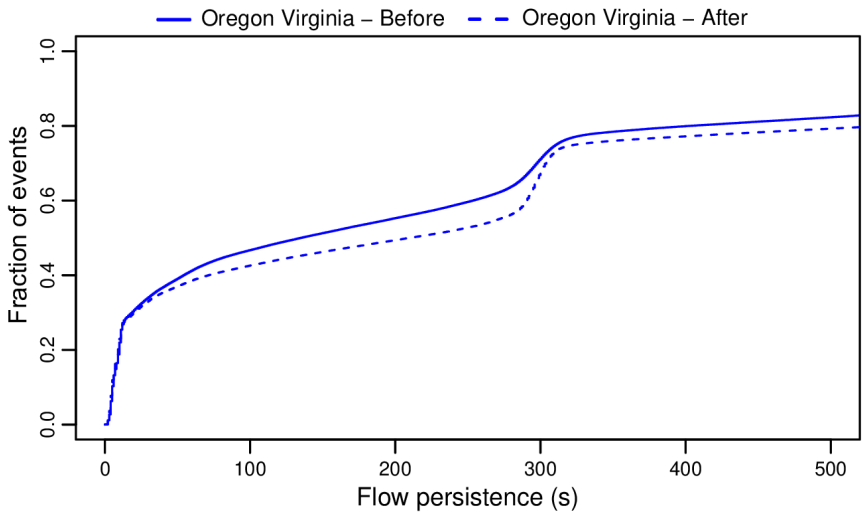


Figure 4.10: CDF of class durations before and after a TE policy change by Amazon.

Different TE mechanisms impact flow persistence. Based on discussions with cloud networking experts, we decided to verify how the persistence of the flows would vary during different times of the year. We measured the *AWS* network for several weeks, starting from December 2018 until the end of February 2019, for the critical pair of Oregon and Virginia regions. We observed a significant change during the month of February. Conversations with cloud networking experts suggested that this could have been due to a change of the *AWS* configuration to a less reactive TE after the high-load period of the Christmas holidays and January. We plot the CDF of the flow persistence (similarly to Fig. 4.8 (a)) in Fig. 4.10 for two distinct weeks of February when we observed the change. While the median flow persistence clearly decreased by roughly a factor of 1.5, one still observes a very low path persistence for $\sim$30% of the flow path change events. However, a thorough investigation of these results is outside the scope of this work.

Are flow latencies correlated with flow churn? While flows can change paths quite frequently (in the order of seconds), we have not yet studied the frequency of path changes of a flow with respect to its flow latency. Specifically, do flows on low latency paths experience higher *flow churn*, i.e., the number of flow rerouting events on a specific path. To investigate this we group paths by rounding the path latency and call the rounded latency a *latency class*. We choose the relatively stable region pair of São Paulo-Montreal to see how flows compete for paths in steady

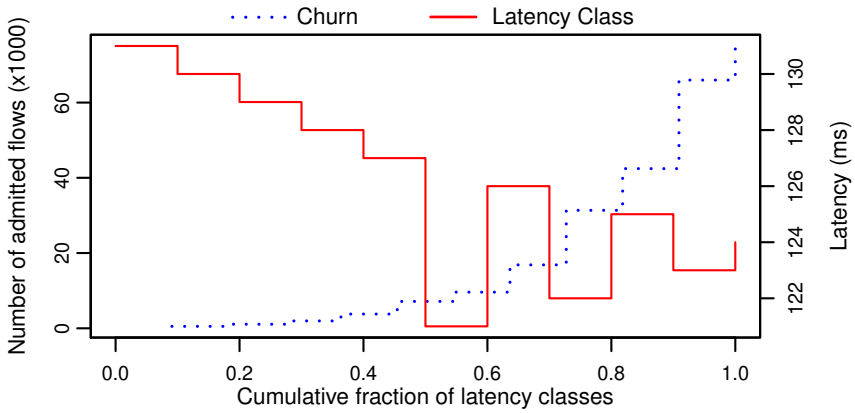
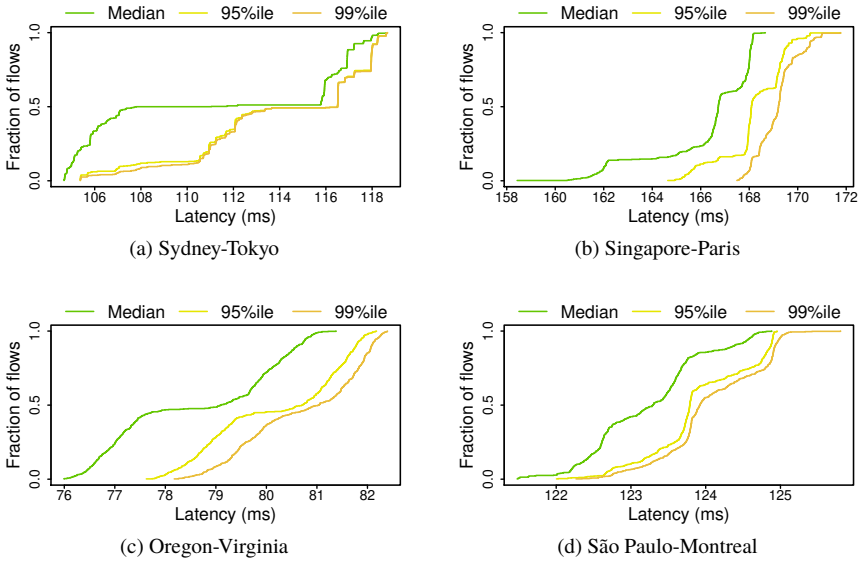


Figure 4.11: Flow churn observed for latency classes found on São Paulo-Montreal. The blue dotted line is a CDF of the number of admitted flows reported by each latency class. The red line is the latency of the associated class.

state. We show in Fig. 4.11 a CDF (blue dotted line) of the flow churn of each latency class. The x-axis shows all the observed latency classes, ordered from left to right by increasing number of flow churns. The corresponding latency of each latency class is shown by the red solid line. We can clearly observe a negative correlation, where paths belonging to low latency classes exhibit higher flow churn (with only a few outliers). We reason that this could occur due to flows being opportunistically routed to low latency paths (provided they are available) with the possibility of subsequently being preempted by higher priority flows. Therefore, given the increased competitiveness for low latency paths, flows experience more churn on such paths, while the reverse is true for higher latency paths. This type of greedy shortest-path TE mechanisms have been widely used in MPLS-TE [15, 110] and B4 [112] networks. Our results suggest that the Amazon *AWS* network uses similar TE mechanisms, although we did not get confirmation of this from the *AWS* team.

Summary. We have shown that flows change their latency classes frequently and more than 50% of the flows moving to a new path change paths within 10 seconds. We have also established that these changes often happen in tandem in certain region pairs. We presented our results to cloud networking experts and they confirmed that low-latency paths tend to experiences higher flow churn, suggesting TE tends to reactively move flows to low latency paths whenever capacity is available. However, as noted earlier, these frequent path changes can negatively impact TCP's congestion control algorithm and ultimately tenant applications.



(a) Sydney-Tokyo

(b) Singapore-Paris

(c) Oregon-Virginia

(d) São Paulo-Montreal

Figure 4.12: Distribution of flow latency percentiles on 512 monitored flows. Flow latencies can differ, even over two days. This indicates that some flows are consistently unfairly penalized *despite* frequent path changes.

4.3.4 Flow latency fairness

The previous sections highlighted issues due to short persistence of flows on paths. However, one might assume that frequent latency class changes means that flows are unlikely to have unfavorable path assignments for prolonged durations. Unfortunately, our results show that this is not the case.

Fig. 4.12 shows a CDF of the median, 95$^{\text{th}}$, and 99$^{\text{th}}$ percentile base propagation latencies of all the flows over a one week timespan. We see that these plots resemble a step function, but with significant differences among flows. For instance, in Fig. 4.12(d), the median latency changes from 105 to 118 ms — a 12% increase. The same applies for the higher percentiles as well.

To further study unfairness, we also look at path changes. Fig. 4.13 shows a CDF of the total number of path changes (over the same one week interval) experienced by different flows. Similarly, we see that certain flows are more likely to exhibit more frequent path changes, thus further exacerbating the level of unfairness among flows.

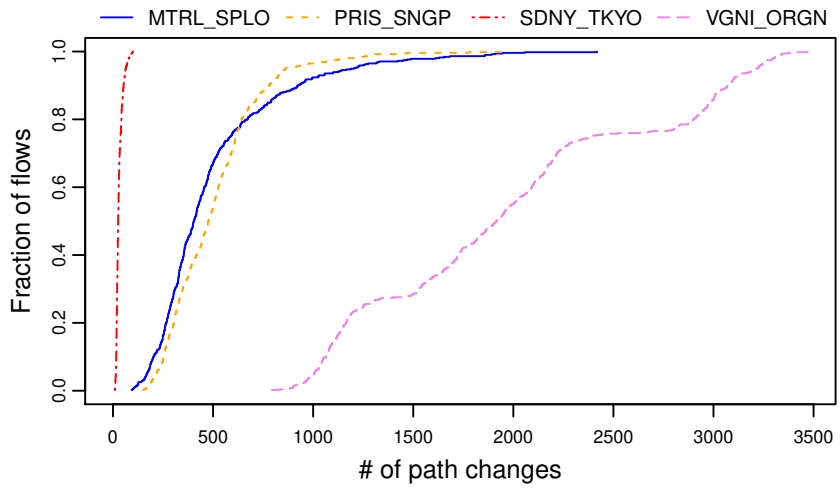


Figure 4.13: CDF of path changes experienced by different flows.

Summary. These results indicate that *despite* frequent path changes, flows are treated unfairly, with some persistently experiencing poor performance. One reason for this unfairness may lie in the specific implementation of weighted traffic-splitting mechanisms. For instance, implementations based on tables and buckets [125] may tend to rarely move some of the flows, thus leaving them in a poor latency class for a prolonged time.

4.3.5 Impact on application performance

We also conducted application benchmarks between instances deployed in Oregon and Virginia. Before discussing these results, it is relevant to look into path change frequency for a given latency change threshold, which we refer to as *spike*, since some applications may not be affected by smaller latency spikes (i.e. < 1 ms). Fig. 4.14 shows a CDF of the latency persistence for spikes of at least 1 ms, 2 ms, and 5 ms for pings between Oregon and Virginia. We can see that the median flow persistence at 1 ms is ~170 seconds, and this increases up to ~300 seconds at a 5 ms spike threshold. This shows that flows running for more than a few minutes can experience up to 5 ms spikes — almost a 10% latency increase. Spikes less than 5 ms occur infrequently and are not shown in this graph.

For our application benchmarks, we used the *rsync* utility — which is popularly used for mirroring and performing incremental snapshots across storage nodes. We configured our benchmark to transfer 100K files between instances running in both

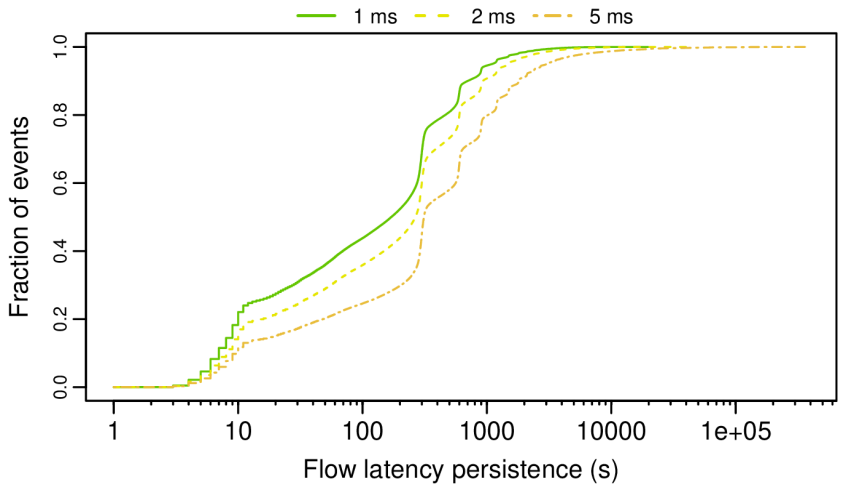


Figure 4.14: Flow latency persistence expressed in terms of a given "spike" threshold.

regions, with each file having a constant size of 1 KB. For each rsync run, we use two distinct strategies to choose the source port for the rsync connection: (a) *Random*: This is the default strategy, where the source port is picked by the OS from the ephemeral range of available ports. (b) *Minimum RTT*: This strategy probes 128 different TCP ports for 10 seconds prior to each rsync run. The port with the lowest average RTT is chosen and a NAT rule is inserted to force the next rsync call to utilize the selected port. We repeated these experiments 500 times and collected the runtimes of rsync calls. Fig. 4.15 shows a CDF of the rsync runtimes for both configurations. We can see that, even when applying a simple minimum RTT strategy for preferential port selection, we observe a ∼7% reduction at the median. In addition, the resulting rsync performance is more predictable — notably, the 99th percentile for Min. RTT is lower than the median for Random and the standard deviation drops by a factor of 2.8x, *i.e.*, from 259 ms to 90 ms. By analyzing TCP dumps of this traffic, we observed that these differences are mainly due to the RTT of the assigned path — which directly impacts the throughput of TCP. We also note that this is not an artifact of TCP slow start, since rsync maintains the same TCP connection across an entire run (spanning 100K files). The impact on shorter flows should be even more noticeable and we leave this for future work.

Cloud provider tenants can arguably use similar strategies to reduce latency for their geo-distributed applications. However, this could increase unfairness, as tenants that do not utilize this information would be prone to using unfavorable

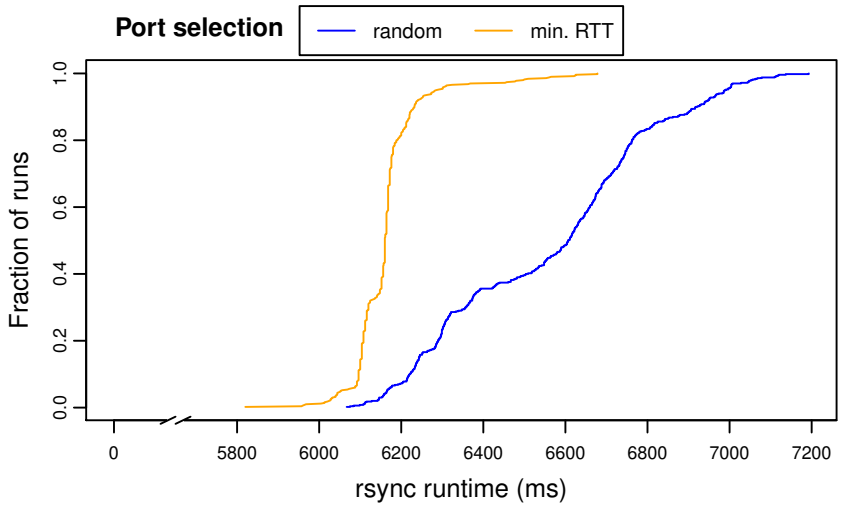


Figure 4.15: CDF of rsync runtimes for transferring 100K files between Oregon and Virginia. Two different strategies for port selection are shown.

path assignments. Moreover, this behavior might be at odds with the TE of the cloud provider — which could be trying to move flows away from congested paths. This raises the important question as to whether cloud providers should make path allocations more transparent and/or allow tenants to control such allocations (perhaps for a price). A game theoretic analysis could aid in answering such questions and we leave this investigation as future work.

Summary. Application performance can be hampered by unfavorable path assignments. Tenants might be incentivized to actively pick ports with the lowest observed latencies in order to achieve better performance. However, this could also exacerbate unfairness among tenants and work antagonistically with the cloud provider's TE policies.

4.4 Augmenting Rein using Path Latencies

We so far presented a detailed analysis of the network characteristics in Amazon *AWS* and tools we used to extract these results. One question we asked ourselves is whether we can leverage our path change detector to further improve upon the performance of the Rein scheduling system (presented in Chapter 3).

To recap, Rein uses two heuristics to schedule multiget requests efficiently in

storage systems. It uses a Shortest Bottleneck First policy (SBF) to allow smaller multigets to bypass larger "elephant" multigets. This reduces head-of-line blocking and allows smaller multigets to finish faster. Our second policy of Slack-driven Scheduling (SDS) decides how the different parts of a multiget are scheduled. Since data can be sharded across different servers, each multiget is normally split-off into smaller fragments called *opsets*, which comprise all the requests going towards a specific server. Each opset may have a different size. The SDS policy aims to *slack*, or deprioritize, smaller opsets as there is no need for them to finish sooner than the largest "bottleneck" opsets. In Rein, this policy has so far taken only the opset sizes into account.

For geographically-distributed services [65], opsets may be sent to servers hosted in different geographic areas. In this case, differences in network latency (not just size) may impact the completion time of each opset, and it can be opportunistic to incorporate this information in the SDS policy. To this end, our path change detector can be used to provide clients with accurate real-time network latency estimations for each region. The clients can then combine this information with the opset size to compute the expected completion time of each opset. This information can be used to discern by how much each opset should be slacked, and it can be tagged to each opset and then applied by the scheduler once the opset arrives at the storage server. This can aid in making Rein's existing policies more latency-aware, and ultimately more efficient.

4.5 Discussion

Comparisons with other Cloud Provider networks. This work does not aim to compare different cloud providers' networks. We merely highlight the presence and effects of very reactive TE mechanisms in the largest cloud provider's network. However, we also ran packet reordering experiments on Google Cloud for its corresponding Oregon-Virginia pair. Our results indicated fewer path changes, only occurring on the order of tens of minutes, leading us to believe that they utilize less reactive TE policies for this particular pair. This is possibly due to over-provisioning and/or more predictable traffic demands as most applications in the Google network are controlled by Google and not tenants. We leave the study of other cloud providers' networks as future work.

Tuning the path change detector. We tuned our path change detector specifically for the Amazon *AWS* network by setting the path change threshold to 0.5ms and validated this value with additional measurements (see Section 4.2). We note that the path detector must be carefully tuned and validated for each network,

using the techniques described in Section 4.2.2. We leave this task as future work.

Impact of VM instances. One may wonder whether the latency variations are due to delays in the *AWS* VM instances. As mentioned in Section 4.2.2, we performed latency measurements within the same region and observed stable latencies. Furthermore, our packet reordering measurements further confirmed the observed latency decreases were due to path changes.

Impact of different packet types (e.g., TCP, UDP). Given the inherent blackbox measurements of this work, we asked ourselves whether some of our traffic was affected by the fact that we used carefully crafted TCP and UDP packets. We shared the results with cloud networking experts and were able to corroborate that what we observed was due to TE activity and not simply an artifact of our probing technique.

Use of flowlet switching. We have conducted packet train experiments that transmit UDP packets at up to 5000 packets per second to see whether the *AWS* network uses flowlet switching [145] — a load balancing technique that aims to preserve flow path assignments for larger batches of packets. Even at these higher packet rates, we still observe packet reordering, which leads us to believe that the network does *not* employ flowlet switching.

4.6 Conclusions

The growth of Internet applications with low-latency and high-bandwidth requirements places tremendous challenges on network operators. To investigate our observations of latency variations, we performed a large-scale measurement of the Amazon *AWS* network and devised techniques to accurately detect path changes. Our results unveiled some surprising results. TE mechanisms in this network seem to operate at approximately 10-second intervals, well below previously reported time scales. Consequently, flows of traffic may be subject to frequent and sharp latency changes as well as persistent unfair treatment. Flow latencies between a single region pair can change dramatically (by up to 32% at the 95^{th} percentile) and expose traffic to greater unfairness across flows. Finally, tenants have incentives to move their traffic to low-latency paths as demonstrated in our *rsync* use case. We believe this work will spur discussions on the impact of high-frequency TE on the design of congestion control mechanisms and cloud applications.

Part II

Leveraging State-of-the-Art Hardware

Chapter 5

Fast Distributed File System IO using Client-local NVM

The advent of byte-addressable non-volatile memory (NVM), such as Intel's Optane DC persistent memory module (PMM) [18], promises to change the landscape for distributed storage. NVM provides high-capacity persistent memory with near-DRAM performance at lower cost. The potential of NVM as a low-cost main memory add-on is driving the adoption of node-local NVM at scale [21, 22, 23]. Remote direct memory access (RDMA) allows NVM access across the network without CPU overhead, raising interest in NVM for high-performance distributed storage.

A common paradigm in distributed file systems, like Amazon EFS [146], NFS [55], Ceph [49], Colossus/GFS [147], and NVM re-designs, like Octopus [50] and Orion [51], is to separate storage servers from clients. In this server-client design, files are stored by servers on machines physically separated from clients running applications. Client main memory is treated as a volatile block cache managed by the client's OS kernel. This design simplifies resource pooling by physically separating application from storage concerns with simple, server-managed data consistency mechanisms.

This simplicity comes at a cost, which becomes apparent as we move from SSD/HDD to NVM storage. In steady state, application performance is limited by the overhead to access kernel-level client caches. Upon a cache miss, multiple network round trips are needed to consult remote metadata servers and to fetch the actual data. On failure, client-server file systems must rebuild caches of failed clients from scratch, involving long fail-over times to re-establish application-level service and necessitating high network utilization during recovery. Third,

managing client caches at fixed page-block granularity amplifies the small IO operations typical of many distributed applications and increases cache coherence overhead when IO is larger than the block size. These costs prevent NVM from reaching its performance potential and have led some within the storage community to advocate for a complete redesign of the file system API [148, 149, 150, 151].

We present Assise, a distributed file system designed to maximize the use of *client-local* NVM without requiring a new API for high performance. Assise unleashes the performance of NVM via pervasive and persistent caching in process-local, socket-local, and node-local NVM. Assise accelerates POSIX file IO by orders of magnitude by leveraging client-local NVM without kernel involvement, block amplification, or unnecessary coherence overheads. Assise provides near-instantaneous application fail-over onto a *hot replica* that mirrors an application's local file system cache in the replica's local NVM. Assise reduces node recovery time by orders of magnitude by locally recovering NVM caches with strong consistency semantics. Finally, Assise leverages cluster-wide NVM via *warm replicas* that provide lower latency reads than slower storage media, such as SSDs. In cascaded hot replica failure scenarios, warm replicas can become hot replicas to preserve near-instantaneous fail-over.

To enable these properties, we design and build to our knowledge the first crash consistent distributed file system cache coherence layer for replicated NVM (CC-NVM). CC-NVM serves cached file system state in Assise with strong consistency guarantees and locality. CC-NVM provides prefix crash consistency [152] by enforcing write order to local NVM via logging and to cross-socket and remote NVM by leveraging the write order of DMA and RDMA, respectively. CC-NVM provides linearizability for all IO operations via leases [153] that can be delegated among nodes, sockets, and processes for local management of file system state. CC-NVM consistently chain-replicates [154] all file system updates to a configurable set of hot and warm replicas for availability.

Using CC-NVM, Assise achieves the following goals:

- **Simple programming model.** Assise supports unmodified applications using the familiar POSIX API with strong linearizability and crash consistency [152].
- **Scalability.** Unlike NVM-aware distributed file systems that are limited to rack-scale [51, 155], Assise provides strong consistency but remains scalable using dynamic delegation of leases to nodes, sockets, and processes; local sharing uses CC-NVM for consistency without network, cross-socket, or kernel communication.
- **Low IO tail latency.** To efficiently support applications with low tail latency requirements, Assise allows kernel-bypass access to authorized local and remote NVM areas. To reduce replicated write latency, Assise provides an optimistic mode using asynchronous chain replication with prefix crash consistency.

- **High availability.** Assise provides near-instantaneous fail-over to a configurable number of replicas and minimizes the time to restore the replication factor after failure.
- **Efficient bandwidth use.** The high bandwidth provided by NVM means that communication can be a throughput bottleneck (cf. Table 5.1). Assise minimizes communication by eliminating redundant writes [156] and reducing coherence protocol overhead via logging.

We make the following contributions:

- We present the design (§5.2) and implementation (§5.3) of Assise, a distributed file system that fully utilizes NVM by persistent caching in client-local NVM as a primary design principle. Assise uses client-local NVM to recover the file system cache for fast fail-over and locally synchronizes reads and writes to file system state.
- We present CC-NVM (§5.2.3), the first persistent and available distributed cache coherence layer. CC-NVM provides locality for data and metadata access, replicates for availability, and provides linearizability and prefix crash consistency for all file system IO.
- We quantify the performance benefits of using local NVM versus remote NVM for distributed file systems (§5.4). We compare Assise's steady-state and fail-over behavior to RDMA-accelerated versions of Ceph with BlueStore [157] and NFS, as well as Octopus [50], a distributed file system designed for RDMA and NVM, using common cloud applications and benchmarks, such as LevelDB, Postfix, MinuteSort, and FileBench.

Our evaluation shows that Assise provides up to 22× lower write latency and up to 56× higher throughput than NFS and Ceph/BlueStore. Assise outperforms Octopus by up to an order of magnitude. Assise scales better than Ceph, providing 6× throughput for Postfix with 48 processes over 3 nodes. Assise is more available than Ceph, returning a recovering LevelDB store to full performance up to 103× faster. Demonstrating that strong consistency with the familiar POSIX API and high performance are not mutually exclusive, Assise finishes a local external sort 3% faster than a hand-tuned implementation using processor loads and stores to memory mapped NVM. Finally, Assise finishes the MinuteSort distributed sorting benchmark up to 2.2× faster than a parallel NFS installation.

Assise supports networked access to remote storage where it makes sense. Assise can automatically migrate cold data that does not fit in NVM to slower, network-attached storage devices, such as SSDs and HDDs. To do so, Assise's implementation builds on Strata [156] as its node-local store.

Memory	R/W Latency	Seq. R/W GB/s	$/GB
DDR4 DRAM	82 ns	107 / 80	9.77 [158]
NVM (local)	175 / 94 ns	32 / 11.2	3.83 [8]
NVM-NUMA	230 ns	4.8 / 7.4	-
NVM-kernel	0.6 / 1 μs	-	-
NVM-RDMA	3 / 8 μs	3.8	-
SSD (local)	10 μs	2.4 / 2.0	0.32 [9]

Table 5.1: Memory & storage price/performance (October 2020).

