Building Efficient Software to Support Content Delivery Services

by

Benjamin Edward Cassell

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2019

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Angela Demke Brown Professor, Department of Computer Science, University of Toronto
Supervisor:	Tim Brecht Associate Professor, Cheriton School of Computer Science, University of Waterloo
Internal Members:	Kenneth Salem Professor, Cheriton School of Computer Science, University of Waterloo
	Bernard Wong Associate Professor, Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Wojciech Golab Associate Professor, Department of Electrical and Computer Engineering, University of Waterloo

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Many content delivery services use key components such as web servers, databases, and key-value stores to serve content over the Internet. These services, and their component systems, face unique modern challenges. Services now operate at massive scale, serving large files to wide user-bases. Additionally, resource contention is more prevalent than ever due to large file sizes, cloud-hosted and collocated services, and the use of resource-intensive features like content encryption. Existing systems have difficulty adapting to these challenges while still performing efficiently. For instance, streaming video web servers work well with small data, but struggle to service large, concurrent requests from disk. Our goal is to demonstrate how software can be augmented or replaced to help improve the performance and efficiency of select components of content delivery services.

We first introduce Libception, a system designed to help improve disk throughput for web servers that process numerous concurrent disk requests for large content. By using serialization and aggressive prefetching, Libception improves the throughput of the Apache and nginx web servers by a factor of 2 on FreeBSD and 2.5 on Linux when serving HTTP streaming video content. Notably, this improvement is achieved without changing the source code of either web server. We additionally show that Libception's benefits translate into performance gains for other workloads, reducing the runtime of a microbenchmark using the diff utility by 50 % (again without modifying the application's source code).

We next implement Nessie, a distributed, RDMA-based, in-memory key-value store. Nessie decouples data from indexing metadata, and its protocol only consumes CPU on servers that initiate operations. This design makes Nessie resilient against CPU interference, allows it to perform well with large data values, and conserves energy during periods of non-peak load. We find that Nessie doubles throughput versus other approaches when CPU contention is introduced, and has 70 % higher throughput when managing large data in write-oriented workloads. It also provides 41 % power savings (over idle power consumption) versus other approaches when system load is at 20 % of peak throughput.

Finally, we develop RocketStreams, a framework which facilitates the dissemination of live streaming video. RocketStreams exposes an easy-to-use API to applications, obviating the need for services to manually implement complicated data management and networking code. RocketStreams' TCP-based dissemination compares favourably to an alternative solution, reducing CPU utilization on delivery nodes by 54 % and increasing viewer throughput by 27 % versus the Redis data store. Additionally, when RDMA-enabled hardware is available, RocketStreams provides RDMA-based dissemination which further increases overall performance, decreasing CPU utilization by 95 % and increasing concurrent viewer throughput by 55 % versus Redis.

Acknowledgements

I would like to thank my supervisor, Tim Brecht, for his guidance and support throughout my Ph.D. His cheerful attitude, research ideas, consistent feedback, and help have been incredibly valuable to me throughout my graduate studies, and I owe a great deal of my success to him.

I extend my thanks as well to the members of my Ph.D. committee for their insight, reviews, and feedback (and, in some cases, for their research collaboration as well): Angela Demke Brown, Ken Salem, Bernard Wong, and Wojciech Golab.

I would also like to thank my friends and colleagues for their collaboration and support, in particular Tyler Szepesi, without whom Libception and Nessie would not exist, and Huy Hoang for his exceptional help with RocketStreams. A further sincere thanks to Ali Abedi, Milad Ghaznavi, Jim Summers, Rayman Preet Singh, Filip Krynicki, Serban Gaciu, Van Szepesi, Alex Szlavik, and Matthew Pankhurst. Their friendship helped carry me through to the end.

Additionally, I am grateful for the generous funding support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Alexander Graham Bell Doctoral Canada Graduate Scholarship, the NSERC Master's Postgraduate Scholarship, the Ontario Graduate Scholarship, the GO-Bell Scholarship, and the David R. Cheriton Graduate Scholarship. Without the financial aid provided by these sources, it would have been impossible for me to complete my degree in a happy, healthy manner.

Finally, a big thanks for the love and confidence of my parents, brothers, in-laws, and extended family, who have always encouraged me. Most importantly, I thank my wonderful wife, Emma Dost, whose patient support has made all of this possible. This thesis is dedicated to her, and there is no person more deserving.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Overview

Content delivery services are composed of groups of systems, including web servers, key-value stores, and databases, that act together to deliver content to users. While many of these systems have long-established pedigrees (for example, the Apache HTTP server [4] was initially released in 1995 and is now over 24 years old), the scale at which they must operate has evolved over time, both in terms of the size of the content being delivered and in terms of the number of users being serviced from a single machine. Recently, reports show that audio-video content alone, which is much larger than traditionally delivered content such as web pages, accounts for over 71 % of fixed-line evening Internet traffic in North America [97, 19]. These same reports estimate that audio and video content will account for over 80 % of downstream North American traffic by the end of the year 2020 [97, 19].

Not only has the scale of content changed over time, but so too have the dynamics of content delivery. As demand grows and fluctuates, many companies have opted to move content delivery services off of dedicated hardware and into the cloud. Tech giants such as Google now run many services on shared servers [122], and smaller companies are also increasingly reliant on cloud deployments provided by companies like Google, Microsoft, and Amazon. Additionally, resource demands are shifting for the systems that make up content delivery services: for example, HTTP web servers can no longer simply send content to users, but rather must grapple with the additional overhead of encrypting the content being delivered. Encryption is a heavily CPU-intensive task, one that will be challenging for services to accommodate as encryption adoption rates increase (over 50 % of all Internet traffic was encrypted in 2018, with further growth expected [98]).

In this thesis we present three software systems, Libception, Nessie, and RocketStreams, which we have designed to help meet the challenges associated with building efficient components of modern content delivery systems. In Section 1.2 we motivate the need for efficient software systems that address the concerns of content delivery service components. Next, in Section 1.3 we discuss our goals in meeting these needs. Section 1.4 outlines the contributions that our research makes towards meeting these goals. Finally, Section 1.5 summarizes this chapter.

1.2 Motivation

Content delivery services are providing data to users at unparalleled rates and with increasing timeliness constraints. Much of this content is large, with HTTP video streaming acting as the largest contributor to Internet traffic [19]. This share is growing [97, 19], with new services like Twitch [119] having served over 290 billion minutes of live streaming video content in 2016 alone [32].

Existing content delivery services' components can struggle to perform well when dealing with modern requirements. Web servers, for example Apache and nginx, have been shown to use disk resources inefficiently when dealing with large and concurrent requests for HTTP streaming video [107]. Resource contention has also begun to affect these systems in other ways. One major source of resource contention