5.1 Background

Distributed applications have diverse workloads, with IO granularities large and small [48], different sharding patterns, and consistency requirements. All demand high availability and scalability. Supporting these properties simultaneously has been the focus of decades of distributed storage research [49, 50, 51, 52, 53, 54, 55]. Before NVM, trade-offs had to be made. For example, by favoring large transfers ahead of small IO, or steady-state performance ahead of crash consistency and fast recovery, leading to the common idiom of remote-storage file system design. We argue that with the arrival of fast NVM, these trade-offs need to be re-evaluated.

The opportunity posed by NVM is two-fold:

Cost/performance. Table 5.1 shows measured access latency, bandwidth, and cost for modern server memory and storage technologies, including Optane DC PMM (measurement details in §5.4). We can see that local NVM access latency and bandwidth are close to DRAM, up to two orders of magnitude better than SSD. At the same time, NVM's per-GB cost is only 39% that of DRAM. NVM's unique characteristics allow it to be used as the top layer in the storage hierarchy, as well as the bottom layer in a server's memory hierarchy.

Fast recovery. Persistent local storage with near-DRAM performance can provide a *recoverable* cache for hot file system data that can persist across reboots. The vast majority of system failures are due to software crashes that simply require rebooting [159, 160, 161]. Caching hot file system data in NVM allows for quick recovery from these common failures.

For these reasons, data center operators are deploying NVM at scale [21, 22, 23]. However, to fully realize its potential, we have to efficiently use local NVM. NVM accessed via RDMA (NVM-RDMA), via loads and stores to another CPU

socket (NVM-NUMA), or via the kernel on the same socket (NVM-kernel) can be an order of magnitude slower in terms of latency and bandwidth.

5.1.1 Related Work

We survey the existing work in distributed storage and highlight why it cannot fully utilize the storage system performance offered by local NVM.

Block and object stores, such as Amazon's EBS [56], S3 [57], and Ursa [48], provide a new API to a multi-layer storage hierarchy that can provide cheap, fault-tolerant access to vast amounts of data. However, block stores have a minimum IO granularity (16KB for EBS) and IO smaller than the block size suffers performance degradation from write amplification [48, 58]. For this reason, Dropbox uses Amazon S3 for data blocks, but keeps small metadata in DRAM for fast access, backed by an SSD [162]. Apache Crail [163] and Blizzard [164] provide file system APIs on top of block stores, but both focus on parallel throughput of large data streams, rather than small IO.

To realize the performance benefits of NVM for all IO, we need to abandon fixed block sizes and instead persist and track IO at its original operation granularity. Hence, Assise leverages logging to persist writes at their original granularity in NVM. A similar model is realized in the RAMcloud [165] key-value store. RAMcloud maintains data in DRAM for performance, using SSDs for asynchronous persistence. However, the capacity limits of DRAM mean that many RAMcloud operations still involve the network, and because DRAM state cannot be recovered after a crash, it is vulnerable to cascading node failures. Even after single node failures, state must be restored from remote nodes and RAMcloud requires a full-bisection bandwidth network for fast recovery. Assise leverages local NVM for recovery and does not require full-bisection bandwidth or asynchronous backup storage.

Client-server file systems, like Ceph [49], use distributed hashing over nodes to provide scalable file service for cloud applications. However, network and system call latency harms file IO latency, as shown in Table 5.1. Typical network access bandwidth to NVM is similarly surpassed by the higher bandwidth of local NVM.

To combat network overheads, several file systems have been built [50, 51] or retrofitted [55, 59, 60] to use RDMA. Octopus [50] and Orion [51] are redesigns that use RDMA for high performance access to NVM. Still, neither leverages kernel-bypass for low-latency IO (Octopus uses FUSE, Orion runs in the kernel) and both pool storage remotely. Like Ceph, Octopus uses distributed hashing to

Concept	Explanation
LibFS	Per-process, user-level file system library
SharedFS	Per-socket system daemon; manages local leases
CC-NVM	Crash-consistent cache coherence with linearizability
Hot replica	Cache-hot replica for fast failover
Warm replica	Provides NVM for low-latency, remote, warm reads
Cluster manager	Fault-tolerant service for membership & leases

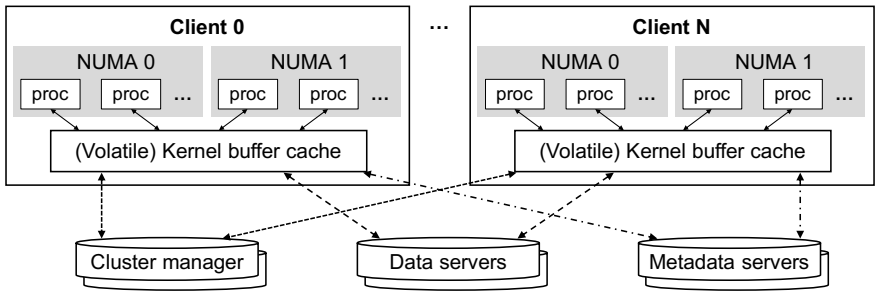
Table 5.2: Concepts used in Assise.

place files on nodes (Octopus does not replicate). Orion can store data locally via "internal clients," but uses a metadata server. Clover [166] is a key-value store that takes the opposite approach, locating metadata with applications, but storing data remotely. All systems perform remote operations in the common case to update data and/or metadata, increasing IO latency.

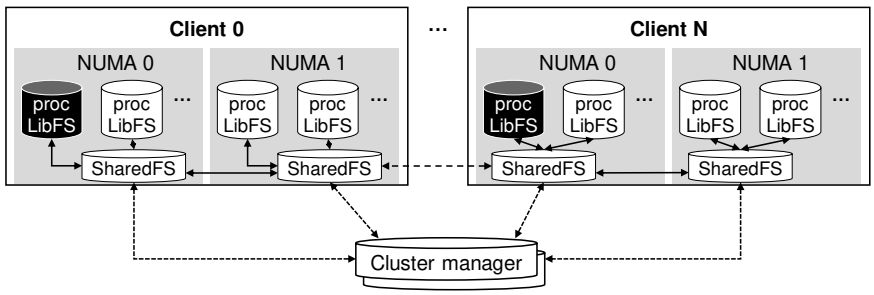
Network latency and limited bandwidth increase operation latency, reduce throughput, and limit scalability. For example, due to update contention at a central metadata server, Orion scales only to a small number of clients. Orion omits an evaluation of server fail-over and recovery (Assise's is in §5.4.4). Tachyon [167] aims to circumvent replication overhead by leveraging the concept of *lineage*, where lost output is recovered by re-executing application code that created the output. However, to do so, Tachyon requires applications to use a complex data lineage tracking API.

To maximize NVM utility, we need to design for a scenario where kernel and networking overheads are high compared to storage access. Hence, Assise eliminates kernel overhead for local IO operations and remote IO incurs a single operation to the nearest replica in the common case, without requiring dedicated metadata servers or a distributed hash to balance load. For scalability, we need to enforce data and metadata consistency locally, which CC-NVM tackles with the help of leases. Unlike Tachyon, Assise supports the classic POSIX file API and is fully compatible with existing applications.

Leases [153, 168] have long been integral to performance in distributed file systems, by allowing local operations to leased portions of the file system name space, with linearizability. Read-only leases are a common design pattern [55, 169, 170, 171], but some research systems have explored using both read and write leases in a similar manner to Assise. A prominent example is Berkeley xFS [54], which maintained a local block-level update log at each node, written as a software RAID 5/6 partitioned across other nodes. Assise differs from xFS by using an operational log, replicating rather than striping the log, and by doing update coalescing.



(a) Traditional distributed FSes with server-side storage (NFS, Ceph, …).



(b) Distributed file system IO architectures. Arrow = RPC/system call. Cylinder = persistence. Black = hot replica.

Figure 5.1: Distributed file system IO architectures. Arrow = RPC/system call. Cylinder = persistence. Black = hot replica.

5.1.2 Remote Storage versus Local NVM

Figure 5.1 contrasts the IO architecture of traditional client-server file systems and Assise. Each subfigure shows two dual-socket nodes executing a number of application processes sharing a distributed file system. Both designs use a replicated cluster manager for membership management and failure detection, but they diverge in all other respects.

Traditional distributed file systems first partition available cluster nodes into clients and servers. Clients cache file system state in a volatile kernel buffer cache that is shared by processors across sockets (NVM-NUMA) and accessed via expensive system calls (NVM-kernel). Persistent file system state is stored in NVM on remote servers. For persistence and consistency, clients thus have to coordinate updates with replicated storage and metadata servers via the network (NVM-RDMA) with higher latency than local NVM. The cluster manager is not involved in IO. Data is typically distributed at random over replicated storage

servers for simplicity and load balance [49]. The overhead of updating a large set of storage nodes atomically means that (crash) consistency is often provided only for metadata, which is centralized.

5.2 Assise Design

Assise avoids remote storage servers and instead uses CC-NVM to coordinate linearizable state among processes. Processes access cached file system state in local NVM directly via a library file system (LibFS), which may be replicated for fail-over (two LibFS hot replicas shown in black in Figure 5.1). CC-NVM coordinates LibFSes hierarchically via per-socket daemons (SharedFS) and the cluster manager. Table 5.2 explains several Assise-related concepts.

Crash consistency modes. Assise supports two crash consistency modes: optimistic or pessimistic [172]. Mount options specify the chosen crash consistency mode. When pessimistic, `fsync` forces immediate, synchronous replication and all writes prior to an `fsync` persist across failures. When optimistic, Assise commits all operations in order, but it is free to delay replication until the application forces it with a `dsync` call [172]. Optimistic mode provides lower latency persistence with higher throughput, but risks data loss after crashes that cannot recover locally (§5.2.4). In either mode, Assise guarantees a prefix crash-consistent file system [152]—all recoverable writes are in order and no parts of a prefix of the write history are missing.

We now describe cluster coordination and membership management in Assise (§5.2.1). We then detail the IO paths (§5.2.2) and show how CC-NVM interacts with them to provide linearizability and prefix crash consistency (§5.2.3). Finally, we describe recovery (§5.2.4) and warm replicas (§5.2.5). We close with a discussion of connected design questions (§5.2.6).

5.2.1 Cluster Coordination and Failure Detection

Like other distributed file systems, Assise leverages a replicated cluster manager for storing the cluster configuration and detecting node failures. Assise uses the ZooKeeper [173] distributed coordination service as its cluster manager.

Cluster coordination. Each SharedFS in Assise registers with the cluster manager. In our prototype, the system administrator decides which SharedFS replicates which parts of the cached file system namespace and the caching policy (hot or warm replica) for arbitrary subtrees; the cluster manager records this mapping. When a subtree is first accessed, LibFSes contact their local SharedFS,

which consults the cluster configuration and sets up an RDMA replication chain from LibFS through the subtree's hot replicas. For each chain, hot replicas preallocate a configurable amount of NVM for replication (sensitivity evaluated in §5.4.2). It is future work to implement a distributed replica discovery service (*e.g.*, using CC-NVM). LibFSes on any node are already able to cache any (meta-)data with linearizability.

Failure detection. The cluster manager sends heartbeat messages to each active SharedFS once every second. If no response is received after a timeout, the node is marked failed and all connected SharedFS are notified. When the node comes back online, it contacts the cluster manager and initiates recovery (§5.2.4).

5.2.2 IO Paths

Application IO interacts first with Assise caches. To keep tail latency low, Assise does not use a shared kernel buffer cache. Instead, LibFS caches file system state first in process-local memory. The LibFS cache uses both NVM and DRAM. NVM stores updates, while DRAM caches reads. LibFS implements the POSIX API at user-level. We now discuss cache operation upon IO, including replication, eviction, and access permissions. Figure 5.2 shows these mechanisms for two hot replicas and one warm replica, using SSDs for cold storage. Cache coherence is discussed in §5.2.3.

5.2.2.1 Write Path

Writes in Assise involve three mechanisms that operate on different time scales:

1. To allow for persistence with low latency, LibFS directly writes into a process-local cache in NVM (Ⓦ). To efficiently support writes of any granularity, the write cache is an *update log*, rather than a block cache.
2. To outlive node failures, the update log is chain-replicated, with kernel-bypass, by LibFS (Ⓢ1, Ⓢ2).
3. When update logs fill beyond a threshold, evictions are initiated (Ⓔ2), moving their contents to SharedFS. We describe replication and eviction next.

Replication and crash consistency. When pessimistic, `fsync` forces immediate, synchronous replication. The caller is blocked until all writes up to the `fsync` have been replicated. Thus, all writes prior to an `fsync` outlive node failures.

When optimistic, Assise is free to delay replication. This provides Assise with an opportunity to *coalesce* [156] temporary durable writes (*i.e.*, overwritten or deleted files), a workload pattern seen in application-level commit protocols [174].

Eliminating these writes allows Assise to conserve network bandwidth. In optimistic mode, Assise initiates replication on `dsync` or upon log eviction.

In both cases, the local update log contents are written to preallocated NVM on the first replica along the replication chain via RDMA (S1). The replica continues chain replication to the next replica (S2), and so on. The final replica in the chain sends an acknowledgment back along the chain to indicate that the chain completed successfully.

Cache eviction. When a LibFS update log fills, it replicates any unreplicated writes and initiates eviction. Eviction is done in least-recently-used (LRU) fashion through the SharedFS shared caches to cold storage (E2, E3). Hot replicas keep *hot* data in NVM, while moving *warm* and *cold* data to cold storage. Warm replicas (§5.2.5) keep hot and cold data in cold storage, while warm data resides in NVM to accelerate warm reads (§5.2.5). Cold storage may be remote (*e.g.*, via NVMe-over-Fabrics [175]). Each replica along the chain evicts in parallel and acknowledges when eviction is finished. This ensures that all replicas cache identical state for fast failover.

For log eviction (E2), issuing direct stores to NVM shared caches on another socket has overhead due to cross-socket hardware cache coherence, limiting throughput [176]. Since CC-NVM provides cache coherence, Assise can bypass hardware cache coherence by using DMA [177] when evicting to NVM-NUMA. This yields up to 30% improvement in cross-socket file system write throughput (§5.4.5).

5.2.2.2 Read Path

LRU cache eviction guarantees that the latest version of all data is always available in the fastest cache. Thus, upon a read, LibFS (1) checks the process-private write and read caches (via a *log hashtable* and *read cache*, shown in Figure 5.2) for the requested data (R1). If not found, LibFS (2) checks the node-local hot SharedFS cache (R2) (via an *extent tree* used to index the SharedFS cache [156]). If the data was found in either of these areas, it is read locally. If not found, LibFS (3) checks the warm replica's SharedFS cache (R3), if it exists, and, in parallel, checks cold storage (R4).

Read cache management. Recently read data is cached in process-local DRAM, except if it was read from local NVM, where DRAM caching does not provide benefit. LibFS prefetches up to 256KB from cold storage and up to 4KB from remote NVM. For remote NVM reads, LibFS first fetches the requested data and then prefetches the remainder. This minimizes small read latency while

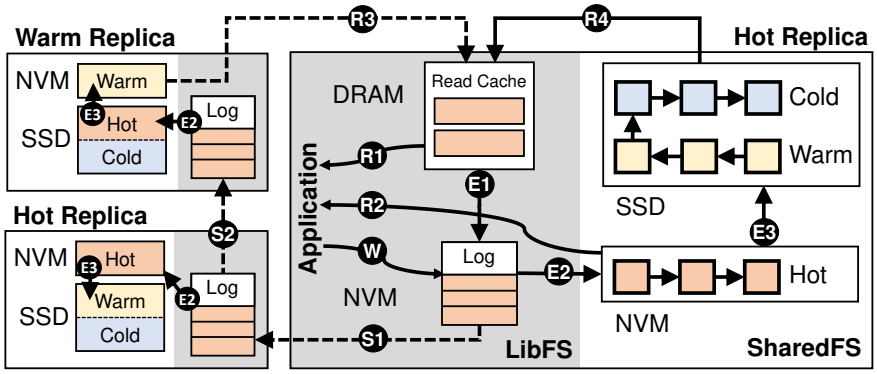


Figure 5.2: Assise IO paths. Dashed line = RDMA operation, solid line = local operation. Shaded areas are per process.

improving the performance of workloads with spatial locality. Data from remote NVM and cold storage is evicted from the read cache to the process-local update log (E1).

5.2.2.3 Permissions and Kernel Bypass

Assise assumes a single administrative domain with UNIX file and directory ownership and permissions. SharedFS enforces that LibFS may access only authorized data, by checking permissions and metadata integrity upon cache eviction and enforcing permissions on reads. To minimize latency of node-local SharedFS cache reads, Assise allows read-only mapping of authorized parts of the SharedFS cache into the LibFS address space. LibFS caches and mappings are invalidated when files or directories are closed and whenever the update log is evicted.

The metadata integrity of the file system is ensured by SharedFS. LibFS operations do not prevent one thread from corrupting another's data in the process-local update log, but SharedFS verifies that all metadata operations are valid before they become visible to other processes. This implies that processes can corrupt *their* own data in their private update log, even after it was written (memory protection keys can mitigate inter-thread data corruption [178]). However, only process-local writes go to the process-local update logs. Multi-process access to any file system object (including a subtree) is linearizable and access-controlled via leases. Processes cannot corrupt shared file system (meta-)data.

5.2.3 Crash Consistent Cache Coherence with CC-NVM

CC-NVM provides distributed cache coherence with linearizability when sharing file system state among processes; it provides prefix semantics upon a crash.

Prefix crash consistency. To provide prefix crash consistency, CC-NVM tracks write order via the update log in process-local NVM. Each POSIX call that updates state is recorded, in order, in the update log. When chain-replicating, CC-NVM leverages the write ordering guarantees of (R)DMA to write the log in order to replicas. In optimistic mode, CC-NVM wraps coalesced file system operations in a Strata transaction [156]. This ensures that file system updates are persisted and replicated atomically and that a prefix of the write history can be recovered (§5.2.4).

Sharing with linearizability. CC-NVM serializes concurrent access to shared state by multiple processes and recovers the same serialization after a crash via leases [153]. Leases provide a simple, fault-tolerant mechanism to delegate access. Leases function similarly to reader-writer locks, but can be revoked (to allow another process access) and expire after a timeout (after which they may be reacquired). In CC-NVM, leases are used to grant shared read or exclusive write access to a set of files and directories—multiple read leases to the same set may be concurrently valid, but write leases are exclusive. Reader/writer semantics efficiently support shared files and directories that are read-mostly and widely used, but also write-intensive files and directories that are not frequently shared. CC-NVM also supports a *subtree lease* that includes all files and directories at or below a particular directory. A subtree lease holder controls access to files and directories within that subtree. For example, a LibFS with an exclusive subtree lease on `/tmp/bwl-ssh/` can recursively create and modify files and directories within this subtree.

Leases must be acquired by LibFS from SharedFS via a system call before LibFS can cache the data covered by the lease. Assise does this upon first IO; leases are kept until they are revoked by SharedFS. This occurs when another LibFS wishes access to a leased file or when a LibFS instance crashes or the lease times out. Lease revocation latency is bounded by a grace period, within which the current lease holder can finish its ongoing IO before releasing contended leases. If LibFS fails to surrender the lease after the grace period, the lease is revoked by SharedFS and any subsequent IO on the leased file is rejected as invalid. SharedFS enforces that the lease holder's read and write caches are cleaned and evicted of the covered data before the lease is transferred. The time taken to do so is bounded by the holder's update log size. SharedFS logs and replicates each lease transfer

in NVM for crash consistency. A LibFS may overlap IO with SharedFS lease replication until `fsync`/`dsync`.

Hierarchical coherence. To localize coherence enforcement, leases are delegated hierarchically. The cluster manager is at the root of the delegation tree, with SharedFSes as children, and LibFSes as leaves (cf. Figure 5.1b). LibFSes request leases first from their local SharedFS. If the local SharedFS is not the lease holder, it consults the cluster manager. If there is no current lease holder, the cluster manager assigns the lease to the requesting SharedFS, which delegates it to the requesting LibFS and becomes its *lease manager*. If a lease manager already exists, SharedFS forwards the request to the manager and caches the lease manager's information (leased namespace and expiration time of lease). The cluster manager expires lease management from SharedFSes every 5 seconds. This allows CC-NVM to migrate lease management to the local SharedFS, while preventing leases from changing managers too quickly, facilitating scalability.

Hierarchical coherence minimizes network communication and thus lease delegation overhead. LibFSes on the same node or socket require only local SharedFS delegation in the common case. This structure maps well to the data sharding patterns of many distributed applications (§5.4.5).

5.2.4 Fail-over and Recovery

Assise caches file system state with persistence in local NVM, which it can use for fast recovery. Assise optimizes recovery performance for the most common crash types.

LibFS recovery. An application process crashing is the most common failure scenario. In this case, the local SharedFS simply evicts the dead LibFS update log, recovering all completed writes (even in optimistic mode) and then expires its leases. Log-based eviction is idempotent [156], ensuring consistency in the face of a system crash during eviction. The crashed process can be restarted on the local node and immediately re-use all file system state. The LibFS DRAM read-only cache has to be rebuilt, with minimal performance impact (§5.4.4).

SharedFS recovery. Another common failure mode is a reboot due to an OS crash. In this case, we can use NVM to dramatically accelerate OS reboot by storing a checkpoint of a freshly booted OS. After boot, Assise can initiate recovery for all previously running LibFS instances, by examining the SharedFS log stored in NVM.

Cache replica fail-over. To avoid waiting for node recovery after a power failure or hardware problem, we immediately fail-over to a hot replica. The replica's SharedFS takes over lease management from the failed node, using the replicated SharedFS log to re-grant leases to any application replicas. The new instances will see all IO that preceded the most recently completed `fsync`/`dsync`.

Writes to the file system can invalidate cached data of the failed node during its downtime. To track writes, the cluster manager maintains an epoch number, which it increments on node failure and recovery. All SharedFS instances are notified of epoch updates. All SharedFS instances share a per-epoch bitmap in a sparse file indicating what inodes have been written during each epoch. The bitmaps are deleted at the end of an epoch when all nodes have recovered.

Node recovery. When a node crashes, the cluster manager makes sure that all of the node's leases expire before the node can rejoin. When rejoining, Assise initiates SharedFS recovery. A recovering SharedFS contacts an online SharedFS to collect relevant epoch bitmaps. SharedFS then invalidates every block from every file that has been written since its crash. This simple protocol could be optimized, for instance, by tracking what blocks were written, or checksumming regions of the file to allow a recovering SharedFS to preserve more of its local data. But the table of files written during an epoch is small and quickly updated during file system operation, and our simple policy has been sufficient.

5.2.5 Warm Replicas

To fully exploit the memory hierarchy presented in Table 5.1, remote NVM can be used as a third-level cache, below local DRAM and local NVM. To do so, we introduce *warm replicas*. Like hot replicas, warm replicas receive all file system updates via chain-replication, but leverage a different update log eviction policy. Warm replicas track the LRU chain for a specified portion of "warm data" beyond the LibFS and SharedFS caches. Warm replicas do not impact the latency of replicated writes, but they reduce read latency for warm data by serving these reads from NVM, rather than cold storage.

LibFSes can read from warm replicas via RDMA with lower latency and higher bandwidth than cold storage (NVM-RDMA versus SSD in Table 5.1). Applications do not run on warm replicas in the common case. In the rare case of a failure cascade crashing all hot replicas, processes can fail-over to warm replicas, albeit with reduced short-term performance. After fail-over, warm replicas become hot replicas and hot data must be migrated back into local NVM.

5.2.6 Discussion

Assise may be deployed at scale. The use of local NVM together with hierarchical lease delegation aligns well with datacenter server, rack, and pod architecture [14]. We discuss factors of Assise's design that impact such a deployment. In particular, the memory overhead of per-process and per-replica update logs, the use of NVM and RDMA at scale, and security.

Update log scalability. Assise uses per-process and per-replica update logs for efficient chain-replication with kernel-bypass. These update logs are preallocated on process creation in our prototype. While update logs can support high performance at moderate size (§5.4.2), a deployment at scale might be concerned with the memory consumption of update logs. In this case, the per-process and per-replica update log size can be adapted dynamically to momentarily available NVM capacity and per-process IO demand. SharedFS can resize logs upon eviction. The most significant overhead for log resizing is memory registration for RDMA. It requires pinning the memory and mapping it in the RDMA NICs. This operation can be overlapped with the log eviction itself. To help reduce the need for frequent resizing, logs can be resized multiplicatively, similar to resizing approaches in prior work [179].

RDMA scalability. Assise uses RDMA reliable connections (RCs) for each process and replica. RCs require the NIC to create and maintain connection state. For larger clusters, maintaining a large number of connections can stress the NIC's limited memory and degrade performance. Several proposals have been made to reduce NIC cache thrashing [180, 181] and Mellanox introduced dynamically-connected (DC) transports [182], which allows connection-sharing and enables a high degree of scalability. Assise can leverage these approaches to scale the use of RDMA.

NVM wear-out. Assise uses local NVM extensively. This use can lead to the wear-out of NVM. To prevent frequent NVM replacement at scale, it is important to minimize writes to the NVM media. Assise's update logs minimize write amplification, but update log eviction in causes a 2× write amplification in the worst case. This write amplification can be partially eliminated via coalescing as seen in workloads, like Varmail (§5.4.3). To further reduce write amplification, update log pages may be remapped to the SharedFS shared cache, without introducing any additional writes [183]. We leave this as future work.

Security. In a large-scale public cloud scenario, data from each tenant is usually encrypted for security. For this purpose, both NVM and RDMA support encryption of data at rest and in-flight. Intel's Optane DC PMMs support transparent hardware encryption of data stored in NVM and modern RDMA NICs [184] support transparent encryption of RDMA operations.

5.3 Implementation

Assise uses *libpmem* [185] for persisting data on NVM and *libibverbs* for RDMA operations in userspace. Assise intercepts POSIX file system calls and invokes the corresponding LibFS implementation of these functions in userspace [186]. The Assise implementation consists of 28,982 lines of C code (LoC), with LibFS and SharedFS using 16,515 and 6,563 LoC, respectively. The remaining 5,904 LoC contain utility code, such as hash tables and linked lists. SharedFS communicates with LibFSes via shared memory [187]. Assise uses Strata code (LoC not counted) for cold storage in SSD and HDD.

Assise uses Intel Optane DC PMM in App-Direct mode. App-Direct exposes NVM as a range of physical memory. It is the most efficient way to access NVM, but it requires OS support. OS-transparent modes have weaker persistence or performance properties [188]. For example, memory mode integrates NVM as *volatile* memory, using DRAM as a hardware-managed level 4 cache. Sector mode exposes NVM as a disk, with attendant IO amplification and disk driver overheads.

5.3.1 Strata as a Building Block

Assise builds upon Strata's local file system functionality and augments it with the CC-NVM cache coherence layer and RDMA to create a replicated and highly efficient distributed file system with prefix crash conistency. Assise inherits several components from Strata, including its use of extent trees to index storage managed by SharedFS (in turn based on Ext4 [189]), the LibFS update log, and log coalescing. We enhance Strata's extent trees to manage directories and Strata's leases to support delegation.

5.3.2 Efficient Network IO with RDMA

Assise makes efficient use of RDMA. For lossless, in-order data transfer among nodes, Assise uses RDMA reliable connections (RCs). RCs have low header overhead, improving throughput for small IO [35, 190]. RCs also provide access to one-sided verbs that bypass CPUs on the receiver side, reducing message transfer times [34, 37] and memory copies [191].

Log replication. Logs are naturally suited for one-sided RDMA operations. Replication typically requires only one RDMA write, reducing header and DMA overheads [190]. Assise uses RDMA *write-with-immediate* for log replication. This operation performs a write and also notifies the remote replica to forward the data to the next replica in the chain. The only exceptions are when the remote log wraps around or when the local log is fragmented (due to coalescing), such that it exceeds the NIC's limit for scatter-gather DMA.

Persistent RDMA writes. The RDMA specification does not define the persistence properties of remote NVM writes via RDMA. In practice, the remote CPU is required to flush any RDMA writes from its cache to NVM. Assise flushes all writes via the CLWB and SFENCE instructions on each replica, before acknowledging successful replication. In the future, it is likely that enhancements to PCIe will allow RDMA NICs to bypass the processor cache and write directly to NVM to provide persistence without CPU support [192].

Remote NVM reads. Assise reads remote data via RPC. To keep the request sizes small, Assise identifies files using their inode numbers instead of their path. As an optimization, DRAM read cache locations are pre-registered with the NIC. This allows the remote node to reply to a read RPC by RDMA writing the data directly to the requester's cache, obviating the need for an additional data copy.

5.4 Evaluation

We evaluate Assise's common-case as well as its recovery performance, and break down the performance benefits attained by each system component. We compare Assise to three state-of-the-art distributed file systems that support NVM and RDMA. Our experiments rely on several microbenchmarks and Filebench [193] profiles, in addition to several real applications, such as LevelDB, Postfix, and MinuteSort. Our evaluation answers the following questions:

- **IO latency and throughput breakdown (§5.4.2).** What is the hardware IO performance of a storage hierarchy with local NVM (Table 5.1)? How close to this performance do the file systems operate under various IO patterns? What are the sources of overhead?
- **Cloud application performance (§5.4.3).** What is the performance of cloud applications with various consistency, latency, throughput, and scalability requirements? What is the overhead of Assise's POSIX API implementation versus hand-tuned, direct use of local NVM? By how much can a warm replica improve read latency? By how much can optimistic crash consistency improve write throughput for real applications?

- **Availability (§5.4.4).** How quickly can applications recover from various failure scenarios?
- **Scalability (§5.4.5).** How well does Assise perform when multiple processes share the file system? By how much can Assise's hierarchical coherence improve multi-process, multi-socket, and multi-node scalability?

Testbed. Our experimental testbed consists of 5× dual-socket Intel Cascade Lake-SP servers running at 2.2GHz, with a total of 48 cores (96 hyperthreads), 384GB DDR4-2666 DRAM, 6TB Intel Optane DC PMM, 375GB Intel Optane DC P4800X series NVMe-SSD, and a 40GbE ConnectX-3 Mellanox InfiniBand NIC, connected via an InfiniBand switch. Exploiting all 6 memory channels per processor, there are 6 DIMMs of DRAM and NVM per socket. NVM is used in App-Direct mode (§5.3). All nodes run Fedora 27 with Linux kernel version 4.18.19.

Hardware performance. We first measure the achievable IO latency and throughput for each memory layer in our testbed server. We do this by using sequential IO and as many cores of a single socket as necessary. We measure DRAM and NVM (App-Direct) latency and throughput using Intel's memory latency checker [194]. NVM-RDMA performance is measured using RDMA read and write-with-immediate (to flush remote processor caches) operations to remote NVM. SSD performance is measured using `/dev/nvme` device files. The IO sizes that yielded maximum performance are 64B for DRAM, 256B for NVM, and 4KB for SSD. Table 5.1 shows these results. The measured IO performance for DRAM, NVM, and SSD matches the hardware specifications of the corresponding devices and is confirmed by others [188]. NVM-RDMA throughput matches the line rate of the NIC. NVM-RDMA write latency has to invoke the remote CPU (to flush caches) and is thus larger than read latency. We now investigate how close to these limits each file system can operate.

State-of-the-art. Table 5.3 shows performance-relevant features of the state-of-the-art and Assise. We can see that no open-source distributed file system provides all of Assise's features. Hence, a direct performance comparison is difficult. We perform comparisons against the Linux kernel-provided NFS version 4 [55] and Ceph version 14.2.1 [49] with BlueStore [157], both retrofitted for RDMA, as well as Octopus [50]. We cannot directly compare with Orion [51] as it is not publicly available, but we emulate its behavior where possible. Only Ceph provides availability via replicated object storage daemons (OSDs), delegating metadata management to a (potentially sharded) metadata server (MDS). Octopus and NFS do not support replication for availability and thus gain an unfair

Feature	Assise	Ceph	NFS	Octopus	Orion
Cache recovery	✓				
Local consistency	✓				
Kernel-bypass	✓				
Linearizability	✓				✓
Data crash consistency	✓				✓
Byte-oriented	✓			✓	✓
Replication	✓	✓			✓
RDMA	✓	✓	✓	✓	✓

Table 5.3: Features of the evaluated distributed file systems.

performance advantage over Assise. However, Assise beats them even while replicating for availability, showing that both features can be had when leveraging local NVM.

Other file systems do not support persistent caches and their consistency semantics are often weaker than Assise's. Assise provides data crash consistency, while both Ceph/BlueStore and Octopus provide only metadata crash consistency [195]. For NFS, crash consistency is determined by the underlying file system. We use EXT4-DAX [196], which also provides only metadata crash consistency. When sharing data, NFS provides *close-to-open consistency* [55], while Octopus and Ceph provide "stronger consistency than NFS" [197], and Assise provides linearizability, which is stronger than Octopus' and Ceph's guarantees. In all performance comparisons, Assise provides stronger consistency than the alternatives. Ceph is the closest comparison point.

File system compliance tests. We tested Assise using xfstests [198] and CrashMonkey [199]. Assise passed all 75 generic xfstests that are recommended for NFS [200]. NFSv4.2 and Ceph v14.2.1 pass only 71 and 69 of these tests, respectively. In part, this is due to their weaker consistency model. Assise also successfully passes CrashMonkey tests, runs all existing Filebench profiles, passes all unit tests for the LevelDB key-value store, and passes MinuteSort validation.

5.4.1 Experimental Configuration

Machines. Each experiment specifies the number (≥ 2) of testbed machines used. By default, machines are used as hot replicas in Assise, as a pool of storage nodes in Octopus, and as OSD and MDS replicas in Ceph. NFS uses only one machine as server, the rest as clients. We place applications on hot replicas for

Assise, on OSD replicas for Ceph, on storage nodes for Octopus, and on clients for NFS. Assise's and Ceph's cluster managers run on 2 additional testbed machines (NFS and Octopus do not have cluster managers). The colocated deployment of applications and OSDs for Ceph is due to the small size of our cluster. It gives Ceph a potential performance advantage over an all-remote OSD deployment.

Network. We use RDMA for the NFS client-server connection. Ceph provides its client-side file system via the Ceph kernel driver and uses IP over InfiniBand, which was the fastest configuration (we also tried FUSE and Accelio [59]). Assise and Octopus use RDMA with kernel-bypass.

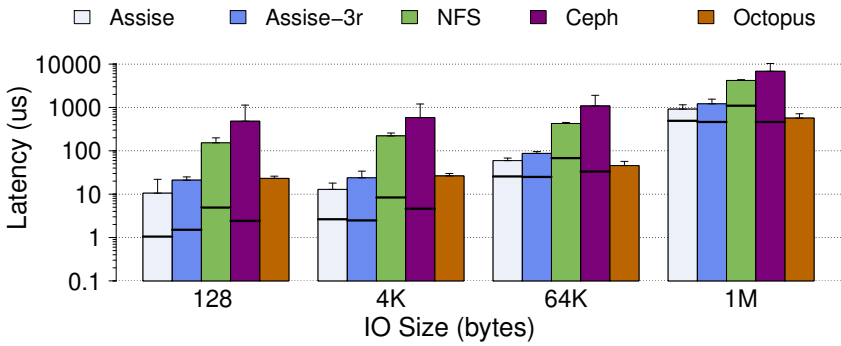
Storage and caches. For maximum efficiency, all file systems use NVM in App-Direct mode to provide persistence (cylinders in Figure 5.1) and DRAM when persistence is not needed (e.g., kernel buffer cache). We investigate Ceph and NFS performance using NVM in memory mode for volatile caches and find it to degrade throughput by up to 25% versus DRAM. For efficient access to NVM, Ceph OSDs use BlueStore and NFS servers use EXT4-DAX. Octopus uses FUSE to provide its file system interface to applications in *direct IO* mode to NVM, bypassing the kernel buffer cache [201].

To evaluate a breadth of cache behaviors with limited application data set sizes, we limit the fastest cache size for all file systems to 3GB. For Ceph and NFS, we limit the kernel buffer cache to 3GB. For Assise, we partition the LibFS cache into a 1GB NVM update log and a 2GB DRAM read cache (the SharedFS second-level cache may use all NVM available), and we run Assise in pessimistic mode.

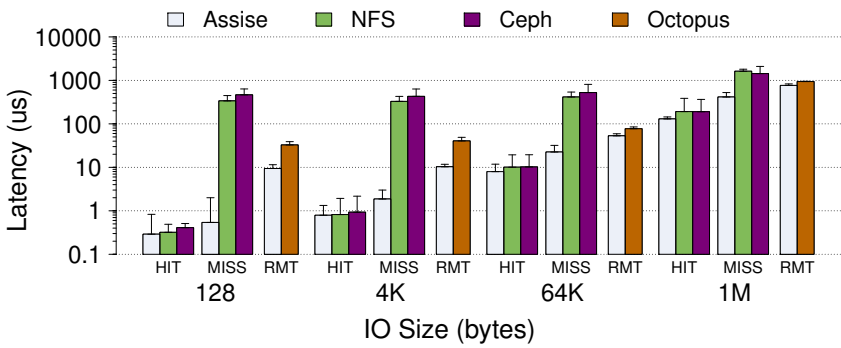
5.4.2 Microbenchmarks

Average and tail write latency. We compare unladen synchronous write latencies with 2 machines (except Assise-3r which uses 3 machines). Synchronous writes involve `write` calls (fixed-width font identifies POSIX calls) that operate locally (except for Octopus where `write` may be remote), and `fsync` calls that involve remote nodes for replication (Assise, Ceph) and/or persistence (Ceph, NFS). Each experiment appends 1GB of data into a single file, and we report per-operation latency. In this case, the file size is smaller than each file system's cache size, so no evictions occur—with gigabytes of cache capacity, this is common for latency-sensitive write bursts.

Figure 5.3a shows the average and 99th percentile sequential write latencies over various common IO sizes (random write latencies are similar for all file systems). We break writes down into `write` (solid line) and `fsync` call latencies (bar). Octopus' `fsync` is a no-op. Assise's local write latencies match that of



(a) Sequential write. `write` latency is solid line, `fsync` is bar height.



(b) Read latencies for cache hits, misses, and remote (RMT) misses.

Figure 5.3: Avg. and 99%ile (error bar) IO latencies. Log scale.

Strata [156]. Assise's average write latency for 128B two-node replicated writes is only 8% higher than the aggregate latencies of the required local and NVM-RDMA writes (cf. Table 5.1). Three replicas (Assise-3r) increase Assise's overhead to 2.2× due to chain-replication with sequential RPCs. The 99th percentile replicated write latency is up to 2.1× higher than the average for 2 replicas. This is due to Optane PMM write tail-latencies [188]. The tail difference diminishes to 19% for 3 replicas due to the higher average.

Ceph and NFS use the kernel buffer cache and interact at 4KB block granularity with servers. For small writes, the incurred network IO amplification during `fsync` is the main reason for up to an order of magnitude higher aggregate write latency than Assise. In this case, their `write` latency is up to 3.2× higher than

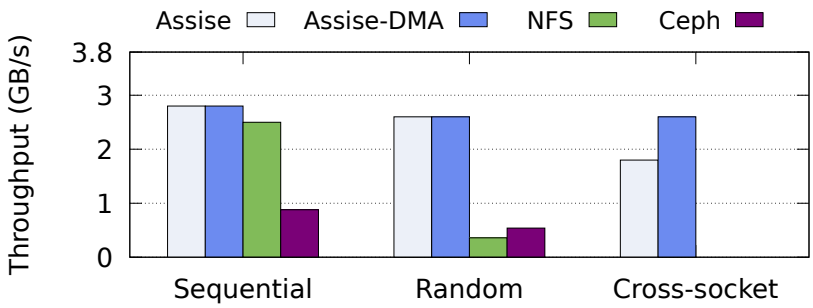
Assise due to kernel crossings and copy overheads. For large writes ($\geq$ 64KB), network IO amplification diminishes but the memory copy required to maintain the kernel buffer cache becomes a major overhead. The latency of large `writes` is higher than Assise's replicated write latency (and up to 2.7$\times$ higher than Assise's non-replicated `write` latency), while aggregate write latency is up to 7.2$\times$ higher than Assise. Ceph has higher `fsync` latency than NFS due to replication.

Octopus eliminates the kernel buffer cache and block orientation, which improves its performance drastically versus NFS and Ceph. However, Octopus still treats all NVM as remote and uses FUSE for file IO. Octopus exhibits up to 2.1$\times$ higher latency than Assise for small (< 64KB) writes. This overhead stems from FUSE (around 10μs [202]) and writing to remote NVM via the network. Large writes ($\geq$ 64KB) amortize Octopus' write overheads. Assise has up to 1.7$\times$ higher write latency due to replication. Octopus does not replicate.

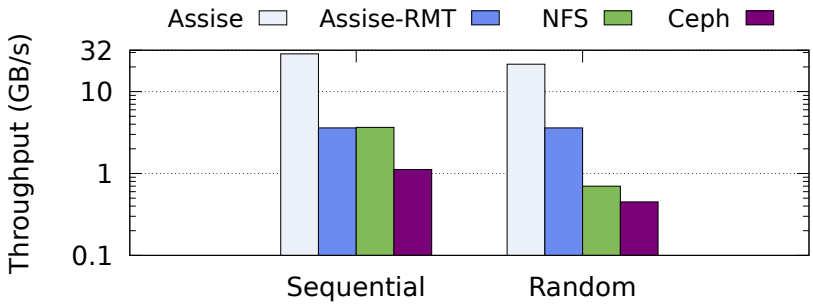
Average and tail read latency. We compare unladen read latencies across different cache configurations. To do this, we read a 1GB file using various IO sizes, once with a warm cache (to report cache hits) and once with a cold cache (to report misses). The results are shown in Figure 5.3b. Assise has a second-layer cache in SharedFS before going remote, and we report three cases for Assise. Reads in Octopus are always remote.

We first compare cache-hit latencies (HIT), where Assise is up to 40% faster than NFS and 50% faster than Ceph. Assise serves data from the LibFS read cache, while NFS and Ceph use the kernel buffer cache. If Assise misses in the LibFS cache, data may be served from the local SharedFS (MISS). Assise-MISS incurs up to 3.2$\times$ higher latency than Assise-HIT due to reading the extent tree index, especially for larger IO sizes that read a greater number of extents. If Assise misses in both caches, it has to read from a remote replica (RMT). Assise-RMT incurs the latency of an RPC using RDMA. When NFS and Ceph miss in the cache, their clients have to fetch from remote servers, which incurs up to orders of magnitude higher average and tail latencies than Assise-RMT and Assise-MISS. Ceph performs worse than NFS due to a more complex OSD read path.

The elimination of a cache hurts Octopus' read performance, because it has to fetch metadata and data (serially) from remote NVM (RMT). Octopus' read latency is up to two orders of magnitude higher than the other file systems hitting in the cache, and up to an order of magnitude lower than NFS and Ceph missing in the cache. Octopus does not handle small ($\leq$ 4KB) reads well due to FUSE overheads, with up to 3.54$\times$ Assise-RMT read latency. This overhead is amortized for larger reads ($\geq$ 64KB), where Octopus incurs up to 1.46$\times$ the read latency of Assise-RMT. By configuring FUSE to use the kernel buffer cache for Octopus, we reduce Octopus' read hit latency to 1.8$\times$ that of Assise-HIT, with the remaining overhead



(a) Write. 3.8GB/s is NVM-RDMA bandwidth.



(b) Read. 32GB/s is NVM read bandwidth.

Figure 5.4: Average throughput with 24 threads at 4KB IO size.

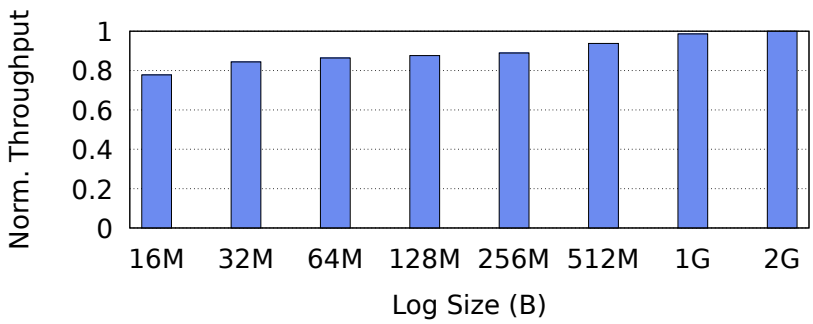


Figure 5.5: Worst-case throughput versus update log size, normalized to 2GB.

due to FUSE. However, using the kernel buffer cache inflates write latencies for Octopus by up to an order of magnitude due to additional buffer cache memory copies.

Peak throughput. Figure 5.4 shows average throughput of sequential and random IO to a 120GB dataset (on the local socket) with 4KB IO size from 24 threads (all cores of one socket). To evaluate a standard replication factor of 3, we use 3 machines for Assise and Ceph. The dataset is sharded over 24 files, and 5GB of data is written per thread. For random writes, a random offset is generated for every IO. `write` calls are not followed by `fsync` and the total amount of accessed data is larger than the cache size, causing cache eviction on write. The cache is initially cold so reads miss in the cache. For Assise, we show cache miss performance from a local and remote SharedFS. Octopus crashes during this experiment and is not shown.

For sequential writes, Assise and NFS achieve 74% and 66% of the NVM-RDMA bandwidth (cf. Table 5.1), respectively, due to protocol overhead for NFS and log header overhead for Assise. Chain-replication in Assise affects throughput only marginally. Ceph replicates in parallel to 2 remote replicas, consuming 3× the network bandwidth. This reduces its throughput to 31% of Assise and 35% of NFS. Assise achieves similar performance for sequential and random writes, as Assise's writes are log-structured. NFS and Ceph perform poorly for random writes due to cache block mis-prefetching incurring additional reads from remote servers, causing Assise to achieve 4.8× Ceph's throughput. NFS throughput is at only 67% that of Ceph, which is due to kernel locking overhead.

To quantify the benefit of bypassing hardware cache coherence for cross-socket writes with DMA, we repeat the benchmark, placing all files on the remote socket. We can see that Assise-DMA attains 44% higher cross-socket throughput than non-temporal processor writes (Assise). Sequential and random writes provide comparable performance. NVM-NUMA writes occur during eviction from the LibFS update log (local socket) to the NVM shared cache (remote socket). When writing to the local socket, Assise-DMA attains identical throughput to Assise, regardless of pattern.

For local sequential and random reads from the local SharedFS cache, Assise achieves 90% and 68%, respectively, of local, sequential NVM bandwidth. The 10% difference for sequential reads to local NVM bandwidth is due to metadata lookups, while random reads additionally suffer PMM buffer misses [188]. Assise remote reads (Assise-RMT) attain full NVM-RDMA bandwidth (3.8GB/s), regardless of access pattern. NFS and Ceph are limited by NVM-RDMA bandwidth for all reads and again have worse random read performance due to prefetching.

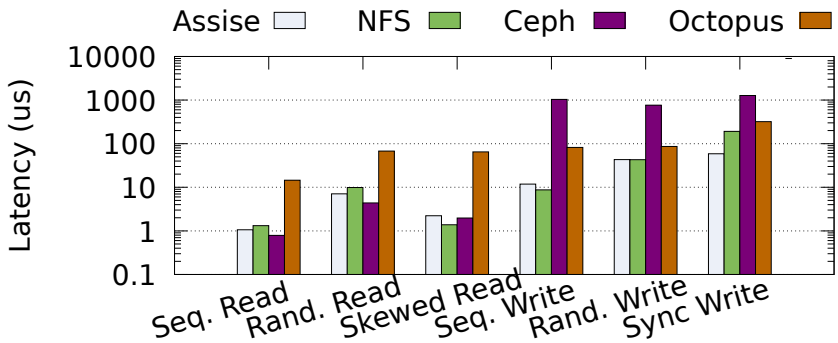


Figure 5.6: Average LevelDB benchmark latencies. Log scale.

Log size sensitivity. To evaluate the impact of log size on write throughput, we conduct a sensitivity analysis. We run a single-process microbenchmark that writes a 1GB file sequentially at 4KB IO granularity. This experiment models a worst case scenario. In the absence of sharing, processes can quickly fill up their allocated log space. Figure 5.5 shows the normalized write throughput at different log sizes. Throughput increases with log size, but the performance impact is small. Throughput increases by only 22% when using a 2GB log size versus a 16MB log size, a 128× increase in log size. For workloads that share data, we expect this gap to be smaller, as logs are evicted upon lease handoff. With 6TB of NVM per machine, Assise can scale to thousands of processes even with 2GB update logs. At 16MB, 100,000s of processes can be supported.

5.4.3 Application Benchmarks

We evaluate the performance of a number of common cloud applications, such as the LevelDB key-value store [203], the Fileserver and Varmail profiles of the Filebench [193] benchmarking suite, emulating file and mail servers, and the MinuteSort benchmark. We use 3 machines for LevelDB and Filebench and 5 machines for MinuteSort.

LevelDB. We run a number of single-threaded LevelDB latency benchmarks using LevelDB's `db_bench`, including sequential and random IO, skewed random reads with 1% of highly accessed keys, and sequential synchronous writes (`fsync` after each write). All benchmarks use a key size of 16B and a value size of 1KB with a working set of 1M KV pairs. Figure 5.6 presents the average measured operation latency, as reported by the benchmark.

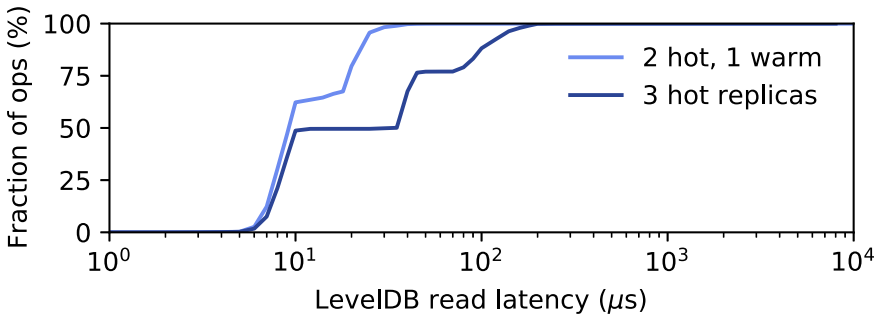


Figure 5.7: LevelDB random read latencies with warm replica.

Assise, Ceph, and NFS perform similarly for reads, where caching allows them to operate close to hardware speeds. For non-synchronous writes, NFS is up to 26% faster than Assise, as these go to its client kernel buffer cache in large batches (LevelDB has its own write buffer), while Assise is 69% faster than NFS for synchronous writes that cannot be buffered. Random IO and synchronous writes incur increasing LevelDB indexing overhead for all systems. Ceph performs worse than NFS for writes because it replicates (as does Assise) and Assise performs 22× better. Octopus bypasses the cache and thus performs worst for reads and better only than Ceph for writes, as it does not replicate.

Warm replica read latency. Warm replicas reduce read latency for warm data by allowing these reads to be served from remote NVM, rather than cold storage. For this benchmark, we configure Assise to limit the aggregate (LibFS and SharedFS) cache to 2GB and use the local SSD for cold storage. We then run the LevelDB random read experiment with a 3GB dataset. We repeat the experiment with two setups: (1) with 3 hot replicas and (2) with 2 hot and 1 warm replica. Figure 5.7 shows a CDF of read latencies. The benchmark accesses data uniformly at random, causing 33% of the reads to be warm. Consequently, at the 50th percentile, read latencies are similar for both configurations (served from cache). At the 66th percentile, reads in the first setup are served from SSD and have 2.2× higher latency than warm replica reads in the second setup. At the 90th percentile, the latency gap extends to 6×.

Filebench. Varmail and Fileserver operate on a working set of 10,000 files of 16KB and 128KB average size, respectively. Files grow via 16KB appends in both benchmarks (mail delivery in Varmail). Varmail reads entire files (mailbox reads) and Fileserver copies files. Varmail and Fileserver have write to read

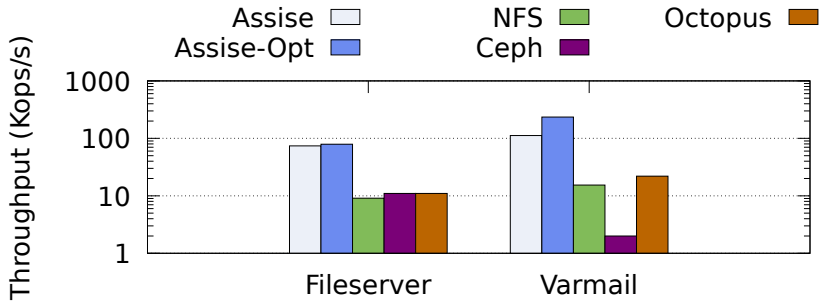


Figure 5.8: Avg. Varmail and Fileserver throughput. Log scale.

System	Processes	Partition [s]	Sort [s]	**Total [s]**	**GB/s**
Assise	160	20.3	43.0	**63.3**	**5.1**
	320	52.1	43.0	**95.1**	**6.7**
NFS	160	60.9	79.3	**140.2**	**2.3**
	320	104.1	84.2	**188.3**	**3.4**
DAX	320	–	44.1	–	–

Table 5.4: Average Tencent Sort duration breakdown.

ratios of 1:1 and 2:1, respectively. Varmail leverages a write-ahead log with strict persistence semantics (`fsync` after log and mailbox writes), while Filebench consistency is relaxed (no `fsync`). Figure 5.8 shows average measured throughput of both benchmarks. Assise outperforms Octopus (the best alternative) by 6.7× for Fileserver and 5.1× for Varmail. Ceph performs worse than NFS for Varmail due to stricter persistence requiring it to replicate frequently and due to MDS contention, as Varmail is metadata intensive.

Optimistic crash consistency. We repeat this benchmark for Assise in optimistic mode (Assise-Opt) and change Varmail to use synchronous writes for the mailbox, but non-synchronous writes for the log. Prefix semantics allow Assise to buffer and coalesce the temporary log write without losing consistency. Assise-Opt achieves 2.1× higher throughput than Assise. Fileserver has few redundant writes and Assise-Opt is only 7% faster.

MinuteSort. We implement and evaluate Tencent Sort [204], the current winner of the MinuteSort external sorting competition [205]. Tencent Sort sorts a partitioned input dataset, stored on a number of cluster nodes, to a partitioned

output dataset on the same nodes. It conducts a distributed sort consisting of 1) a range partition and 2) a mergesort (cf. MapReduce [206]). Step 1 presorts unsorted input files into ranges, stored in partitioned temporary files on destination machines. Step 2 reads these files, sorts their contents, and writes the output partitions. Each step uses one process per partition; the parallelism equals the number of partitions. A distributed file system stores the input, output, and temporary files, implicitly taking care of all network operations.

We benchmark the MinuteSort Indy category, which requires sorting a synthetic dataset of 100B records with 10B keys, distributed uniformly at random. Creating a 2GB input partition per process, we run 160 or 320 processes in parallel, uniformly distributed over 4 machines. MinuteSort does not require replication, so we turn it off. It calls `fsync` only once for each output partition, after the partition is written. We compare a version running a single Assise file system with one leveraging per-machine NFS mounts. For Assise, we configure the temporary and output directories to be colocated with the mergesort processes. We do the same for NFS, by exporting corresponding directories from each mergesort node. We conduct three runs of each configuration and report the average. We use the official competition tools [205] to generate and verify the input and output datasets. We use equal dataset sizes to compare Assise and NFS, rather than equal time. Table 5.4 shows that Assise sorts up to 2.2× faster than NFS. Running twice the number of processes only marginally improves performance, as Assise is bottlenecked by network bandwidth.

To show that Assise's POSIX implementation does not reduce performance, we modify the sort step to map all files into memory using EXT4-DAX and use processor loads and non-temporal stores to sort directly in NVM, rather than using file IO. We can see that the sort phase is 3% slower with DAX. libc buffers IO in DRAM to write 4KB at a time to NVM, performing better than direct, interleaved appends of 100B records.

5.4.4 Availability

Ceph and Assise are fault tolerant. We evaluate how quickly these file systems return an application back to full performance after the fail-over and recovery situations of §5.2.4. To do so, we run LevelDB on the same dataset (§5.4.3) with a 1:1 read-write ratio and measure operation latency before, during, and after fail-over and recovery. We report average results over 5 benchmark runs.

Process fail-over. For this benchmark, we simply kill LevelDB. In this case, the failure is immediately detected by the local OS and LevelDB is restarted. Ceph can reuse the shared kernel buffer cache in DRAM, resulting in LevelDB restoring

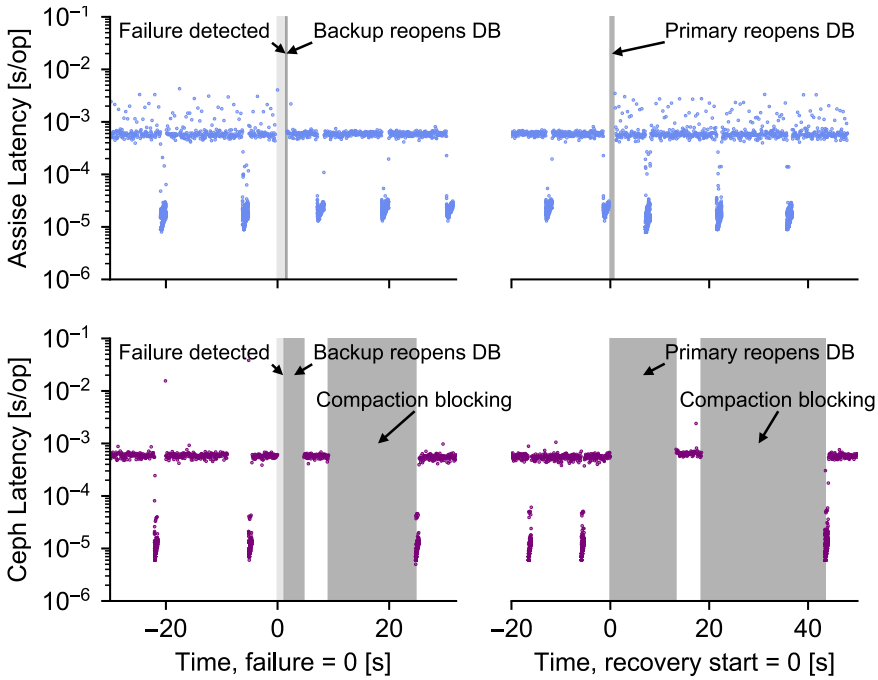


Figure 5.9: LevelDB operation latency time series during fail-over and recovery. Log scale.

its database after 1.63s and returning to full performance after an additional 2.15s, for an aggregate 3.78s fail-over duration. With Assise, the DB is restored in 0.71s, including recovery of the log of the failed process and reacquisition of all leases. Full-performance operations occur after an additional 0.16s, for an aggregate 0.87s fail-over time. Assise recovers this case 4.34× faster than Ceph, showing that process-local caches do not impede fast recovery.

OS fail-over. NVM's performance allows for instant local recovery of an OS failure, rather than requiring a backup replica. To demonstrate, we run the primary in a virtual machine (VM). We kill the primary VM, then immediately start a new VM from a snapshot stored in NVM. The snapshot starts in 1.66s. We restart SharedFS within the new VM, which recovers the file system within 0.23s. Finally, as in the process fail-over experiment, LevelDB is restarted and resumes database operations after another 0.68s. The aggregate fail-over time is 2.57s, 40× faster than Ceph's fail-over to a backup replica (evaluated next).

Fail-over to hot backup. All following experiments use 2 machines (primary and backup). The LevelDB client processes poll the file system's cluster manager for membership state, using a standard primary-backup ZooKeeper design pattern for node fail-over [207]. LevelDB initially runs on the primary, where we inject failures. Failures are detected by LevelDB clients using a 1s heartbeat timeout via the cluster manager. Once a node failure is detected, LevelDB immediately restarts on the backup.

A time series of measured LevelDB operation latencies during one experiment run is shown in Figure 5.9. Pre-failure, we see bursts of low latency in between stretches of higher latency. This is LevelDB's steady-state. Bursts show LevelDB writes to its own DRAM log. These are periodically merged with files when the DRAM log is full, causing writes that are higher latency (and sometimes blocking with Ceph), as the writes wait on the log to become available.

We inject a primary failure by killing the primary's file system daemon (SharedFS for Assise and OSD for Ceph) and LevelDB. During primary failure, no operations are executed. It takes 1s to detect the failure and restart LevelDB on the backup (light shaded box). Due to unclean shutdown, LevelDB first checks its dataset for integrity before executing further operations (dark shaded box). For failover, Assise need only evict the per-process log (up to 1GB) on the backup hot replica, making fail-over near-instantaneous. LevelDB returns to full performance in both latency and throughput 230ms after failure detection. Ceph takes 3.7s after failure detection to return to full performance. However, LevelDB stalls soon thereafter upon compaction (further dark shaded box), which involves access to further files, resulting in an additional 15.6s delay, before reaching steady-state. Ceph's long aggregate fail-over time of 23.7s is due to Ceph losing its DRAM cache, which it rebuilds during LevelDB restart. Assise reaches full performance after failure detection 103× faster than Ceph. LevelDB performs better on the backup, as neither file system has to replicate.

Primary recovery. After 30s, we restart the file system daemons on the primary, emulating the time for a machine reboot from NVM. During this time, many file system operations occur on the backup that need to be replayed on the primary. As soon as the primary is back online, we cleanly close the database on the backup and restart on the primary. Both Assise and Ceph allow applications to operate during primary recovery, but performance is affected. Assise detects outdated files via epochs and reads their contents from the remote hot replica upon access. Once read, the local copy is updated, causing future reads to be local. LevelDB returns to full performance 938ms after restarting it on the recovering primary. Ceph also rebuilds the local OSD, but eagerly. Ceph takes 13.2s before LevelDB serves its first operation due to contention with OSD recovery and suffers

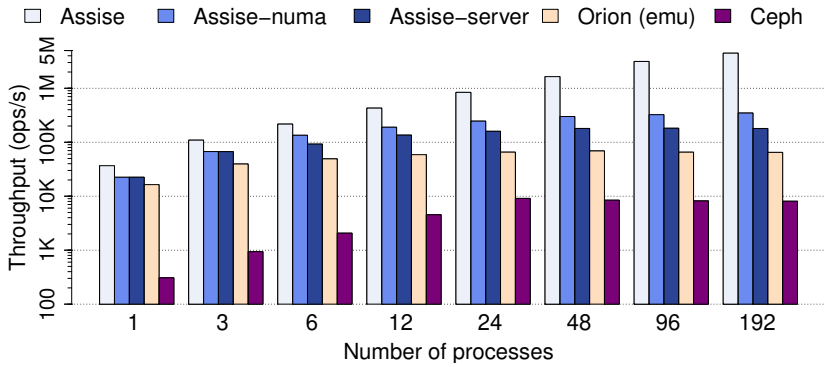


Figure 5.10: Scalability of atomic 4KB file operations. Log scale.

another delay of 24.9s on first compaction, reaching full performance 43.4s after recovery start. Assise recovers to full performance 46× faster than Ceph.

Fail-over to cold backup. We measure cascaded LevelDB fail-over time to an Assise replica with a cold cache. LevelDB serves its first request on the cold backup 303ms after failure detection, but with higher latency due to SSD reads. LevelDB returns to full performance after another 2.5s. At this point, the entire dataset has migrated back to cache.

5.4.5 Scalability

We evaluate Assise's scalability via 1) sharded file operations under increasing load and increasingly localized lease management, and 2) parallel email delivery in Postfix [208].

5.4.5.1 Sharded File Operations

Processes in parallel create, write, and rename 4KB files with random data in private directories. This benchmark uses 3 machines (6 sockets) and can scale throughput linearly with the number of processes. To eliminate network bottlenecks to scalability, we turn replication off.[1] Figure 5.10 presents average throughput over 5 runs of an increasing number of processes, each operating on 480K files, balanced over processor sockets. Ceph uses 3 sharded MDSes (1 per machine). However, MDS sharding has negligible impact on Ceph's performance.

[1] Due to the small size of our cluster, primary nodes would replicate to each other and scalability would be limited by per-node link bandwidth. A larger cluster would replicate to independent machines for each primary.

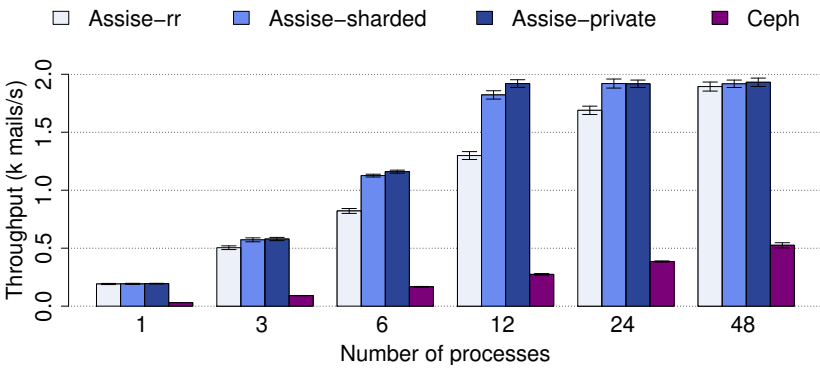


Figure 5.11: Postfix mail delivery throughput scalability.

Ceph's remote MDSes have high overhead for atomic operations, as each client has to communicate with remote MDSes. This design prevents scalability beyond 8Kops/s. We emulate Orion by restricting CC-NVM to use a single SharedFS lease manager. In this case, data is stored on local NVM, but atomic operations still use a remote lease manager. Orion has RDMA mechanisms that simplify communicating with its MDS, but these mechanisms cannot be used for operations that affect multiple inodes (e.g., renames). Orion and Assise both use RDMA RPCs. While Orion operates in the kernel, our emulation uses user-level RDMA, which is light-weight, and Orion (emu) outperforms Ceph by 8×.

To break down the benefit of local lease management in Assise, we progressively shard it, first by server (Assise-server), then by socket (Assise-numa), and finally by process (Assise). Assise-server outperforms Orion (emu) by 2.77× and Assise-numa improves throughput by another 1.93×. Assise scales linearly with the number of processes until it hits NVM write bandwidth, improving throughput by another 12.86×. Assise outperforms Orion by 69× and Ceph by 554× at scale.

5.4.5.2 Postfix

We use the unmodified Postfix mail server to measure the performance of parallel mail delivery. A load balancer machine forwards incoming email from as many client machines as necessary to maximize throughput to Postfix daemons running on 3 machines, configured as replicas. On each Postfix machine, a pool of delivery processes pull mail from the machine-local incoming mail queue and deliver it to user Maildir directories on a cluster-shared distributed file system. To ensure atomic delivery, a Postfix delivery process writes each incoming email to a new file in a process-private directory and then renames this file to the recipient's Maildir.

We send 80K emails from the Enron dataset [209], with each email reaching an average of 4.5 recipients. This results in a total of 360K email deliveries. Each email has an average size of 200KB (including attachments) and the dataset occupies 70GB. We repeat each experiment 3 times and report average mail delivery throughput and standard deviation (error bars) in Figure 5.11 over an increasing number of delivery processes, balanced over machines. We compare various Assise configurations and Ceph with 2 MDSes (1 and 3 MDSes performed similarly).

Round-robin. In the first configuration (Assise-rr) the load balancer uses a round-robin policy to send emails to mail queues. Due to a lack of locality, mails delivered to the same Maildir often require synchronization across machines, causing CC-NVM to frequently delegate leases remotely, which increases delivery latencies. Despite this, Assise-rr is able to outperform Ceph by up to 5.6× at scale. Ceph cannot improve throughput much further—even with 300 delivery processes, its throughput improves by 8% versus 48 processes.

Sharded. We shard Maildirs by Enron suborganization over machines [210]. The load balancer is configured to prefer the recipient's shard. For mail messages with multiple recipients, it picks the shard with the most receivers. In case of mail queue overload, the load balancer sends mail to a random unloaded shard. Sharding users in this manner provides up to 20% better performance (Assise-sharded) due to the fact that repeated deliveries to users of the same clique are likely to occur on the same server, allowing CC-NVM to synchronize delivery locally. At 15 processes, we are network-bound due to replication. Sharding did not improve Ceph's performance.

Private directories. We shard Maildirs by delivery process, using process IDs for Maildir subdirectories, thereby eliminating the need for synchronization (Assise-private). This change is not backward compatible with existing mail readers, but it is the logical limit for sharding-based optimization. Assise-private scales linearly until it is bottlenecked by network bandwidth, but performance is similar to Assise-sharded. This shows that local synchronization in Assise has minimal overhead. Ceph performance continues to be gated by the MDS.

Summary. Our results show that, with careful sharding of the workload, Assise's hierarchical coherence allows LibFS processes to synchronize deliveries locally, providing almost the same performance as private directories.

5.5 Conclusion

Assise is a distributed file system that provides low tail latency, high throughput, scalability, and high availability with a strong consistency model. To take advantage of low-latency NVM, Assise demonstrates that file system metadata and data should be colocated with applications. Colocation not only enables high performance, but also fast recovery. Assise proposes a novel, crash-consistent cache coherence protocol that can leverage the performance of NVM, while providing linearizability. Assise uses hot replicas in NVM to minimize application recovery time and ensure data availability, while leveraging a crash-consistent file system cache-coherence layer (CC-NVM) to provide scalability. In comparing with several state-of-the-art file systems, our results show that Assise improves write latency up to 22×, throughput up to 56×, fail-over time up to 103×, and scalability up to 6× versus Ceph, while providing stronger consistency semantics. Assise is available at https://github.com/ut-osa/assise.

Chapter 6

Offloading Arbitrary Computations to RNICs

Heterogeneous compute is becoming more prevalent, as CPU cycles turn into an increasingly scarce resource. Specifically, offloads to network compute devices have been gaining popularity [29, 30, 62, 63, 192, 211, 212]. System operators wish to reserve CPU cycles for application execution, while common, oft-repeated operations may be offloaded. NIC offloads, in particular, have the benefit that they reside in the network data path and NICs can carry out operations on in-flight data with low latency [62].

For this reason, remote direct memory access (RDMA) [24] has become ubiquitous [10]. Mellanox ConnectX NICs [25] have pioneered ubiquitous RDMA support and Intel has added RDMA support to their 800 series of Ethernet network adapters [31]. RDMA focuses on the offload of simple message passing (via Send/Recv verbs) and remote memory access (via Read/Write verbs) [24]. Both primitives are widely used in networked applications and their offload is extremely useful. However, RDMA is not designed for more complex offloads that are also common in networked applications. For example, remote data structure traversal and hash table access are not normally deemed realizable with RDMA [32]. This led to many RDMA-based systems requiring multiple network round-trips or to reintroduce involvement of the server's CPU to execute such requests [33, 34, 35, 36, 37, 38, 61].

To support complex offloads, the networking community has developed a number of SmartNIC architectures [16, 26, 213, 214, 215]. SmartNICs incorporate more powerful compute capabilities via CPUs or FPGAs. They can execute arbitrary programs on the NIC, including complex offloads. However, these

SmartNICs are not ubiquitous and their smaller volume implies a higher cost. SmartNICs can cost up to 5.7× more than commodity RDMA NICs (RNICs) at the same link speed (§6.1.1). Due to their custom architecture, they are also a management burden to the system operator, who has to support SmartNICs apart from the rest of the fleet.

We ask whether we can avoid this tradeoff and attempt to use the ubiquitous RNICs to realize complex offloads. To do so, we have to solve a number of challenges. First, we have to answer if and how we can use the RNIC interface, which consists only of simple data movement verbs (READ, WRITE, SEND, RECV, etc.) and no conditionals or loops, to realize complex offloads. Our solution has to be general so that offload developers can use it to build complex *RDMA programs* that can perform a wide range of functionality. Second, we have to ensure that our solution is efficient and that we understand the performance and performance variability properties of using RNICs for complex offloads. Finally, we have to answer how complex RNIC offloads integrate with existing applications.

In this work, we show that RDMA is *Turing-complete*, making it possible to use RNICs to implement complex offloads. To do so, we implement conditional branching via *self-modifying* RDMA verbs. Clever use of the existing compare-and-swap (CAS) verb enables us to dynamically modify the RNIC execution path by editing subsequent verbs in an RDMA program, using the CAS operands as a predicate. Just like self-modifying code executing on CPUs, self-modifying verbs require careful control of the execution path to avoid consistency issues due to RNIC verb prefetching. To do so, we rely on the WAIT and ENABLE RDMA verbs [192, 216] that provide execution dependencies. WAIT allows us to halt execution of new verbs until past verbs have completed, providing strict ordering among RDMA verbs. By controlling verb prefetching, ENABLE enforces consistency for verbs modified by preceding verbs. ENABLE also allows us to create loops by re-triggering earlier, already-executed verbs in an RDMA work queue—allowing the NIC to operate autonomously without CPU intervention.

Based on these primitives, we present RedN, a principled, practical approach to implementing complex RNIC offloads. Using self-modifying RDMA programs, we develop a number of building blocks that lift the existing RDMA verbs interface to a Turing-complete set of programming abstractions. Using these abstractions, we explore what is possible in terms of offload complexity and performance with just a commodity RNIC. We show how to integrate complex RNIC offloads, developed with RedN principles, into existing networked applications. RedN affords offload developers a practical way to implement complex NIC offloads on commodity RNICs, without the burden of acquiring and maintaining SmartNICs. Our code is available at: `https://redn.io`.

We make the following contributions:

• We present RedN, a principled, practical approach to offloading arbitrary computation to RDMA NICs. RedN leverages RDMA ordering and compare-and-swap primitives to build conditionals and loops. We show that these primitives are sufficient to make RDMA Turing-complete.
• Using RedN, we present and evaluate the implementation of various offloads that are useful in common server computing scenarios. In particular, we implement hash table lookup with Hopscotch hashing and linked list traversal.
• We evaluate the complexity and performance of offload in a number of use cases with the Memcached key-value store. In particular, we evaluate offload of common key-value get operations, as well as performance isolation and failure resiliency benefits. We demonstrate that RNIC offload with RedN can realize all of these benefits. It can reduce average latency of get operations by up to 2.6× compared to state-of-the-art one-sided RDMA key-value stores (*e.g.*, FaRM-KV [34]), as well as traditional two-sided RPC-over-RDMA implementations. Moreover, RedN provides superior performance isolation, improving latency by up to 35× under contention, while also providing higher availability under host-side failures.

6.1 Background

RDMA was conceived for high-performance computing (HPC) clusters, but it has grown out of this niche [10]. It is becoming ever-more popular due to the growth in network bandwidth, with stagnating growth in CPU performance, making CPU cycles an increasingly scarce resource that is best reserved to running application code. With RNICs now considered commodity, it is opportunistic to explore the use-cases where their hardware can yield benefits. These efforts, however, have been limited by the RDMA API, which constrains the expression of many complex offloads. Consequently, the networking community has built SmartNICs using FPGAs and CPUs to investigate new complex offloads.

6.1.1 SmartNICs

To enable complex network offloads, SmartNICs have been developed [16, 213, 217, 218]. SmartNICs include dedicated computing units or FPGAs, memory, and several dedicated accelerators, such as cryptography engines. For example, Mellanox BlueField [16] has 8×ARMv8 cores with 16GB of memory and 2×25GbE ports. These SmartNICs are capable of running full-fledged operating systems, but also ship with lightweight runtime systems that can provide kernel-bypass access to the NIC's IO engines.

Related work on SmartNIC offload. SmartNICs have been used to offload complex tasks from server CPUs. For example, StRoM [32] uses an FPGA NIC to implement RDMA verbs and creates generic kernels (or building blocks) that perform various functions, such as traversing linked lists. KV-Direct [29] uses an FPGA NIC to accelerate key-value accesses. iPipe [62] and Floem [212] are programming frameworks that simplify complex offload development for primarily CPU-based SmartNICs. E3 [211] transparently offloads microservices to SmartNICs.

The cost of SmartNICs. While SmartNICs provide the capabilities for complex offloads, they come at a cost. For example, a dual-port 25GbE BlueField SmartNIC at $2,340 costs 5.7× more than the same-speed ConnectX-5 RNIC at $410 (cf. [219]). Another cost is the additional management required for SmartNICs. SmartNICs are a special piece of complex equipment that system administrators need to understand and maintain. SmartNIC operating systems and runtimes can crash, have security flaws, and need to be kept up-to-date with the latest vendor patches. This is an additional maintenance burden on operators that is not incurred by RNICs.

6.1.2 RDMA NICs

The processing power of RDMA NICs (RNICs) has doubled with each subsequent generation. This allows RNICs to cope with higher packet rates and more complex, hard-coded offloads (e.g., reduction operations, encryption, erasure coding).

We measure the verb processing bandwidth of several generations of Mellanox ConnectX NICs, using the Mellanox `ib_write_bw` benchmark. This benchmark performs 64B RDMA writes and, as such, it is not network bandwidth limited due to the small RDMA write size. We find that the verb processing bandwidth doubles with each generation, as we can see in Table 6.1. This is primarily due to a doubling in processing units (PUs) in each generation.[1] As a result, ConnectX-6 NICs can execute up to 110 million RDMA verbs per second using a single NIC port. This increased hardware performance further motivates the need for exploiting the computational power of these devices.

Related work on RDMA offload. RDMA has been employed in many different contexts, including accelerating key-value stores and file systems [12, 34, 35, 37, 51], consensus [33, 36, 38, 61], distributed locking [220], and even nuanced use-cases such as efficient access in distributed tree-based indexing structures [221]. These systems operate within the confines of RDMA's intended

[1] Discussions with Mellanox affirmed our findings.

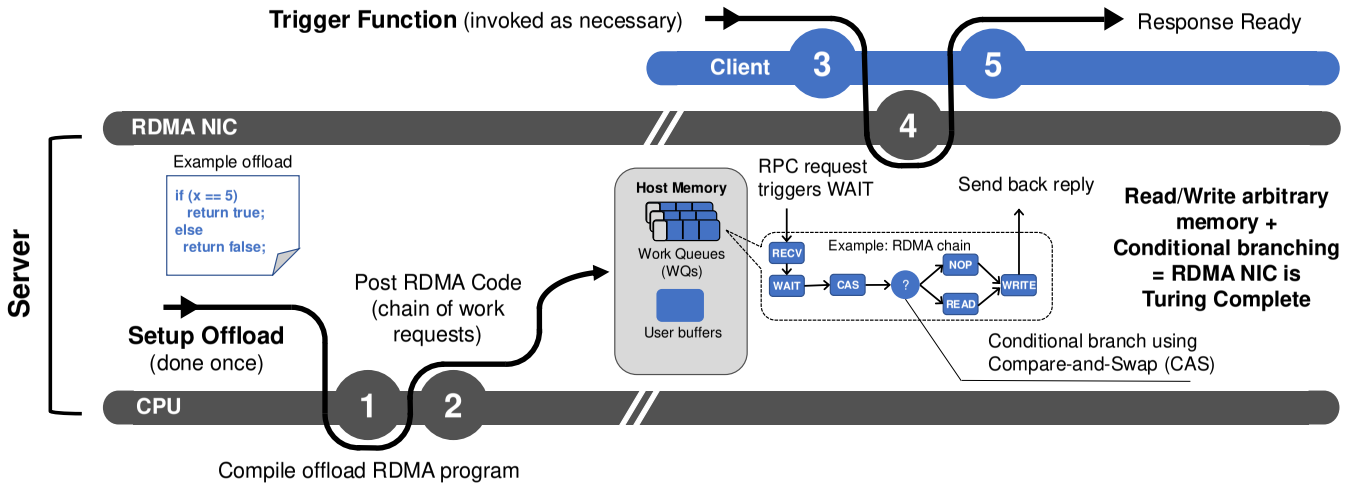


Figure 6.1: RDMA NICs can implement complex offloads if we allow conditional branches to be expressed. Conditional branching can be implemented by using CAS verbs to modify subsequent verbs in the chain, without any hardware modification.

RNIC	PUs	Throughput
ConnectX-3 (2014)	2	15M verbs/s
ConnectX-5 (2016)	8	63M verbs/s
ConnectX-6 (2017)	16	112M verbs/s

Table 6.1: # of Processing Units (PUs) and performance of ConnectX NICs.

use as a *data movement* offload (via remote memory access and message passing). When complex functionality is required, these systems involve multiple RDMA round-trips and/or rely on host CPUs to carry out the complex operations.

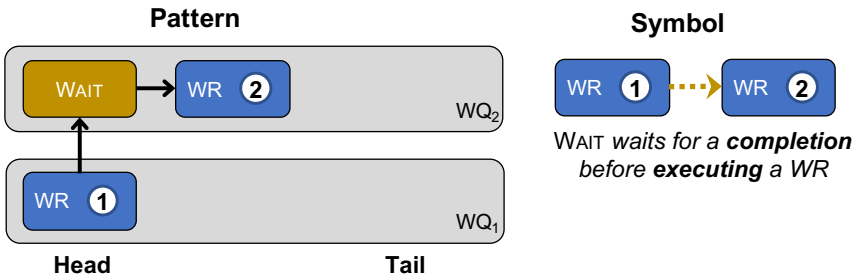
Within the storage context, Hyperloop [192] demonstrated that pushing the RNIC offload capabilities is possible. Hyperloop combines RDMA verbs to implement complex storage operations, such as chain replication, without CPU involvement. However, it does not provide a blueprint for offloading arbitrary processing and cannot offload functionality that uses any type of conditional logic (e.g., walking a remote data structure). Moreover, the Hyperloop protocol is likely incompatible with next-generation RNICs, as its implementation relies on changing work request ownership—a feature that is deprecated for ConnectX-4 and newer cards.

Unlike this body of previous work, we aim to unlock the general-purpose processing power of RNICs and provide an unprecedented level of programmability for complex offloads, by using novel combinations of existing RDMA verbs (§6.2).

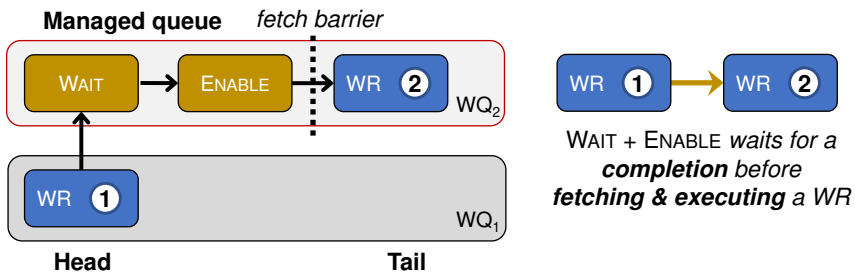
6.2 The RedN Computational Framework

To achieve our aforementioned goals, we develop a framework that enables complex offloads, called RedN. RedN's key idea is to combine widely available capabilities of RNICs to enable self-modifying RDMA programs. These programs—chains of RDMA operations—are capable of executing dynamic control flows with conditionals and loops. Fig. 6.1 illustrates the usage of RedN. The setup phase involves (1) preparing/compiling the RDMA code required for the service and (2) posting the output chain(s) of RDMA WRs to the RNIC. Clients can then use the offload by invoking a trigger (3) that causes the server's RNIC to (4) execute the posted RDMA program, which returns a response (5) to the client upon completion.

To further understand this proposed framework, we first look into the execution models offered by RNICs, and the ordering guarantees they provide for RDMA verbs. We then look into the expressivity of traditional RDMA verbs and explore



(a) Completion order.



(b) Doorbell order.

Figure 6.2: Work request ordering modes that guarantee a total order of operations 6.2a and, a more restrictive "doorbell" order 6.2b, where operations are fetched by the NIC one-by-one. The symbols on the right will be used as notation for these WR chains in the examples of §6.2.

parallels with CPU instruction sets. We use these insights to describe strategies for expressing complex logic using traditional RDMA verbs, *without requiring any hardware modifications.*

6.2.1 RDMA execution model

The RDMA interface specifies a number of data movement verbs (READ, WRITE, SEND, RECV, etc.) that are *posted* as *work requests* (WRs) by offload developers into *work queues* (WQs) in host memory. The RNIC starts execution of a sequence of WRs in a WQ once the offload developer triggers a *doorbell*—a special register in RNIC memory that informs the RNIC that a WQ has been updated and should be executed.

Work request ordering. Ordering rules for RDMA WRs distinguish between write WRs and non-write WRs that return a value. Within each category of operations, RDMA guarantees in-order execution of WRs within a single WQ. In particular, write WRs (i.e., SEND, WRITE, WRITEIMM) are totally ordered with regard to each other, but writes may be reordered before prior non-write WRs.

We call the default RDMA ordering mode *work queue (WQ) ordering*. Sophisticated offload logic often requires stronger ordering constraints, which we construct with the help of two RDMA verbs. Fig. 6.2 shows two stricter ordering modes that we introduce and how to achieve them.

The WAIT verb stops WR execution until the completion of a specified WR from another WQ or the preceding WR in the same WQ. We call this *completion ordering* (Fig. 6.2a). It achieves total ordering of WRs along the execution chain (which potentially involves multiple WQs). It can be used to enforce data consistency, similar to data memory barriers in CPU instruction sets—to wait for data to be available before executing the WRs operating on the data. Moreover, WAIT allows developers to *pre-post* chains of RDMA verbs to the RNIC, without immediately executing them.

In all the aforementioned ordering modes, the RNIC is free to prefetch into its cache the WRs within a WQ. Thus, the execution outcome reflects the WRs at the time they were fetched, which can be incoherent with the versions that reside in host memory in case these were later modified. To avoid this issue, the RNIC allows placing a WQ into *managed* mode, in which WR prefetch is disabled. The ENABLE verb is then used to explicitly start the prefetching of WRs. This allows for existing WRs to be modified within the WQ, as long as this is done before completion of the posted ENABLE—similar to an instruction barrier. We achieve a full (data and instruction) barrier, by using WAIT and ENABLE in sequence. We call this *doorbell ordering* (Fig. 6.2b). Doorbell ordering allows developers to modify WR chains in-place. In particular, it allows for *data-dependent, self-modifying* WRs.

Thus, we have shown that we can control WR fetch and execution via special verbs, which we will exploit in the next section to develop full-fledged RDMA programs. These verbs are widely available in commodity RNICs (e.g., Mellanox terms them cross-channel communication [216]).

6.2.2 Dynamic RDMA Programs

While a static sequence of RDMA WRs is already a rudimentary RDMA program, complex offloads require *data-dependent execution*, where the logic of the offload is dependent on input arguments. To realize data-dependent execution, we construct *self-modifying RDMA code*.

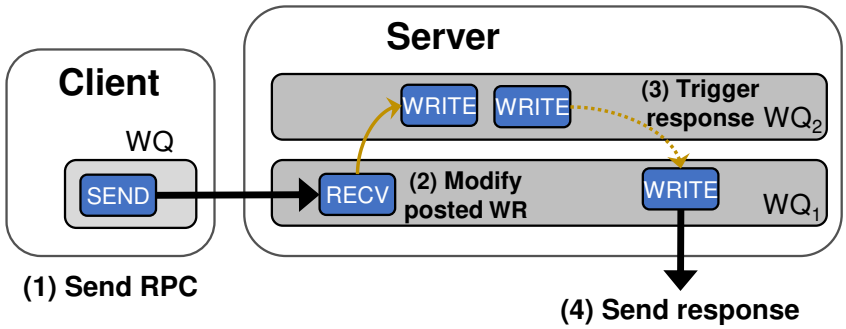


Figure 6.3: Clients can trigger posted operations. Thick solid lines represent (meta)data movements.

Self-modifying RDMA code. Doorbell ordering enables a restricted form of self-modifying code, capable of data-dependent execution. To illustrate this concept, we use the example of a server host that offloads an RPC handler to its RNIC as shown in Fig. 6.3. The RPC response depends on the argument set by the client and thus the RDMA offload is data-dependent. The server posts the RDMA program that consists of a set of WRs spanning two WQs. The client invokes the offload by issuing a SEND operation. At the RNIC, the SEND triggers the posted RECV operation. Observe that RECV specifies where the SEND data is placed. We configure RECV to inject the received data into the posted WR chain in WQ_2 to modify its attributes. We achieve this by leveraging doorbell ordering, to ensure that posted WRs are not prefetched by the RNIC and can be altered by preceding WRs.

This is an instance of self-modifying code. As such, clients can pass arguments to the offloaded RPC handler and the RNIC will dynamically alter the executed code accordingly. However, this by itself is not sufficient to provide a Turing-complete offload framework.

Turing completeness of RDMA. Turing completeness implies that a system of data-manipulation rules, such as RDMA, are computationally universal. For RDMA to be Turing-complete, we need to satisfy two requirements [222]:
T1: Ability to read/write arbitrary amounts of memory.
T2: Conditional branching (e.g. if/else statements).

T1 can be satisfied for limited amounts of memory with regular RDMA verbs, whereas T2 has not been demonstrated with RDMA NICs. However, to truly be capable of accessing an *arbitrary* amount of memory, we need a way of realizing

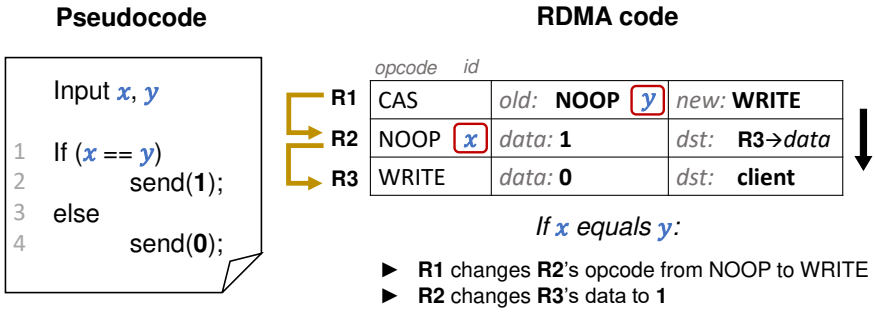


Figure 6.4: Simple *if* example and equivalent RDMA code. Conditional execution relies on self-modifying code using CAS to enable/disable WRs based on the operand values.

loops. Loops open up a range of sophisticated use-cases and lower the number of constraints that programmers have to consider for offloads. To highlight their importance, we add them as a third requirement, necessary to fulfill the first:
T3: The ability to execute code repeatedly (loops).

In the next sub-sections, we show how dynamic execution can be used to satisfy all the aforementioned requirements. A proof sketch of Turing completeness is given in Appendix A.

6.2.3 Conditionals

Conditional execution—choosing what computation to perform based on a runtime condition—is typically realized using conditional branches, which are not readily available in RDMA. To this end, we introduce a novel approach that uses self-modifying CAS verbs. The main insight is that this verb can be used to check a condition (*i.e.*, equality of x and y) and then perform a swap to modify the attributes of a WR. We describe how this is done in Fig. 6.4. We insert a CAS that compares the 64-bit value at the address of R2's *opcode* attribute (initially NOOP) with its *old* parameter (also initially NOOP). We then set the *id* field of R2 to x. This field can be manipulated freely without changing the behavior of the WR, allowing us to use it to store x. Operand y is stored in the corresponding position in the *old* field of R1. This means that if x and y are equal, the CAS operation will succeed and the value in R1's *new* field—which we set to WRITE—will replace R2's opcode. Hence, in the case $x = y$, R2 will change from a NOOP into a WRITE operation. This WRITE is set to modify the *data* value of the return operation (R3) to 1. If x and y are not equal, the default value 0 is returned.

Now that we have established the utility of this technique for basic conditionals,

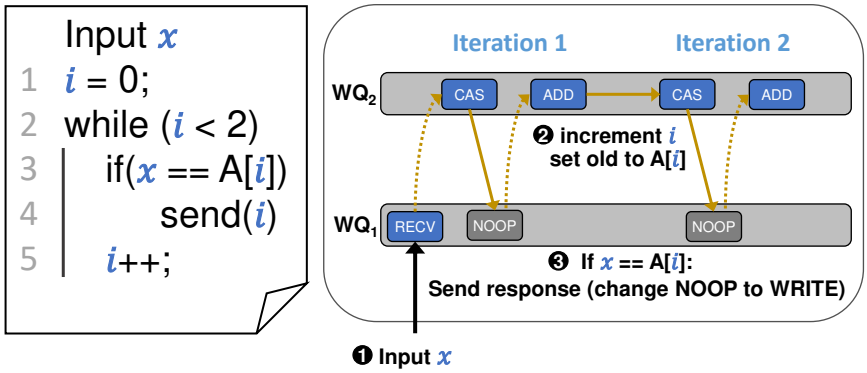


Figure 6.5: while loop using CAS. Loop is unrolled since loop size is fixed and set to 2.

we next look into how to can be used to support loop constructs.

6.2.4 Loops

To support loop constructs efficiently, we require (1) conditional branching to test the loop condition and break if necessary, and (2) WR re-execution, to repeat the loop body. We develop each, in turn, below.

Consider the while loop example in Figure 6.5. This offload searches for x in an array A and sends the corresponding index. The loop is static because A has finite size (in this case, size =2), known a priori. To simplify presentation, consider the case $A[i] = i, \forall i$. Without this simplification, the example would include an additional WRITE to fetch the value at $A[i]$.

The loop body uses a CAS verb to implement the if condition (line 3), followed by an ADD verb to increment i (line 6). Given that the loop size is known a priori (size $=$ 2), RedN can unroll the while loop in advance and post the WRs for all iterations. As such, there is no need to check the condition at line 2. For each iteration, if the CAS succeeds, the NOOP verb in WQ_1 will be changed to WRITE—which will send the response back to the client. However, it is clear that, regardless of the comparison result, all subsequent iterations will be executed. This is inefficient since, if the send (line 4) occurs before the loop is finished, a number of WRs will be wastefully executed by the NIC. This is impractical for larger loop sizes or if the number of iterations is not known a priori.

Unbounded loops and termination. Figure 6.6 modifies the previous example to make it such that the loop is unbounded. For efficiency, we add a break that

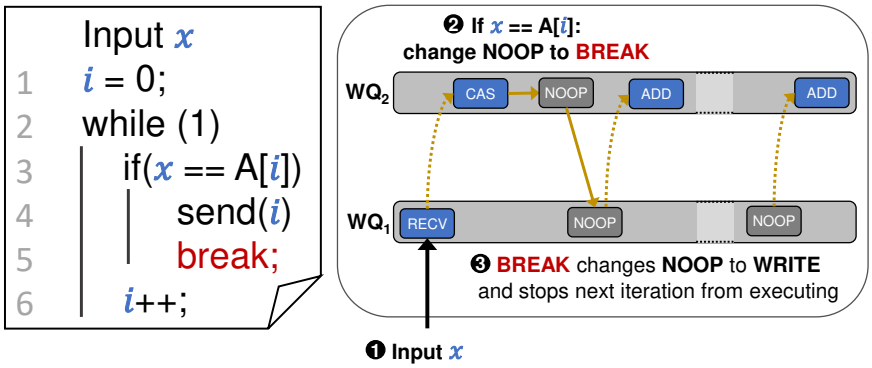


Figure 6.6: *while* loop with breaks realized using CAS. Breaking can be achieved by using CAS to change a NOOP WR to an RDMA WRITE that prevents the execution of subsequent iterations in the loop.

exits the loop if the element is found. The role of break is to prevent additional iterations from being executed. We use an additional NOOP that is formatted such that, once transformed into a WRITE by the CAS operation, it prevents the execution of subsequent iterations in the loop. This is done by modifying the last WR in the loop such that it does not trigger a completion event. The next iteration in the loop, which WAITS on such an event (via completion ordering), will therefore not be executed. Moreover, the WRITE will also modify the opcode of the WR used to send back the response from NOOP to WRITE.

As such, break allows efficient and unbounded loop execution. However, it still remains necessary for the CPU to post WRs to continue the loop after all its WRs are executed. This consumes CPU cycles and can even increase latency if the CPU is unable to keep up with the speed of WR execution.

Unbounded loops via WQ recycling. To allow the NIC to recycle WRs without CPU intervention, we make use of a novel technique that we call *WQ recycling*. RNICs iterate over WQs, which are circular buffers, and execute the WRs therein. By design, each WR is meant to execute only once. However, there is no fundamental reason why WRs cannot be reused since the RNIC does not actually erase them from the WQ. To enable recycling of a WR chain, we insert a WAIT and ENABLE sequence at the tail of the WQ. This instructs the RNIC to wrap around the tail and re-execute the WR chain for as many times as needed.

It is important to note that WQ recycling is not a panacea. To allow the tail of the WQ to wrap around, all posted WAIT and ENABLE WRs in the loop need to have their *wqe_count* attribute updated. This attribute is used to determine the index

RedN Constructs		Number of WRs	Operand limit [bits]
if		1C + 1A + 3E	48
while	Unrolled	1C + 1A + 3E	
	Recycled	3C + 2A + 4E	

Table 6.2: Breakdown of the overhead of our constructs. *C* refers to copy verbs, *A* refers to atomic RDMA verbs, and *E* refers to WAIT/ENABLE verbs. while loops that use WQ recycling incur 2 additional READS, 1 ADD, and 1 ENABLE WR.

of the WR that these ordering verbs affect. In ConnectX NICs, these indices are maintained internally by the RNIC and their values are monotonically increasing (instead of resetting after the WQ wraps around). As such, the *wqe_count* values need to be incremented to match. This incurs overhead (as seen in Table 6.2) and requires an additional ADD operation in combination with other verbs. As such, loop unrolling, where each iteration is manually posted by the CPU, is overall less taxing on the RNIC. However, WQ recycling avoids CPU intervention, allowing the offload to remain available even amid host software failures (as we will see later in §6.4.6).

6.2.5 Putting it all together

With conditional branching, we can dynamically alter the control flow of any function on an RNIC. Loops allow us to traverse arbitrary data structures. Together, we have transformed an RNIC into a general processing unit. In this section, we discuss the usability aspects from overhead, security, programmability, and expressiveness perspectives.

Building blocks. We abstract and parameterize the RDMA chains required for conditional branching and looping into if and while constructs. The overhead in terms of RDMA WR chains of our constructs is shown in Table 6.2. We can see a breakdown of the minimum number of operations required for each. Inequality predicates, such as $<$ or $>$, can also be supported by combining equality checks with MAX or MIN, as seen later in Table 6.3. However, their availability is vendor-specific and currently only supported by ConnectX NICs.

Operand limits. RedN's limit is based on the supported size for the CAS verb, which is 64 bits. The operand is provided as a 48-bit value, encoded in its *id* and other neighboring fields (which can also be freely modified without affecting execution). The remaining bits are used for modifying the opcode of the WR

depending on the result of the comparison. We note that our advertised limits only signify what is possible with the number of operations we allocate for our constructs. For instance, despite the 48-bit operand limit for our constructs, we can chain together multiple CAS operations to handle different segments of a larger operand (we do not rely on the atomicity property of CAS). As such, there is no fundamental limitation, only a performance penalty.

Offload setup. To offload an RDMA program, clients first create an RDMA connection to the target server and send an RPC to initiate the offload. We envision that the server already has the offload code; however, other ways of deploying the offload are possible. Upon receiving a connection request, the server creates one or more managed local WQs to post the offloaded code. Next, it registers two main types of memory regions for RDMA access: (a) a code region, and (b) a data region. The code region is the set of remote RDMA WQs created on the server, which are unique to each client and need to be accessible via RDMA to allow self-modifying code. Code regions are protected by memory keys—special tokens required for RDMA access—upon registration (at connection time), prohibiting unauthorized access. The data region holds any data elements used by the offload (*e.g.*, a hash table). Data regions can be shared or private, depending on the use-case.

Security. RedN does not solve security challenges in existing RDMA or Infiniband implementations [223]. However, RedN can help RDMA systems become more secure. For such systems, *one-sided* RDMA operations (*e.g.*, RDMA READ and WRITE) are frequently used [34, 37, 50, 192, 224, 225] as they avoid CPU overheads at the responder. However, doing so requires clients to have direct read and/or write memory access. This can compromise security if clients are buggy and/or malicious. To give an example, FaRM allows clients to write messages directly to shared RPC buffers. This requires clients to behave correctly, as they could otherwise overwrite or modify other clients' RPCs. RedN allows applications to use *two-sided* RDMA operations (*e.g.*, SEND and RECV), which do not require direct memory access, while *still* fully bypassing server CPUs. As we demonstrate in our use-cases in §6.4, SEND operations can be used to trigger offload programs without any CPU involvement.

Isolation. Given that RedN implements dynamic loops, clients can abuse such constructs to consume more than their fair share of resources. Luckily, popular RNICs, like ConnectX, provide WQ rate-limiters [226] for performance isolation. As such, even if clients trigger non-terminating offload code, they still have to adhere to their assigned rates. Moreover, offloaded code can be configured by the servers to be auditable through completion events, created automatically after a WR

is executed. These events can be monitored and servers can terminate connections to clients running misbehaving code.

Parallelism. RDMA WR fetch and execution latencies are more costly compared to CPU instructions, as WRs are fetched/executed via PCIe (microseconds vs. nanoseconds). As such, to hide WR latencies, it is important to parallelize logically unrelated operations. Like threads of execution in a CPU, each WQ is allocated a single RNIC PU to ensure in-order execution without inter-PU synchronization. As such, we carefully tune our offloaded code to allow unrelated verbs to execute on independent queues to be able to parallelize execution as much as possible. The benefits of parallelism are evaluated in §6.4.2.

6.3 Implementation

Our offload framework is implemented in C with ∼2,300 lines of code—this includes our use cases (∼1400), and convenience wrappers for RDMA verbs (*libibverbs*) API (∼900).

Our approach does not require modifying any RDMA libraries or drivers. RedN uses low-level functions provided by Mellanox's ConnectX driver (*libmlx5*) to expose in-memory WQ buffers and register them to the RNIC, allowing WRs to be manipulated via RDMA verbs. We configure the ConnectX-5 firmware to allow the WR *id* field to be manipulated freely, which is required for conditional operations as well as WR recycling. This is done by modifying specific configuration registers on the NIC [227].

RedN is compatible with any ConnectX NICs that support WAIT and ENABLE (*e.g.*, ConnectX-3 and later models).

6.4 Evaluation

We start by characterizing the underlying RNIC performance (§6.4.1) to understand how it affects our implemented programming constructs. Then, in our evaluation against state-of-the-art RNIC and SmartNIC offloads, we show that RedN:

1. Speeds up remote data structure traversals, such as hash tables (§6.4.2) and linked lists (§6.4.3) compared to vanilla RDMA offload;
2. Accelerates (§6.4.4) and provides performance isolation (§6.4.5) for the Memcached key-value store;
3. Provides improved availability for applications (§6.4.6)—allowing them to run in spite of OS & process crashes;

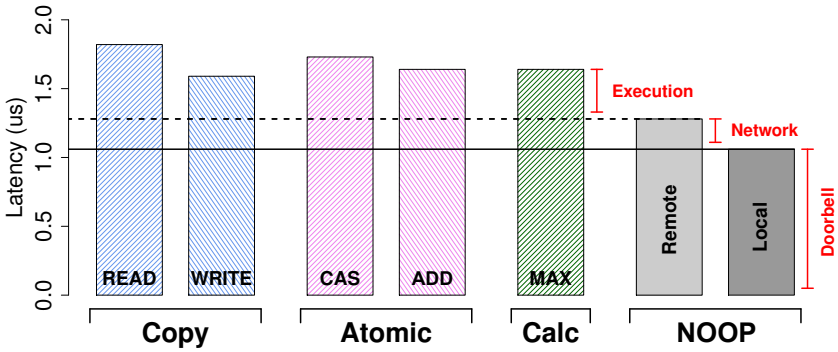


Figure 6.7: Latencies of different RDMA verbs. The solid line marks the latency of ringing the doorbell via MMIO. The difference between dashed and solid lines estimates network latency.

4. Exposes programming constructs generic enough to enable a wide-variety of use-cases (§6.4.2–§6.4.6);

Testbed. Our experimental testbed consists of 3× dual-socket Haswell servers running at 3.2 GHz, with a total of 16 cores, 128 GB of DRAM, and 100 Gbps dual-port Mellanox ConnectX-5 Infiniband RNICs. All nodes are running Ubuntu 18.04 with Linux Kernel version 4.15 and are connected via back-to-back Infiniband links.

NIC setup. For all of our experiments, we use reliable connection (RC) RDMA transport, which supports the RDMA synchronization features we use. All WQs that enforce doorbell order are initialized with a special "managed" flag to disable the driver from issuing doorbells after a WR is posted. The WQ size is set to match that of the offloaded program.

6.4.1 Microbenchmarks

We run microbenchmarks to break down RNIC verb execution latency, understand the overheads of our different ordering modes, and determine the processing bandwidth of different RDMA verbs and of our constructs.

6.4.1.1 RDMA Latency

We break down the performance of RDMA verbs, configured to perform 64B IO, by measuring their average latencies after executing them 100K times. All verbs are

executed remotely, unless otherwise stated. As seen in Fig. 6.7, WRITE has a latency of 1.6 µs. It uses posted PCIe transactions, which are one-way. Comparatively, non-posted verbs such as READ or atomics such as fetch-and-add (ADD) and compare-and-swap (CAS) need to wait for a PCIe completion and take ∼1.8 µs.[1] Overall, the execution time difference is small among verbs, even for more advanced, vendor-specific *Calc* verbs that perform logical and arithmetic computations (*e.g.*, MAX).

To break down the different latency components for RDMA verb execution, we first estimate the latency of issuing a doorbell and copying the WR to the RNIC. This can be done by measuring the execution time of a NOOP WR. This time can be subtracted from the latencies of other WRs to give an estimate of their execution time once the WR is available in the RNIC's cache. We also quantify the network cost by executing remote and local loopback NOOP WRs (shown on the right-hand side) and measuring the difference—roughly 0.25 µs for our back-to-back connected nodes. Overall, these results show low verb execution latency, justifying building more sophisticated functions atop. We next measure the implications of ordering for offloads.

6.4.1.2 Ordering Overheads

We show the latency of executing chains of RDMA verbs using different ordering modes. All the posted WRs within a chain are NOOP, to simplify isolating the performance impact of ordering. We start by measuring the latency of executing a chain of verbs posted to the same queue but absent any constraints (WQ order), and compare it to the ordering modes that we introduced in Fig. 6.2—completion order and doorbell order. WQ order only mandates in-order updates to memory, which allows for increased concurrency. Operations that are not modifying the same memory address can execute concurrently and the RNIC is free to prefetch multiple WRs with a single DMA[1]. We can see in Fig. 6.8 that the latency of a single NOOP is 1.21 µs and the overhead of adding subsequent verbs is roughly 0.17 µs per verb. The first verb is slower since it requires an initial doorbell to tell the NIC that there is outstanding work. For completion ordering, less concurrency is possible since WRs await the completions of their predecessors, and the overhead of increases slightly to 0.19 µs per additional WR. For doorbell order, no latency-hiding is possible, as the NIC has to fetch WRs from memory one-by-one, which results in an overhead of 0.54 µs per additional WR. These results signify that, doorbell ordering should be used conservatively, as there is more than 0.5 µs latency increase for every instance of its use, compared to more relaxed ordering modes.

[1] Older-generation NICs (*e.g.*, ConnectX-4) use a proprietary concurrency control mechanism to implement atomics, resulting in higher latencies than later generations that rely on PCIe atomic transactions. [WALEED: add cite] [1] The number of operations fetched by the RNIC can change dynamically. The Prefetch mechanism in ConnectX RNICs is proprietary.

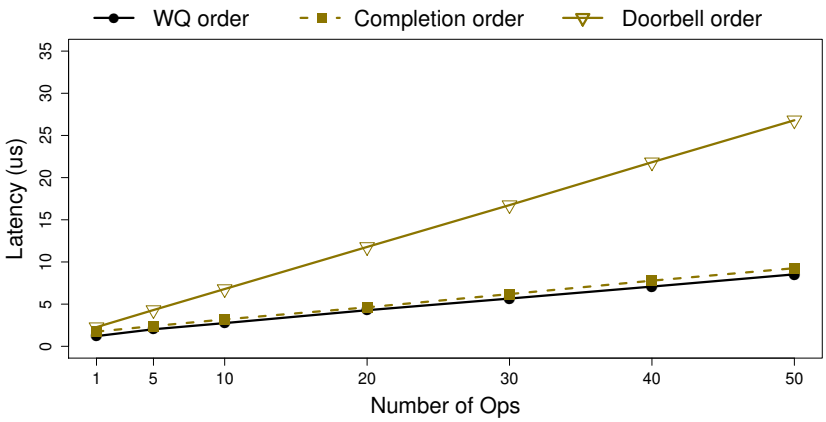


Figure 6.8: Execution latency of RDMA verbs posted using different ordering modes. More restrictive modes such as Doorbell order add non-negligible overheads as it requires the NIC to fetch WRs sequentially.

6.4.1.3 RDMA Verb Throughput

We show the throughput of the common RDMA verbs in Table 6.3 for a single ConnectX-5 port. ConnectX cards assign compute resources on a per port basis. For ConnectX-5, each port has 8 PUs. Atomic verbs, such as CAS, offer a comparatively limited throughput (8× lower than regular verbs) due to memory synchronization across PCIe.

In addition, we measure the performance of RedN's if and while constructs. Using 48-bit operands, a ConnectX-5 NIC can execute 700K if constructs per second. This is due to the need for CAS to ensure doorbell ordering between CAS and the subsequent WR it modifies. This causes the throughput to be bound by NIC processing limits. Unrolled while loops require the same number of verbs per iteration as an if statement and their throughput is identical. while loops with WQ recycling have reduced performance due to having to execute more WRs per iteration.

6.4.2 Offload: Hash Lookup

After evaluating the overheads of RedN's ordering modes and constructs, we next look into the performance of RedN for offloading remote access to popular data structures. We first look into hash tables, given their prominent use in key-value stores for indexing stored objects. To perform a simple *get* operation, clients first have to lookup the desired key-value entry in the hash table. The entry can either

Operation			Throughput (M ops/s)	Support
Atomic	CAS		8.4	Native
	ADD			
Copy	READ		65	
	WRITE		63	
Calc	MAX		63	Mellanox
Constructs	if		0.7	RedN
	while	Unrolled	0.7	
		Recycled	0.3	

Table 6.3: Throughput of common RDMA verbs and RedN's constructs on a single port of a ConnectX-5. if and unrolled while have identical performance. while loops with WQ recycling require additional WRs and therefore have a lower throughput.

have the value directly inlined or a pointer to its memory address. The value is then fetched and returned back to the client. Hopscotch hashing is a popular hashing scheme that resolves collisions by using H hashes for each entry and storing them in 1 out of H buckets. Each bucket has a *neighborhood* that can probabilistically hold a given key. A lookup might require searching more than one bucket before the matching key-value entry is found. To support dynamic value sizes, we assume the value is not inlined in the bucket and is instead referenced via a pointer.

For distributed key-value stores built with RDMA, *get* operations are usually implemented in one of two ways:

One-sided approaches first retrieve the key's location using a one-sided RDMA READ operation and then issue a second READ to fetch the value. These approaches typically require two network round-trips at a minimum. This greatly increases latency but does not require involvement of the server's CPU. Many systems utilize this approach to implement lookups, including FaRM [34] and Pilaf [37].

Two-sided approaches require the client to send a request using an RDMA SEND or WRITE. The server intercepts the request, locates the value and then returns it using one of the aforementioned verbs. This widely used [12, 35] approach follows traditional RPC implementations and avoids the need for several roundtrips. However, this comes at the cost of server CPU cycles.

6.4.2.1 RedN's Approach

To offload key-value *get* operations, we leverage the offload schemes introduced in §6.2.3 and §6.2.4.

Fig. 6.9 describes the RDMA operations involved for a single-hash lookup. To *get* a value corresponding to a key, the client first computes the hashes for its key. For this use-case, we set the number of hashes to two, which is common in practice [228]. The client then performs a SEND with the value of the key x and address of the first bucket $H_1(x)$, which are then captured via a RECV WR posted on the server. The RECV WR (R1) inserts x into the old field of the CAS WR (R3) and the bucket address $H_1(x)$ into the READ WR (R2). The READ WR retrieves the bucket and sets the source address (*src*) of the response WR (R4) to the address of the value (*ptr*). It also inserts the bucket's key into the *id* field to prepare it for the conditional check. Finally, CAS (R3) checks whether the expected value old, which is set to key x, matches the *id* field in (R4), which is set to the bucket's key. If equal, (R4)'s opcode is changed from NOOP to WRITE, which then returns the value from the bucket. Given that each key may be stored in multiple buckets (two in our setup), these lookups may be performed sequentially or in parallel, depending on the offload configuration.

6.4.2.2 Results

We evaluate our approach against both one-sided and two-sided implementations of key-value *get* operations. We use FaRM's approach [34] to perform one-sided lookups. FaRM uses Hopscotch hashing to locate the key using approximately two RDMA READs — one for fetching the buckets in a neighborhood that hold the key-value pairs and another for reading the actual value. The neighborhood size is set to 6 by default, implying a 6× overhead for RDMA metadata operations. For two-sided lookups, our RPC to the host involves a client-initiated RDMA SEND to transmit the *get* request, and an RDMA WRITE initiated by the server to return the value after performing the lookup.

Latency. Fig. 6.10 shows a latency comparison of KV *get* operations of RedN against one-sided and two-sided baselines. We evaluate two distinct variations of two-sided. The *event-based* approach blocks for a completion event to avoid wasting CPU cycles, whereas the *polling-based* approach dedicates one CPU core for polling the completion queue. We use 48-bit keys and vary the value size. The value size is given on the x-axis. In this scenario, we assume no hash collisions and that all keys are found in the first bucket. RedN is able to outperform all baselines — fetching a 64 KB key-value pair in 16.22 µs, which is within 5% of a single network round-trip READ (Ideal). RedN is able to deliver

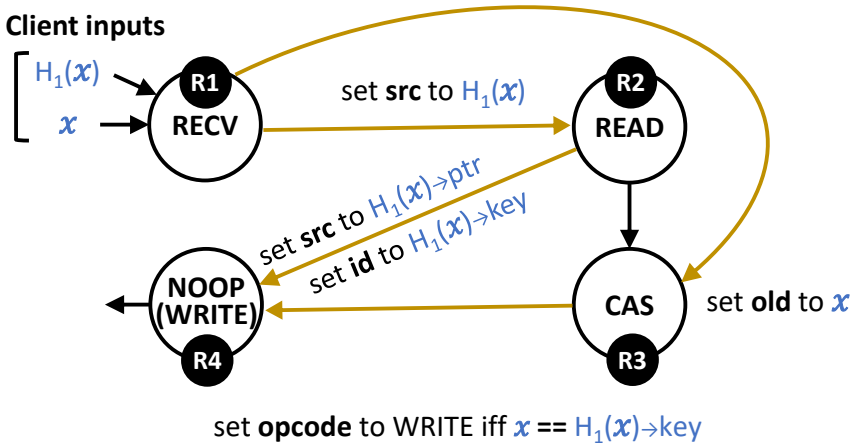


Figure 6.9: Hash lookup RDMA program. Black arrows indicate order of execution of WRs in their WQs. Brown arrows indicate self-modifying code dependencies and require doorbell ordering. x is the requested key and $H_1(x)$ is its first hash. The acronym *src* indicates the "source address" field of WRs. *old* indicates the "expected value" at the target address of the CAS operation. The *id* field is used for storing conditional operands.

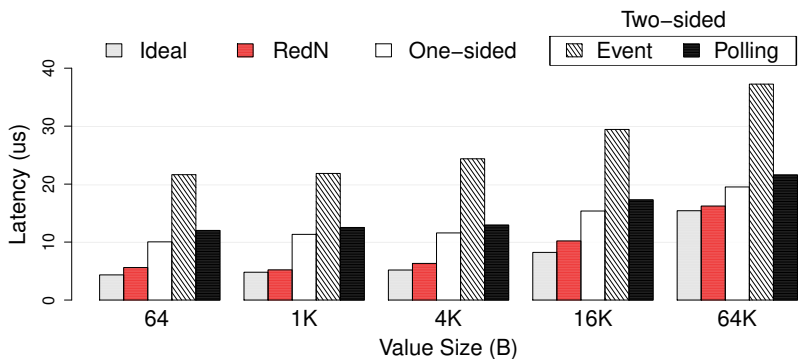


Figure 6.10: Average latency of hash lookups. *Ideal* shows the latency of a single network round-trip READ.

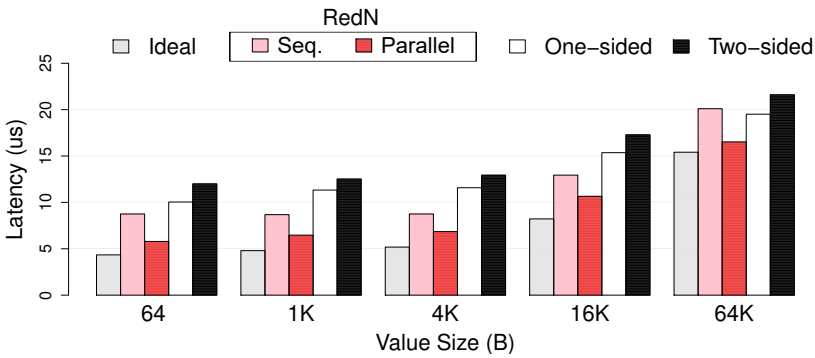


Figure 6.11: Average latency of hash lookups during collisions. *Ideal* shows the latency of a single network round-trip READ.

close-to-ideal performance because it bypasses the server's CPU ***and*** fetches the value in a single network RTT. Compared to RedN, one-sided operations incur up to 2× higher latencies, as they require two RTTs to fetch a value. Two-sided implementations do not incur any extra RTT; however, they require server CPU intervention. The polling-based variant consumes an entire CPU core but provides competitive latencies. Event-based approaches block for completion events to avoid wasting CPU cycles and incur much higher latencies as a consequence. RedN is able to outperform polling-based and event-based approaches by up to 2 and 3.8×, respectively. Given the much higher latencies of event-based approaches, for the remainder of this evauation, we will only focus on polling-based approaches and simply refer to them hereafter as *two-sided*.

Fig. 6.11 shows the latency in the presence of hash collisions. In this case, we assume a worst case scenario, where the key-value pair is always found in the second bucket. In this scenario, we introduce two offload variants for RedN—RedN-Seq & RedN-Parallel. The former performs bucket lookups sequentially within a single WQ. The latter parallelizes bucket lookups by performing the lookups across two different WQs to allow execution on different NIC PUs. We can see that RedN-Parallel maintains similar latencies to lookups with no hash collisions (*i.e.*, *RedN* in Fig. 6.10), since bucket lookups are almost completely parallelized. It is worth noting that parallelism in this case does not cause unnecessary data movement, since the value is only returned when the corresponding key is found. For the other bucket, the WRITE operation (R4 in Fig. 6.9) is a NOOP. RedN-Seq, on the other hand, incurs at least 3 µs of extra latency as it needs to search the buckets one-by-one. As such, whenever possible, operations with no dependencies should be executed in parallel. The trade-off is

Hash lookup	**IO Size**			
	$\leq$ 1 KB		**64 KB**	
Port config.	Single	Dual	Single	Dual
Rate (ops/s)	500K	1M	180K	190K
Bottleneck	NIC PU		IB bw	PCIe bw

Table 6.4: NIC throughput of hash lookups and its bottlenecks.

IO Size	**System**	**Median**	**99^{th}ile**
64 B	RedN	5.7 µs	6.9 µs
	StRoM	~7 µs	~7 µs
4 KB	RedN	6.7 µs	8.4 µs
	StRoM	~12 µs	~13 µs

Table 6.5: Latencies of hash *gets*. StRoM results obtained from [32].

having to allocate extra WQs for each level of parallelism.

Throughput. We describe our throughput in Table 6.4. At lower IO, RedN is bottlenecked by the NIC's processing capacity due to the use of doorbell ordering—reaching 500K ops/s on a single port (1M ops/s with dual ports). At 64 KB, RedN reaches the single-port IB bandwidth limit (~ 92 Gbps). Dual-port configs are limited by ConnectX-5's 16× PCIe 3.0 lanes.

SmartNIC comparison. We compare our performance for hashtable *gets* against StRoM [32], a programmable FPGA-based SmartNIC. Since we do not have access to a programmable FPGA, we extract the results from [32] for comparison, and report them in Table 6.5. RedN uses the same experimental settings as before. Our hashtable configuration is functionally identical to StRoM's and our client and server nodes are also connected via back-to-back links. We can see that RedN provides lower lookup latencies than StRoM. StRoM uses a Xilinx Virtex 7 FPGA, which runs at 156.25 MHz, and incurs at least two PCIe roundtrips to retrieve the key and value. Our evaluation shows that RedN can provide latency that is in-line with more expensive SmartNICs.

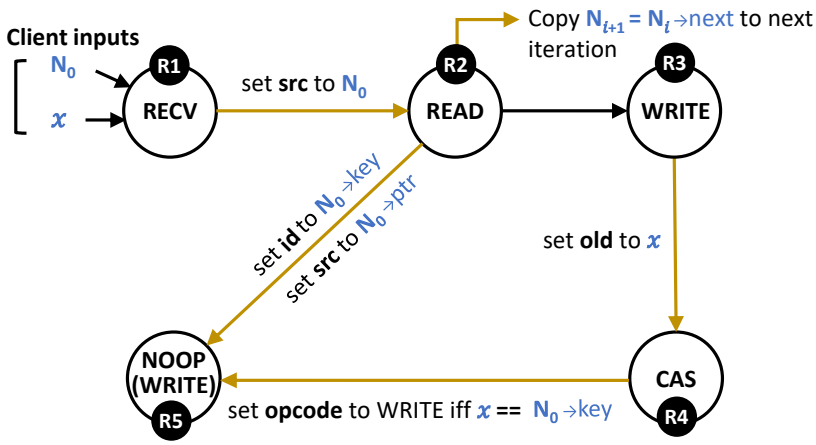


Figure 6.12: Linked list RDMA program.

6.4.3 Offload: List Traversal

Next, we explore another data structure also popularly used in storage systems. We focus on linked lists that store key-value pairs, and evaluate the overhead of traversing them remotely using our offloads. Similar to the previous use-case, we focus on one-sided approaches, as used by FaRM and Pilaf [34, 37].

Linked list processing can be decomposed into a *while* loop for traversing the list and an *if* condition for finding and returning the key. We describe the implementation of our offload in Fig. 6.12. The client provides the key x and address of the first node in the list N_0. A READ operation (R2) is then performed to read the contents of the first node and update the values for the return operation (R5). We also use a WRITE operation (R3) to prepare the CAS operation (R4) by inserting key x in its *old* field. As an optimization, this WRITE can be removed and, instead, x can be inserted directly by the RECV operation. This, however, will need to be done for every CAS to be executed and, as such, this approach is limited to smaller list sizes, since RECVs can only perform 16 scatters.

For this use-case, we introduce two offload variations. The first, referred to simply as RedN, uses the implementation in Fig. 6.12. The second uses an additional *break* statement between R4 and R5 to exit the loop in order to avoid executing any additional operations.

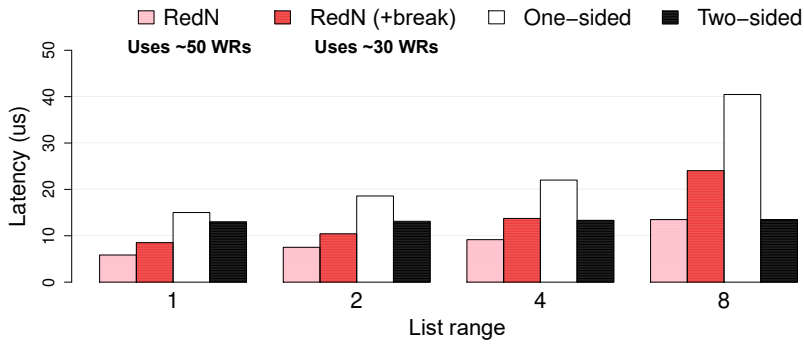


Figure 6.13: Average latency of walking linked lists.

6.4.3.1 Results

Fig. 6.13 shows the latency of one-sided and two-sided variants against RedN at various linked list ranges — where range represents the highest list element that the key can be randomly placed in. The size of the list itself is set to a constant value of 8. We setup the linked list to use key and value sizes of 48 bits and 64 bytes, respectively, and perform 100k list traversals for each system. The requested key is chosen at random for each RPC. In the variant labelled "RedN", we do not use *breaks* and assume that all 8 elements of the list need to be searched. RedN outperforms all baselines for all list ranges until 8 — providing up to a 2× improvement. *RedN (+break)* executes a break statement with each iteration and performs worse than RedN due to the extra overhead of checking the condition of the *break*. However, using a break statement increases the offload's overall efficiency since no unneeded iterations are executed after the key is found — using an average of 30 WRs across all experiments. Without breaks, RedN will need to execute all subsequent iterations even after the key-value pair is found/returned and it uses more than 65% more WRs. As such, while RedN is able to provide better latencies, using a break statement is more sensible for longer lists.

6.4.4 Use Case: Accelerating Memcached

Based on our earlier experience offloading remote data structure traversals, we set out to see: 1) how effective our aforementioned techniques are in a real system, and 2) what are the challenges in deploying it in such settings. Memcached is a key-value store that is often used as a caching service for large-scale storage services. We use a version of Memcached that employs cuckoo hashing [228]. Since Memcached does not natively support RDMA, we modify it with ∼700 LoC

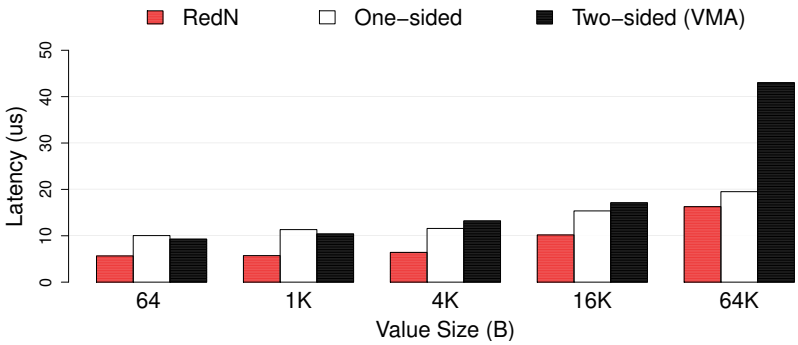


Figure 6.14: Memcached *get* latencies with different IO sizes.

to integrate RDMA capabilities, allowing the RNIC to register the hash table and storage object memory areas. We also modify the buckets, so that the addresses to the values are stored in big endian — to match the format used by the WR attributes. We then use RedN to offload Memcached's *get* requests to allow them to be serviced directly by the RNIC without CPU involvement. We compare our results to various configurations of Memcached.

To benchmark Memcached, we use the Memtier benchmark, configure it to use UDP (to reduce TCP overheads for the baselines), and issue 1 million *get* operations using different key-value sizes. To create a competitive baseline for two-sided approaches, we use Mellanox's VMA [229]—a kernel-bypass userspace TCP/IP stack that boosts the performance of sockets-based applications by intercepting their socket calls and using kernel-bypass to send/receive data. We configure VMA in polling-mode to optimize for latency. In addition, we also implement a one-sided approach, similar to the one introduced in section 6.4.2.

Fig. 6.14 shows the latency of *gets*. As we can see, RedN's offload for hash *gets* is up to 1.7× faster than one-sided and 2.6× faster than two-sided. Despite the latter being configured in polling-mode, VMA incurs extra overhead since it relies on a network stack to process packets. In addition, to adhere to the sockets API, VMA has to memcpy data from send and receive buffers, further inflating latencies—which is why it performs comparatively worse at higher value sizes.

6.4.5 Use Case: Performance Isolation

One of the benefits of exposing the latent turing power of RNICs is to enforce isolation among applications. CPU contention in multi-tenant and cloud settings can lead to arbitrary context switches, which can, in turn, inflate average and tail

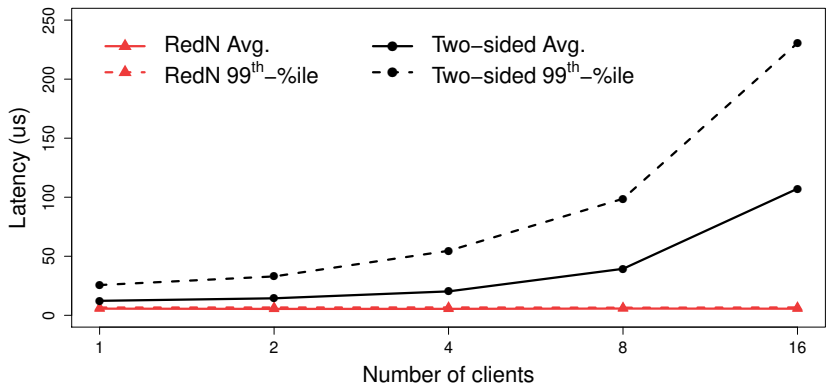


Figure 6.15: Memcached *get* latencies under hardware contention with varying numbers of writer-clients.

latencies. We explore such a scenario by sending background traffic to Memcached using one or more writer (clients). These writers generate *set* RPCs in a closed loop to load the Memcached service. At the same time, we use a single reader client to generate *get* operations. To stress CPU resources while minimizing lock contention, each reader/writer is assigned a distinct set of 10K keys, which they use to generate their queries. The keys within each set are accessed by the clients sequentially.

We can see in Fig. 6.15 that, as we increase the # of writers, both the average and 99^{th} percentile latencies for two-sided increase dramatically. For RedN, CPU contention has no impact on the performance of the RNIC and both the average and 99^{th} percentiles sit below 7 µs. At 16 writers, RedN's 99^{th} percentile latency is 35× lower than the baseline.

This indicates that RNIC offloads can also have other useful effects. Service providers may opt to offload high priority traffic for more predictable performance or allocate server resources to tenants to reduce contention.

6.4.6 Use Case: Failure Resiliency

We now consider server failures and how failure is affected by RNICs. Table 6.6 shows failure rates of server software and hardware components. NICs are much less likely to fail than software components—NIC annualized failure rate (AFR) is an order of magnitude lower. Even more importantly, NICs are partially decoupled from their hosts and can still access memory (or NVM) in the presence of an OS failure. This means that RNICs are capable of offloading key system functionality

Component	AFR	MTTF	Reliability
OS	41.9%	20,906	99%
DRAM	39.5%	22,177	99%
NIC	1.00%	876,000	99.99%
NVM	$< 1.00\%$	2 million	99.99%

Table 6.6: Failure rates of different server components [38, 230]. AFR means annualized failure rate, whereas MTTF stands for mean time to failure and is expressed in hours. RNICs can still access memory even in the presence of an OS failure.

that can allow servers to continue operating despite OS failures (albeit in a degraded state). To put this to the test, we conduct a fail-over experiment to explore how RedN can enhance a service's failure resiliency.

Process crashes. We look into how we can allow an RNIC to continue serving RPCs after a Memcached instance crashes. We find that this is not simple in practice. RNICs access many resources in application memory (*e.g.*, queues, doorbell records, *etc.*) that are required for functionality. If the process hosting these resources crashes, the memory belonging to these components will be automatically freed by the operating system resulting in termination of the RDMA program. To counteract this, we use [231] forks to create an empty hull parent for hosting RDMA resources and then allow Memcached to run as a child process. Linux systems do not free the resources of a crashed child until the parent also terminates. As such, keeping the RDMA resources tied to an empty process allows us to continue operating in spite of application failures. We run an experiment (timeline shown in Fig. 6.16) where we send *get* queries to a single instance of Memcached and then simply kill Memcached during the run. The OS detects the application's termination and immediately restarts it. Despite this, we can see that a vanilla Memcached instance will take at least 1 second to bootstrap, and 1.25 additional seconds to build its metadata and hashtables. With RedN, no service disruption is experienced and *get* queries continue to be issued without recovery time.

OS failure. We also programmatically induce a kernel panic using `sysctl`, freezing the system. This is a simpler case than process crashes, since we no longer have to worry about the OS freeing RDMA resources. For brevity, we do not show these results, but we experimentally verified that RedN offloads continue operating

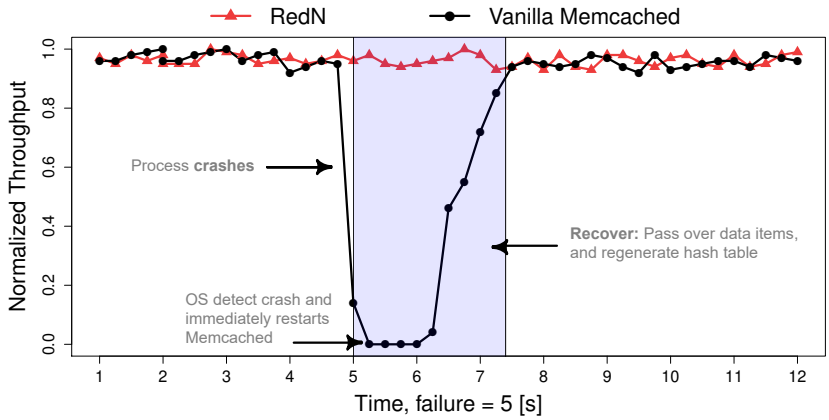


Figure 6.16: RedN can survive process crashes and continue serving RPCs via the RNIC without interruption.

in the presence of an OS crash.

6.5 Offloading Assise's IO using RedN

In Assise (Chapter 5), we presented a distributed file system model that leverages client-local NVRAM to provide large performance gains. However, one of the limitations of this model is that it also burdens the client machines' CPU, which now has to manage file system (meta)data. This introduces scalability challenges since the clients' CPU resources are stressed.

The RedN framework provides a solution to this problem. We can use our constructs to offload tasks performed on Assise's critical IO path to both improve latency and reduce CPU usage. For instance, to perform remote file or directory reads, Assise relies on a traditional two-sided RPC implementation that requires the server CPU to first search a hash table, and then return the data via an RDMA operation. These reads can be fully offloaded using RedN, as we have demonstrated in section 6.4.2. For remote writes, which are required to replicate data in Assise, their logic is also offloadable using RedN. Assise uses chain replication which simply copies data to the NVRAM write log on the replicas and does not require any conditional logic. As such, both the read and write paths can be fully offloaded. In doing so, we can reduce the CPU burden on the client machines, allowing for better scalability while simultaneously accelerating Assise's IO.

6.6 Discussion

Client scalability. RedN requires servers to manage at least two WQs per client, which is not higher than other RDMA systems. RedN can still introduce scalability challenges with thousands of clients since RNIC cache is limited. However, Mellanox's dynamically-connected (DC) transport service [232], which allows unused connections to be recycled, can circumvent many such scalability limits.

Intel RNICs. Next-generation Intel RNICs are expected to support atomic verbs, such as CAS—which RedN uses to implement conditionals. To control when WRs can be fetched by the NIC, Intel uses a validity bit in each WR header. This bit can be dynamically modified via an RDMA operation to mimic ENABLE. However, there is no equivalent for the WAIT primitive, meaning that clients cannot trigger a pre-posted chain. One possible workaround for this is to use another PCIe device on the server to issue a doorbell to the RNIC, allowing the WR chain to be triggered. We leave the exploration of such techniques as future work.

Insights for next-generation RNICs. Our experience with RedN has shown that keeping WRs in server memory (to allow them to be modified by other RDMA verbs) is a key bottleneck. If the NIC's cache was made directly accessible via RDMA, WRs can be pre-fetched in advance and unnecessary PCIe round-trips on the critical path can be avoided. We hope future RNICs will support such features.

6.7 Conclusion

We show that, in spite of appearances, commodity RDMA NICs are Turing-complete and capable of performing complex offloads without *any* hardware modifications. We take this insight and explore the feasibility and performance of these offloads. We find that, using a commodity RNIC, we can achieve up to 2.6× and 35× speed-up versus state-of-the-art RDMA approaches, for key-value get operations under uncontended and contended settings, respectively, while allowing applications to gain failure resiliency to OS and process crashes. We believe that this work opens the door for a wide variety of innovations in RNIC offloading which, in turn, can help guide the evolution of the RDMA standard.
RedN is available at `https://redn.io`.

Chapter 7

Required Reflections

Ethics & sustainability is one of the main focuses for research at KTH. Distributed storage systems form the backbone of many critical services, and improving their speed and predictability can offer many benefits on a societal level. Before concluding our thesis, it is useful to discuss its potential impact on society and any ethical considerations.

7.1 Sustainability

Sustainable development is defined in the 1987 Bruntland Commission Report [233] by the United Nations Educational, Scientific and Cultural Organization (UNESCO) as:

"Development that meets the needs of the present without compromising the ability of future generations to meet their own needs."

With regards to the societal and economical aspects of sustainability, the work in this thesis focuses on accelerating distributed storage systems, which are critical to services in many different areas (*e.g.*, traffic management, teaching, research, *etc.*). Improving the performance of such services translates into increased end-user satisfaction and can improve overall productivity. Our individual contributions to this end goal are as follows:

1. Our first contribution (Rein in Chapter 3) proposes scheduling techniques that leverage certain properties of the request structure to reduce aggregate median and tail latencies by up to 1.5×, and 1.9×, respectively.

2. Our second contribution (Assise in Chapter 5) redesigns distributed file systems to exploit the low-latency and persistence properties of NVM and combines them with high-speed RDMA networks. Comparing against state-of-the-art file systems, our results show that Assise improves write latency up to 22×, and throughput up to 56×.
3. Our third contribution (RedN in Chapter 6) shows that RNICs are Turing-complete, effectively providing us with an additional compute resource. We show that offloading common key-value store operations to RNICs can provide a speed-up of up to 2.6× and 35× in contended and non-contended settings, respectively. This helps reduce overall CPU usage, which can lead to a reduction in energy consumption as a consequence. While we cannot foresee the potential impact on energy-savings, RedN provides the tools necessary to propel future research in the area of RDMA offloads and their efficiency.

7.2 Ethical considerations

All collected traces presented in our studies do not exploit any private information about individuals or otherwise deal with any sensitive data. For instance, our SoundCloud real workload trace is anonymized and does not contain any user information. In our Amazon *AWS* study, we do not gather information about cloud tenants or the underlying network infrastructure, but simply report the latencies gathered through ping measurements. Our results show that TCP and UDP ports can be assigned different paths and that tenants may be incentivized to pick the ports with the lowest observed latencies to improve their application performance. This can of course work against the cloud provider's traffic engineering goals and exacerbate unfairness among tenants. Such techniques for choosing the best-performing ports are already well-known [139], and our work does not provide any software tools or artifacts to facilitate their usage.

Chapter 8

Conclusion & Future Work

Rethinking distributed storage system designs is now warranted given the recent increase in heterogeneity across workloads, networks, and hardware components. This dissertation finds that these heterogeneities present both challenges and opportunities. It then goes on to provide blueprints for next-generation distributed storage systems that provide fast and predictable performance amidst heterogeneity.

In Rein, I showed that, in production workloads, request sizes are skewed. I then demonstrated that this property can be exploited to schedule requests efficiently and improve performance predictability for key-value stores. I showed how these scheduling heuristics can be used in a real distributed key-value store.

Next, I performed a measurements study in one of the largest cloud provider networks which focused on latency variations across regions. This study showed that these variations occur sporadically and at very short time-scales (order of tens of seconds). I then outline the implications of this on geo-distributed systems and how this information can augment Rein's scheduling heuristics.

In Assise, I developed a low-latency distributed file system that leverages state-of-the-art storage devices— namely, NVRAM—and manages them efficiently with other types of storage. This work identifies new bottlenecks for NVRAM-based distributed file systems and makes several key design decisions to provide low latency and high scalability. It is one of the first few works in the area that evaluates the performance of distributed storage systems using real NVRAM hardware.

Lastly, in RedN, I provide a method to turn commodity RNICs into the equivalent of general-purpose processors. This enables the execution of complex logic on the NIC, thereby allowing application developers to leverage its computational resources to accelerate storage systems. This work is the first to prove that RDMA is Turing complete, opening a wide set of possibilities for RNIC offloads.

8.1 Future work

This thesis focused on how to make distributed storage systems faster and more predictable amidst heterogeneity. However, we only scratched the surface.

In part I, we proposed ways to improve Rein's scheduling heuristics by leveraging information about the network latency. However, we have not yet prototyped or evaluated these proposed improvements in practice and leave this as future work.

In the Assise project, we evaluated the efficacy of our proposed file system model using different types of benchmarks and applications. However, most of our experiments were confined to a testbed of five nodes — primarily due to unavailability of larger NVRAM testbeds at the time this work was undertaken. It would be interesting to study how well Assise scales to a larger number of nodes. In addition, Assise has been evaluated using only one type of NVRAM —namely Intel Optane DC PMM DIMMs— and an evaluation of its performance with upcoming NVRAM technologies [19, 20] may be warranted. Moreover, one could also explore how Assise can be accelerated with programmable NICs and switches (a discussion on how RedN may be used for this purpose can be found in §6.5). We already took a first step in a spin-off project called LineFS [234], where we looked into how Assise's performance and scalability can be further improved using SmartNICs.

Our offload framework for RNICs—RedN—should help pave the way for researchers to test the limits of RDMA and explore different use-cases. Our work mainly focused on offloading hash table and linked list traversals, however, many other data structures (*e.g.*, B-Trees) and use-cases (*e.g.*, complex database transactions) have not been explored. Moreover, RedN currently only introduces *if* and *while* constructs as building blocks for RDMA offloads. It does not automate the process of producing RDMA code from high-level languages. Building a C compiler for RedN can make creating RDMA programs more straightforward and reduce the burden on the programmer. One could also extend RDMA offloads to include other ASIC types (e.g. GPUs), allowing for more elaborate chains of offloads. RedN provides unprecedented control over RNICs and opens the door for further research into advanced RDMA offloads which can ultimately help guide the evolution of the RDMA standard.

In the following sub-sections, I discuss several follow-up works that can be potentially pursued.

8.1.1 Implementing Locks in RDMA

Lock managers are essential to the operation of many distributed systems — including key-value stores [34, 35], OLTP databases [224, 225], and distributed file systems [12]. Several proposals [220] have been made to speed-up locking using RDMA primitives. However, given the limited RDMA API, these approaches only partially offload lock acquisitions to the NIC. In more complex scenarios (e.g. lock contention), these partial offload solutions need to fallback to either server CPU or require additional network roundtrips. Moreover, in transactional systems, such offloads do not provide ways to chain together lock acquisition with transaction execution. In other words, several roundtrips are required to acquire the lock first, and then execute the transaction, even if the lock manager is co-located with the database. To workaround these limitations, it would be interesting to explore how to use the RedN framework to realize the first complete locking implementation on RDMA NICs. Lock acquisition can be implemented using the conditional constructs offered by RedN, allowing NICs to become a full-fledged lock managers. Moreover, since RedN can also offload database operations, locking can be combined with these operations to implement complex transactions inside the NIC. In doing so, we avoid extra network roundtrips, which are typically required if lock acquisition is offloaded separately. Beyond that, it would be interesting to explore if more rich features can be supported. For example, can our offloads prevent lock starvation? This occurs if lock requests are blocked for longer periods, which can impact tail latency. Similarly, one could also explore whether we can provide more flexible policies (e.g. priority classes for requests) to support different tenant needs. Furthermore, it might be interesting to see whether we can offload the lease mechanism in Assise, given that it is functionally-similar to locks.

8.1.2 Accelerating Distributed Deep Learning

Deep learning is a special type of machine learning that imitates the ways humans gain certain types of knowledge, and has been applied successful in several domains, including computer vision, speech recognition, and natural language processing. Distributed deep learning is becoming a necessity to deal with models of increasing complexity. These systems commonly perform their computations on simple data structures called tensors, which are essentially multi-dimensional arrays. These tensors are typically synchronized between servers using RPCs, which necessitate extra memory copies to/from RPC communication buffers. RDMA has recently been used in this domain [235] since it allows zero-copy transfers to remote memory using one-sided verbs such as READ and WRITE. Current RDMA-based distributed deep learning systems, however, while more

efficient, still require CPU involvement for communication. This is because tensor sizes are not necessarily known a priori, and servers need to first fetch their metadata (specifically, their size) before copying them via RDMA. This does not scale well with the large bandwidth of RDMA and requires multiple roundtrips. To overcome such limitations, one could use our offload framework—RedN—to perform both the metadata check and the data copy in one step, all within the NIC. These two actions should be offloadable using the tools provided by our framework. In doing so, server CPUs will no longer be involved in RDMA communication – allowing for increased scalability. Moreover, this technique can also be used to copy tensors directly to GPU memory. This is possible via the GPUDirect RDMA technology, which allows RDMA NICs to access GPU memory without having to go through server memory, further improving efficiency. It would be interesting to see how these optimizations can reduce the runtime of distributed deep learning systems.

8.1.3 Enhancing RDMA Security

As mentioned in the previous text, RDMA technology is seeing increased use in cloud datacenters. This is primarily due to the benefits afforded by its one-sided operations, which allow clients to bypass server CPU and directly read/write to server memory. However, one of the obstacles to wider RDMA adoption is its security aspects, and recent findings have shown that many vulnerabilities exist [236]. One-sided operations in specific require direct read and/or write memory access. This can compromise security if clients are buggy and/or malicious. For example, FaRM-KV [34] – a well-known RDMA-based key-value store – allows clients to write messages directly to shared RPC buffers. This requires clients to behave correctly, as they can otherwise overwrite or modify other clients' RPCs. I believe RedN can help change the landscape of RDMA security. For one, RedN allows us to use more secure operations like RDMA SEND/RECV (which do not require memory privileges) while still bypassing server CPU. RedN makes this possible by chaining together operations. For instance, instead of performing an RDMA READ, which allows clients to directly read data from the server, RedN can emulate this using two RDMA SEND operations, one for sending the request with the memory address, and the other to return the response. This means that servers no longer need to provide direct memory permissions to clients. Moreover, given the flexibility of RedN, servers can perform complex operations like input checks, to make sure that clients are not trying to read unauthorized memory addresses. It would be meaningful to explore how such techniques can be used to overcome security vulnerabilities in RDMA.

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," in *SIGCOMM*, 2010.

[2] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," in *NSDI*, 2012.

[3] G. Kumar, G. Ananthanarayanan, S. Ratnasamy, and I. Stoica, "Hold 'em or Fold 'em? Aggregation Queries under Performance Variations," in *EuroSys*, 2016.

[4] 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 *NSDI*, 2013.

[5] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha, "Sharing the Data Center Network," in *NSDI*, 2011.

[6] W. Reda, M. Canini, L. Suresh, D. Kostić, and S. Braithwaite, "Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling," in *Twelfth European Conference on Computer Systems*. ACM, 2017. doi: 10.1145/3064176.3064209 p. 95–110.

[7] W. Reda, K. Bogdanov, A. Milolidakis, H. Ghasemirahni, M. Chiesa, G. Q. Maguire Jr, and D. Kostić, "Path Persistence in the Cloud: A Study of the Effects of Inter-Region Traffic Engineering in a Large Cloud Provider's Network," *ACM SIGCOMM Computer Communication Review*, p. 11–23, 2020. doi: 10.1145/3402413.3402416

[8] "Intel Optane DC Persistent Memory Module 128GB," Oct. 2020, google Shopping search. Lowest non-discount price.

[9] "Intel SSD DC P4610 1.6TB," Apr. 2019, google Shopping search. Lowest non-discount price.

[10] O. Cardona, "Towards Hyperscale High Performance Computing with RDMA," 2019, https://pc.nanog.org/static/published/meetings/NANOG76/1999/20190612_Cardona_Towards_Hyperscale_High_v1.pdf.

[11] W. Reda, M. Canini, D. Kostić, and S. Peter, "RDMA is Turing complete, we just did not know it yet!" in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[12] T. E. Anderson, M. Canini, J. Kim, D. Kostić, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel, "Assise: Performance and Availability via Client-local NVM in a Distributed File System," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020. doi: 10.5555/3488766.3488823 pp. 1011–1027.

[13] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, 2007, pp. 29–42.

[14] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *SIGCOMM*, 2008.

[15] A. Pathak, M. Zhang, Y. C. Hu, R. Mahajan, and D. A. Maltz, "Latency inflation with MPLS-based traffic engineering," in *IMC*, 2011.

[16] "Mellanox BlueField," https://www.mellanox.com/products/bluefield-overview.

[17] M. Le Gallo and A. Sebastian, "An overview of phase-change memory device physics," *Journal of Physics D: Applied Physics*, vol. 53, no. 21, p. 213002, 2020.

[18] "Intel Optane DC persistent memory," Mar. 2019, http://www.intel.com/optanedcpersistentmemory.

[19] B. Hoberman, "The emergence of practical mram," *Crocus Technologies*, 2009.

[20] M. LaPedus, "Tower invests in Crocus, tips MRAM foundry deal," Jun 2009, https://www.eetimes.com/tower-invests-in-crocus-tips-mram-foundry-deal/ Accessed 9-January-2022.

[21] ZDNet, "Baidu swaps DRAM for Optane to power in-memory database," Aug. 2019, https://www.zdnet.com/article/baidu-swaps-dram-for-optane-to-power-in-memory-database/.

[22] ——, "Google cloud taps new Intel memory module for SAP HANA workloads," Jul. 2018, https://www.zdnet.com/article/google-cloud-taps-new-intel-memory-module-for-sap-hana-workloads/.

[23] InsideHPC, "Intel Optane DC persistent memory comes to Oracle Exadata X8M," Sep. 2019, https://insidehpc.com/2019/09/intel-optane-dc-persistent-memory-comes-to-oracle-exadata-x8m/.

[24] "Rdma consortium," http://www.rdmaconsortium.org/.

[25] "ConnectX series," https://www.mellanox.com/products/ethernet/connectx-smartnic.

[26] "NetFPGA platform," https://netfpga.org/.

[27] "High-level synthesis compiler - intel® hls compiler," https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[28] F. Christensen, "Challenges in using hls for fpga design," Apr 2019, https://semiengineering.com/challenges-in-using-hls-for-fpga-design/ Accessed 12-Jan-2022.

[29] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "Kv-direct: high-performance in-memory key-value store with programmable nic," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 137–152.

[30] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 36–51.

[31] "Intel Ethernet 800 Series Network Adapters," https://www.intel.com/content/www/us/en/products/docs/network-io/ethernet/network-adapters/ethernet-800-series-network-adapters/e810-cqda1-100gbe-brief.html.

[32] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso, "StRoM: Smart Remote Memory," *Proc. of EuroSys. ACM*, 2020.

[33] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. Marathe, and I. Zablotchi, "The Impact of RDMA on Agreement," *arXiv preprint arXiv:1905.12143*, 2019.

[34] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[35] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA efficiently for key-value services," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 295–306.

[36] M. Kazhamiaka, B. Memon, C. Kankanamge, S. Sahu, S. Rizvi, B. Wong, and K. Daudjee, "Sift: resource-efficient consensus with RDMA," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019, pp. 260–271.

[37] C. Mitchell, Y. Geng, and J. Li, "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 103–114.

[38] M. Poke and T. Hoefler, "Dare: High-performance State Machine Replication on RDMA Networks," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 107–118.

[39] J. Dean and L. A. Barroso, "The Tail At Scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013.

[40] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, 2009, pp. 202–208.

[41] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan, "Speeding up Distributed Request-Response Workflows," in *SIGCOMM*, 2013.

[42] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection," in *NSDI*, 2015.

[43] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, "Low Latency via Redundancy," in *CoNEXT*, 2013.

[44] A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA: Proportional Allocation of Resources for Distributed Storage Access," in *FAST*, 2009.

[45] D. Shue, M. J. Freedman, and A. Shaikh, "Performance Isolation and Fairness for Multi-tenant Cloud Storage," in *OSDI*, 2012.

[46] Y. Lu, Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. Greenberg, "Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services," *Performance Evaluation*, vol. 68, no. 11, pp. 1056–1071, 2011.

[47] M. Mitzenmacher, "The Power of Two Choices in Randomized Load Balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 10, Oct. 2001.

[48] H. Li, Y. Zhang, D. Li, Z. Zhang, S. Liu, P. Huang, Z. Qin, K. Chen, and Y. Xiong, "Ursa: Hybrid block storage for cloud-scale virtual disks," in *Fourteenth EuroSys Conference 2019*, ser. EuroSys '19, 2019. doi: 10.1145/3302424.3303967. ISBN 978-1-4503-6281-8 pp. 15:1–15:17.

[49] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006. ISBN 1-931971-47-1 pp. 307–320.

[50] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: An RDMA-enabled Distributed Persistent memory file system," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 773–785.

[51] J. Yang, J. Izraelevitz, and S. Swanson, "Orion: A distributed file system for non-volatile main memory and RDMA-capable networks," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 221–234.

[52] D. Hitz, J. Lau, and M. Malcolm, "File system design for an nfs file server appliance," in *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, ser. WTEC'94. Berkeley, CA, USA: USENIX Association, 1994, pp. 19–19.

[53] S. Watanabe, *Solaris 10 ZFS Essentials*. Prentice Hall, 2009.

[54] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang, "Serverless network file systems," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995. doi: 10.1145/224056.224066. ISBN 0-89791-715-4 pp. 109–126.

[55] T. Haynes and D. Noveck, "Network file system (NFS) version 4 protocol," Mar. 2015, https://tools.ietf.org/html/rfc7530.

[56] "Amazon Elastic Block Store (EBS)," https://aws.amazon.com/ebs/.

[57] "Amazon S3," https://aws.amazon.com/s3/.

[58] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees," in *26th ACM Symposium on Operating Systems Principles*, ser. SOSP '17, 2017.

[59] "Accelio," Aug. 2018, https://github.com/accelio/accelio.

[60] N. S. Islam, M. Wasi-ur Rahman, X. Lu, and D. K. Panda, "High performance design for hdfs with byte-addressability of nvm and rdma," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016. doi: 10.1145/2925426.2926290. ISBN 978-1-4503-4361-9 pp. 8:1–8:14.

[61] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui, "APUS: Fast and Scalable Paxos on RDMA," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 94–107.

[62] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto smartNICs using iPipe," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 318–333.

[63] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein, "NICA: An Infrastructure for Inline Acceleration of Network Applications," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 345–362.

[64] 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 *SOSP*, 2007.

[65] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, and L. Puzar, "TAO: How Facebook Serves the Social Graph," in *SIGMOD*, 2012.

[66] "Apache Cassandra," http://cassandra.apache.org/.

[67] "Elasticsearch," https://www.elastic.co/products/elasticsearch.

[68] T. A. Roemer, "A Note on the Complexity of the Concurrent Open Shop Problem," *J. of Scheduling*, vol. 9, no. 4, pp. 389–396, Aug. 2006. doi: 10.1007/s10951-006-7042-y

[69] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized Task-Aware Scheduling for Data Center Networks," in *SIGCOMM*, 2014.

[70] A. Wierman and B. Zwart, "Is Tail-Optimal Scheduling Possible?" *Operations Research*, vol. 60, no. 5, pp. 1249–1257, Sep. 2012.

[71] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *OSDI*, 2008.

[72] "Redis," http://redis.io/.

[73] "Memcached," https://www.memcached.org/.

[74] "MongoDB," https://www.mongodb.com/.

[75] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store," in *SIGMETRICS*, 2012.

[76] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *SoCC*, 2010.

[77] D. Chakrabarty, Y. Zhou, and R. Lukose, "Budget Constrained Bidding in Keyword Auctions and Online Knapsack Problems," in *WINE*, 2008.

[78] Z. Wu, C. Yu, and H. V. Madhyastha, "CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services," in *NSDI*, 2015.

[79] R. G. Christopher Stewart, Aniket Chakrabarti, "Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs," in *ICAC*, 2013.

[80] H. T. Vo, C. Chen, and B. C. Ooi, "Towards Elastic Transactional Cloud Storage with Range Query Support," *Proc. VLDB Endow.*, vol. 3, no. 1-2, Sep. 2010.

[81] S. Das, D. Agrawal, and A. El Abbadi, "G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud," in *SoCC*, 2010.

[82] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues, "AUTOPLACER: Scalable Self-Tuning Data Placement in Distributed Key-value Stores," in *ICAC*, 2013.

[83] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "PriorityMeister: Tail Latency QoS for Shared Networked Storage," in *SoCC*, 2014.

[84] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren, "Stout: An Adaptive Interface to Scalable Cloud Storage," in *ATC*, 2010.

[85] M. E. Haque, Y. hun Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, "Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services," in *ASPLOS*, 2015.

[86] J. Li, K. Agrawal, S. Elnikety, Y. He, I.-T. A. Lee, C. Lu, and K. S. McKinley, "Work Stealing for Interactive Services to Meet Target Latency," in *PPoPP*, 2016.

[87] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient Coflow Scheduling with Varys," in *SIGCOMM*, 2014.

[88] M. Chowdhury and I. Stoica, "Efficient Coflow Scheduling Without Prior Knowledge," in *SIGCOMM*, 2015.

[89] L. Flatto and S. Hahn, "Two Parallel Queues Created by Arrivals with Two Demands I," *SIAM Journal on Applied Mathematics*, vol. 44, no. 5, pp. 1041–1053, 1984.

[90] E. Varki, A. Merchant, and H. Chen, "The M/M/1 Fork-Join Queue with Variable Sub-Tasks," 2002, http://www.cs.unh.edu/~varki/publication/2002-nov-open.pdf.

[91] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data," in *EuroSys*, 2013.

[92] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in Map-reduce Clusters Using Mantri," in *OSDI*, 2010.

[93] N. Bansal and M. Harchol-Balter, "Analysis of SRPT Scheduling: Investigating Unfairness," in *SIGMETRICS*, 2001.

[94] O. Boxma and B. Zwart, "Tails in Scheduling," *ACM SIGMETRICS Performance Evaluation Review*, vol. 34, no. 4, pp. 13–20, 2007.

[95] H. Rihani, P. Sanders, and R. Dementiev, "MultiQueues: Simpler, Faster, and Better Relaxed Concurrent Priority Queues," *CoRR*, vol. abs/1411.1209, 2014.

[96] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The SprayList: A Scalable Relaxed Priority Queue," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 11–20, 2015.

[97] M. Wimmer, F. Versaci, J. L. Träff, D. Cederman, and P. Tsigas, "Data Structures for Task-based Priority Scheduling," in *PPoPP*, 2014.

[98] M. Welsh, D. Culler, and E. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," in *SOSP*, 2001.

[99] "Akka," http://akka.io/.

[100] R. A. Cohen, "An Introduction to PROC LOESS for Local Regression," in *SUGI*, 1999.

[101] N. Shavit and I. Lotan, "Skiplist-Based Concurrent Priority Queues," in *IPDPS*, 2000.

[102] M. M. Michael and M. L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," in *PODC*, 1996.

[103] H. Sundell and P. Tsigas, "Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609–627, 2005.

[104] C. Kalantzis, "Eventual Consistency != Hopeful Consistency, talk at Cassandra Summit," 2013, https://www.youtube.com/watch?v=A6qzx_HE3EU.

[105] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage," in *NSDI*, 2014.

[106] "ScyllaDB," http://scylladb.com/.

[107] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable Low Latency for Data Center Applications," in *SOCC*. ACM, 2012.

[108] J. Moy, "Ospf version 2," *Internet Request for Comments*, vol. RFC 2328 (INTERNET STANDARD), apr 1998, http://www.rfc-editor.org/rfc/rfc2328.txt. Updated by RFCs 5709, 6549, 6845, 6860, 7474.

[109] B. Fortz and M. Thorup, "Optimizing OSPF/IS-IS Weights in a Changing World," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 4, pp. 756–767, 2002.

[110] "Multiprotocol label switching working group," June 2009, http://www.ietf.org/html.charters/mpls-charter.html.

[111] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," in *SIGCOMM*, 2007.

[112] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally-Deployed Software Defined WAN," in *SIGCOMM*, 2013.

[113] Bing, "Microsoft Bing," https://www.bing.com Accessed 1-Oct-2018.

[114] V. Jalaparti, I. Bliznets, S. Kandula, B. Lucier, and I. Menache, "Dynamic pricing and traffic engineering for timely inter-datacenter transfers," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 73–86.

[115] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 15–26, Aug. 2013. doi: 10.1145/2534169.2486012

[116] A. Takács, A. Császár, J. Bíró, R. Szabó, and T. Henk, "Path integrity aware traffic engineering [telecom traffic]," in *Global Telecommunications Conference, 2004. GLOBECOM'04. IEEE*, vol. 2. IEEE, 2004, pp. 692–696.

[117] M. Zhang, B. Karp, S. Floyd, and L. Peterson, "RR-TCP: a reordering-robust TCP with DSACK," in *11th IEEE International Conference on Network Protocols, 2003. Proceedings.* IEEE, 2003, pp. 95–106.

[118] R. Ludwig and R. H. Katz, "The Eifel algorithm: making TCP robust against spurious retransmissions," *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 1, pp. 30–36, 2000.

[119] E. Blanton and M. Allman, "On making TCP more robust to packet reordering," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 1, pp. 20–30, 2002.

[120] M. Laor and L. Gendel, "The effect of packet reordering in a backbone link on application throughput," *IEEE network*, vol. 16, no. 5, pp. 28–36, 2002.

[121] K. Bogdanov, M. Peón-Quirós, G. Q. Maguire Jr, and D. Kostić, "The nearest replica can be farther than you think," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 16–29.

[122] J. Hamilton, "Aws re:invent 2016: Tuesday night live," https://www.youtube.com/watch?v=AyOAjFNPAbA Accessed 17-Jun-2018.

[123] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon *et al.*, "TAO: How Facebook Serves the Social Graph," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 791–792.

[124] T. Roughgarden and É. Tardos, "How bad is selfish routing?" *Journal of the ACM (JACM)*, vol. 49, no. 2, pp. 236–259, 2002.

[125] Zhiruo Cao, Zheng Wang, and E. Zegura, "Performance of hashing-based schemes for internet load balancing," in *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 1, March 2000. doi: 10.1109/INFCOM.2000.832203. ISSN 0743-166X pp. 332–341.

[126] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm," *Internet Request for Comments*, vol. RFC 2992 (Informational), Nov. 2000. doi: 10.17487/RFC2992 http://www.rfc-editor.org/rfc/rfc2992.txt.

[127] C. L. Hedrick, "Routing Information Protocol," *Internet Request for Comments*, vol. RFC 1058 (Historic), Jun. 1988. doi: 10.17487/RFC1058 http://www.rfc-editor.org/rfc/rfc1058.txt.

[128] D. Dhody, U. Palle, R. Singh, and R. Gandhi, "PCEP Extensions for MPLS-TE LSP Automatic Bandwidth Adjustment with Stateful PCE," *Internet Drafts*, vol. PCE Working Group, Nov. 2018, https://tools.ietf.org/html/draft-ietf-pce-stateful-pce-auto-bandwidth-08.

[129] B. Augustin, T. Friedman, and R. Teixeira, "Measuring load-balanced paths in the internet," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 2007, pp. 149–160.

[130] Í. Cunha, R. Teixeira, and C. Diot, "Measuring and characterizing end-to-end route dynamics in the presence of load balancing." in *PAM*, vol. 11. Springer, 2011, pp. 235–244.

[131] H. Pucha, Y. Zhang, Z. M. Mao, and Y. C. Hu, "Understanding Network Delay Changes Caused by Routing Events," *ACM SIGMETRICS Performance Evaluation Review - SIGMETRICS '07 Conference Proceedings*, vol. 35, no. 1, pp. 73–84, 2007.

[132] V. Arun and H. Balakrishnan, "Copa: Practical delay-based congestion control for the internet," in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. USENIX Association, 2018.

[133] K. L. Bogdanov, W. Reda, G. Q. Maguire Jr, D. Kostić, and M. Canini, "Fast and accurate load balancing for geo-distributed storage systems," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2018, pp. 386–400.

[134] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "iPlane: An Information Plane for Distributed Services," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 367–380.

[135] B. Augustin, T. Friedman, and R. Teixeira, "Multipath tracing with paris traceroute," in *End-to-End Monitoring Techniques and Services, 2007. E2EMON'07. Workshop on*. IEEE, 2007, pp. 1–8.

[136] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira, "Avoiding traceroute anomalies with paris traceroute," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM, 2006, pp. 153–158.

[137] C. Pelsser, L. Cittadini, S. Vissicchio, and R. Bush, "From Paris to Tokyo: on the suitability of ping to measure latency," in *IMC*, 2013.

[138] I. Cunha, R. Teixeira, D. Veitch, and C. Diot, "Predicting and tracking internet path changes," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 122–133, 2011.

[139] G. Detal, C. Paasch, S. van der Linden, P. Mérindol, G. Avoine, and O. Bonaventure, "Revisiting flow-based load balancing: Stateless path selection in data center networks," *Computer Networks*, vol. 57, no. 5, pp. 1204–1216, 2013.

[140] B. Hesmans, G. Detal, S. Barré, R. Bauduin, and O. Bonaventure, "SMAPP: Towards smart multipath TCP-enabled applications," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '15. New York, NY, USA: ACM, 2015. doi: 10.1145/2716281.2836113. ISBN 978-1-4503-3412-9 pp. 28:1–28:7.

[141] D. Meyer, "AWS Remains Dominant Player in Growing Cloud Market, SRG Reports," https://www.sdxcentral.com/articles/news/aws-remains-dominant-player-in-growing-cloud-market-srg-reports/.

[142] A. AWS, "Setting the Time for Your Linux Instance," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/set-time.html Accessed 2-Feb-2019.

[143] "All measurement data used in this paper can be found via this link," goo.gl/25BKte.

[144] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 123–137.

[145] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching," in *14th USENIX Symposium on Networked Systems Design and Implementation NSDI'17)*, 2017, pp. 407–420.

[146] "Amazon Elastic File System (EFS)," https://aws.amazon.com/efs/.

[147] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *19th ACM Symposium on Operating Systems Principles*, ser. SOSP '03, 2003. doi: 10.1145/945445.945450. ISBN 1-58113-757-5 pp. 29–43.

[148] M. Stonebraker, "Operating system support for database management," *Commun. ACM*, vol. 24, no. 7, pp. 412–418, Jul. 1981. doi: 10.1145/358699.358703

[149] *NVM Programming Model (NPM) Version 1.2*, SNIA, Jun. 2017.

[150] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *13th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '18, 2018. ISBN 978-1-939133-08-3 pp. 461–476.

[151] S. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "RECIPE: Reusing concurrent in-memory indexes for persistent memory," in *27th ACM Symposium on Operating Systems Principles*, 2019.

[152] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin, "Robustness in the Salus scalable block store," in *10th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI '13, 2013, pp. 357–370.

[153] C. Gray and D. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *12th ACM Symposium on Operating Systems Principles*, ser. SOSP '89, 1989. doi: 10.1145/74850.74870. ISBN 0-89791-338-8 pp. 202–210.

[154] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *6th Symposium on Operating Systems Design and Implementation*, ser. OSDI'04, 2004.

[155] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed OS for hardware resource disaggregation," in *13th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '18, 2018. ISBN 978-1-939133-08-3 pp. 69–87.

[156] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017. doi: 10.1145/3132747.3132770. ISBN 978-1-4503-5085-3 pp. 460–477.

[157] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis, "File systems unfit as distributed storage backends: Lessons from 10 years of Ceph evolution," in *27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, 2019.

[158] "DDR4-3200 DRAM ECC Registered 128GB," Oct. 2020, google Shopping search. Lowest non-discount price.

[159] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10, 2010, pp. 61–74.

[160] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann, and T. Engbersen, "Failure analysis of virtual and physical machines: Patterns, causes and characteristics," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '14, 2014. doi: 10.1109/DSN.2014.18. ISBN 978-1-4799-2233-8 pp. 1–12.

[161] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. Morgan Kaufmann, 2017. ISBN 0128119055, 9780128119051

[162] C. Metz, "The epic story of Dropbox's exodus from the Amazon cloud empire," Mar. 2016, https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/.

[163] "Apache Crail," http://crail.apache.org/.

[164] J. Mickens, E. B. Nightingale, J. Elson, K. Nareddy, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, and O. Khan, "Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications," in *11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI '14, 2014. ISBN 978-1-931971-09-6 pp. 257–273.

[165] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015. doi: 10.1145/2806887

[166] S.-Y. Tsai, Y. Shan, and Y. Zhang, "Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores," in *2020 USENIX Annual Technical Conference*, ser. USENIX ATC '20, 2020. ISBN 978-1-939133-14-4 pp. 33–48.

[167] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *ACM Symposium on Cloud Computing*, ser. SOCC '14, 2014. doi: 10.1145/2670979.2670985. ISBN 978-1-4503-3252-1 pp. 6:1–6:15.

[168] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, p. 321–359, Nov. 1989. doi: 10.1145/75104.75105

[169] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. West, "Scale and performance in a distributed file system," in *Eleventh ACM Symposium on Operating Systems Principles*, ser. SOSP '87, 1987. doi: 10.1145/41457.37500. ISBN 089791242X p. 1–2.

[170] "The Sprite Operating System," https://www2.eecs.berkeley.edu/Research/Projects/CS/sprite/sprite.html, Aug. 2017.

[171] M. Burrows, "Efficient data sharing," Ph.D. dissertation, University of Cambridge, UK, 1988.

[172] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13.

New York, NY, USA: ACM, 2013. doi: 10.1145/2517349.2522726. ISBN 978-1-4503-2388-8 pp. 228–243.

[173] "Apache ZooKeeper," https://zookeeper.apache.org, Aug. 2017.

[174] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "All file systems are not created equal: On the complexity of crafting crash-consistent applications," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014. ISBN 978-1-931971-16-4 pp. 433–448.

[175] "NVM Express over Fabrics 1.1," 2019, https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf.

[176] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," in *24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, 2019. doi: 10.1145/3297858.3304077. ISBN 978-1-4503-6240-5 pp. 427–439.

[177] T. Le, J. Stern, and S. Briscoe, "Fast memcpy with SPDK and Intel I/OAT DMA engine," Apr. 2017, https://software.intel.com/en-us/articles/fast-memcpy-using-spdk-and-ioat-dma-engine.

[178] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen, "Performance and protection in the ZoFS user-space NVM file system," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 478–493.

[179] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, ser. FAST'16. Berkeley, CA, USA: USENIX Association, 2016. ISBN 978-1-931971-28-7 pp. 323–338.

[180] H. Qiu, X. Wang, T. Jin, Z. Qian, B. Ye, B. Tang, W. Li, and S. Lu, "Toward effective and fair RDMA resource sharing," in *2nd Asia-Pacific Workshop on Networking*, 2018, pp. 8–14.

[181] Y. Chen, Y. Lu, and J. Shu, "Scalable RDMA RPC on reliable connection with efficient resource sharing," in *14th EuroSys Conference 2019*, 2019, pp. 1–14.

[182] A. Rosenbaum and A. Margolin, "Dynamically-Connected Transport," 2018, https://www.openfabrics.org/images/2018workshop/presentations/303_ARosenbaum_DynamicallyConnectedTransport.pdf. Talk. 14th Annual Open Fabrics Alliance Workshop.

[183] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "SplitFS: Reducing software overhead in file systems for persistent memory," in *27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, 2019. doi: 10.1145/3341301.3359631. ISBN 9781450368735 p. 494–508.

[184] Mellanox, "Mellanox Introduces Revolutionary DPU based SmartNICs for Making Secure Cloud Possible," 2019, https://blog.mellanox.com/2019/08/mellanox-introduces-revolutionary-smartnics-for-making-secure-cloud-possible/.

[185] "Persistent memory programming," Aug. 2017, http://pmem.io/.

[186] "syscall_intercept," https://github.com/pmem/syscall_intercept.

[187] A. Baumann, P. Barham, P.-E. Dagand, T. 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 ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009. doi: 10.1145/1629575.1629579. ISBN 978-1-60558-752-3 pp. 29–44.

[188] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the Intel Optane DC Persistent Memory Module," Apr. 2019, https://arxiv.org/abs/1903.05714v2.

[189] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux Symposium*, vol. 2, Ottawa, ON, Canada, Jun. 2007.

[190] P. MacArthur and R. D. Russell, "A performance study to guide RDMA programming decisions," in *IEEE 14th International Conference on High Performance Computing and Communication and IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, 2012, pp. 778–785.

[191] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes, "Tailwind: Fast and atomic RDMA-based replication," in *2018 USENIX Annual Technical Conference*, ser. USENIX ATC '18, 2018. ISBN 9781931971447 pp. 851–863.

[192] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan, "Hyperloop: group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 297–312.

[193] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *USENIX ;login:*, vol. 41, no. 1, 2016.

[194] "Intel Memory Latency Checker," https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[195] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering," in *10th USENIX Conference on File and Storage Technologies*, ser. FAST'12, 2012.

[196] "Supporting filesystems in persistent memory," https://lwn.net/Articles/610174/, Sep. 2014.

[197] Ceph Documentation, "Differences from POSIX," http://docs.ceph.com/docs/master/cephfs/posix/.

[198] "Xfstests," 2019, https://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git/.

[199] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, "Crashmonkey and ACE: Systematically testing file-system crash consistency," *ACM Trans. Storage*, vol. 15, no. 2, Apr. 2019. doi: 10.1145/3320275

[200] "NFS - Xfstests," 2019, http://wiki.linux-nfs.org/wiki/index.php?title=Xfstests&oldid=5652.

[201] "Octopus - github repository," https://github.com/thustorage/octopus.

[202] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: Performance of user-space file systems," in *15th USENIX Conference on File and Storage Technologies*, ser. FAST'17, 2017. ISBN 9781931971362 pp. 59–72.

[203] J. Dean and S. Ghemawat, "LevelDB: A Fast Persistent Key-Value Store," https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html, 2011.

[204] J. Jiang, L. Zheng, J. Pu, X. Cheng, C. Zhao, M. R. Nutter, and J. D. Schaub, "Tencent sort," Tencent Corporation, Tech. Rep., 2016, http://sortbenchmark.org/TencentSort2016.pdf.

[205] "Sort benchmark home page," http://sortbenchmark.org/.

[206] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.

[207] F. Junqueira and B. Reed, *ZooKeeper: Distributed Process Coordination*, 1st ed. O'Reilly Media, Inc., 2013. ISBN 1449361307

[208] W. Venema, "Postfix project," http://www.postfix.org/.

[209] B. Klimt and Y. Yang, "The Enron corpus: A new dataset for email classification research," in *European Conference on Machine Learning*. Springer, 2004, pp. 217–226.

[210] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder, "Grapevine: An exercise in distributed computing," in *Eighth ACM Symposium on Operating Systems Principles*, ser. SOSP '81, 1981. doi: 10.1145/800216.806606. ISBN 0-89791-062-1 pp. 178–179.

[211] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana, "E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 363–378.

[212] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson, "Floem: A Programming System for NIC-Accelerated Network Applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 663–679.

[213] "Catapult," https://www.microsoft.com/en-us/research/project/project-catapult/.

[214] "Cavium-Xpliant," https://www.openswitch.net/cavium/.

[215] "Stingray," https://www.broadcom.com/products/ethernet-connectivity/smartnic.

[216] "Mellanox RDMA Aware Networks Programming User Manual," https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[217] "Agilio CX SmartNICs," https://www.netronome.com/products/agilio-cx/.

[218] "LiquidIO II SmartNICs," https://www.marvell.com/products/ethernet-adapters-and-controllers/liquidio-smart-nics/liquidio-ii-smart-nics.html.

[219] "Mellanox store," http://store.mellanox.com/.

[220] D. Y. Yoon, M. Chowdhury, and B. Mozafari, "Distributed Lock management with RDMA: Decentralization without Starvation," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 1571–1586.

[221] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska, "Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks," in *Proceedings of the 2019 International Conference on Management of Data*. ACM, 2019, pp. 741–758.

[222] M. Gabbrielli and S. Martini, *Programming Languages: Principles and Paradigms*, ser. Undergraduate Topics in Computer Science. Springer London, 2010, p. 145. ISBN 9781848829145

[223] A. K. Simpson, A. Szekeres, J. Nelson, and I. Zhang, "Securing RDMA for High-Performance Datacenter Storage Systems," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[224] 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 Symposium on Operating Systems Principles*, 2015, pp. 87–104.

[225] X. Wei, Z. Dong, R. Chen, and H. Chen, "Deconstructing RDMA-enabled distributed transactions: Hybrid is better!" in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 233–251.

[226] "ibv_modify_qp_rate_limit(3) - Linux man page," https://man7.org/linux/man-pages/man3/ibv_modify_qp_rate_limit.3.html.

[227] "Mellanox PCX," https://github.com/Mellanox/pcx/tree/master/config.

[228] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 371–384.

[229] "LibVMA," https://github.com/Mellanox/libvma/wiki/Architecture.

[230] "Product brief: Intel Optane SSD DC P4800X series," Aug. 2017, http://www.intel.com/content/www/us/en/solid-state-drives/optane-ssd-dc-p4800x-brief.html.

[231] A. Rosenbaum, "Multiprocess Sharing of RDMA Resources," 2018, https://openfabrics.org/images/2018workshop/presentations/103_ARosenbaum_Multi-ProcessSharing.pdf.

[232] "Dynamically Connected (DC) QPs," https://docs.mellanox.com/display/rdmacore50/DynamicallyConnected(DC)QPs.

[233] G. Brundtland and M. Khalid, "UN brundtland commission report," *Our Common Future*, p. 76, 1987.

[234] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel, "LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 756–771.

[235] J. Xue, Y. Miao, C. Chen, M. Wu, L. Zhang, and L. Zhou, "Fast Distributed Deep Learning over RDMA," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–14.

[236] J. Xing, K.-F. Hsu, Y. Qiu, H. Yang, Ziyang Liu, and A. Chen, "Bedrock: Programmable network support for secure rdma systems," in *USENIX Security*, 2022.

[237] S. Dolan, "mov is Turing-complete," *Cl. Cam. Ac. Uk*, pp. 1–4, 2013.

Appendix A

Turing completeness sketch

To show that RDMA is Turing complete, we need to establish that RDMA has the following three properties:

1. Can read/write arbitrary amounts of memory.
2. Has conditional branching (e.g., if & else statements).
3. Allows nontermination.

Our paper already demonstrates that these properties can be satisfied using our constructs but, for completeness, we also analogize our system with `x86` assembly instructions that have been proven to be capable of simulating a Turing machine. Dolan [237] demonstrated that this is in fact possible using just the `x86 mov` instruction. As such, we need to prove that RDMA has sufficient expressive power to emulate the `mov` instruction.

A.1 Emulating the `x86 mov` instruction

To provide an RDMA implementation for `mov`, we first need to consider the different addressing modes used by Dolan [237] to simulate a Turing machine. The addressing mode describes how a memory location is specified in the `mov` operands.

Table A.1 shows a list of all required addressing modes, their `x86` syntax, and one possible implementation for each with RDMA. R operands denote registers but, since RDMA operations can only perform memory-to-memory transfers, we assume these registers are stored in memory. For simplicity, we only focus on `mov` instructions used to perform *loads* but note that *stores* can be implemented in a similar manner.

For *immediate* addressing, the operand is part of the instruction and is passed directly to register R_{dst}. This can be implemented simply using an WriteImm which takes a constant in its *immediate* parameter and writes it to a specified memory location (register R_{dst} in this case). To perform more complex operations, *indirect* allows `mov` to use the value of the operand as a memory address. This enables the dynamic modification of the address at runtime, since it depends on the contents of the register when the instruction is executed. To implement this, we use two write operations with doorbell ordering (refer to §6.2.1 for a discussion of our ordering modes). The first Write changes the *source address* attribute of the second Write operation to the value in register R_{src}. This allows the second Write operation to write to register R_{dst} using the value at the memory address pointed to by R_{src}. Lastly, *indexed* addressing allows us to add an offset (R_{off}) to the address in register R_{src}. This can be done by simply performing an RDMA Add operation between the two writes with doorbell ordering, in order to add the offset register value R_{off} to R_{src}. This allows us to finally write the value $[R_{src} + R_{off}]$ to R_{dst}. With these three implementations, we showcase that RDMA can in fact emulate all the required `mov` instruction variants.

A.2 Allowing nontermination

To simulate a real Turing machine, we need to also satisfy the code nontermination requirement. In the `x86` architecture, this can be achieved via an unconditional jump [237] that loops back to the start of the program. For RDMA, this can also be achieved by having the CPU re-post the WRs after they are executed. While this is sufficient for Turing completeness it, nevertheless, wastes additional CPU cycles and can also impact latency if CPU cores are busy or unable to keep up with WR execution. As an alternative, RedN provides a way to loop back without any CPU interaction by relying on Wait and Enable to recycle RDMA WRs (as described in §6.2.4.0.2). Regardless of which approach is employed, RDMA is capable of performing an unconditional jump to the beginning of the program. This means that we can emulate all `x86` instructions used by Dolan [237] for simulating a Turing machine.

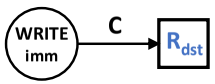
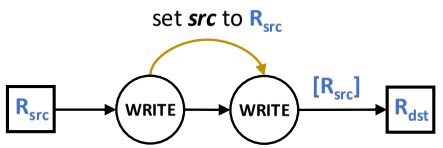
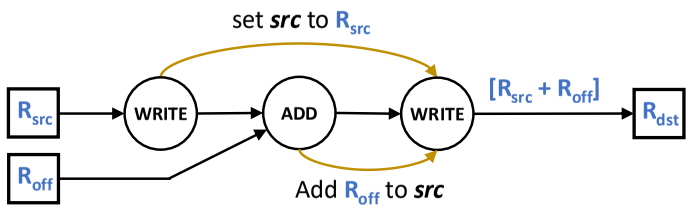
Addressing mode	x86 syntax	RedN equivalent
Immediate	mov R_{dst}, C	
Indirect	mov R_{dst}, $[R_{src}]$	
Indexed	mov R_{dst}, $[R_{src} + R_{off}]$	

Table A.1: Addressing modes for the x86 mov instruction and their RDMA implementation in RedN.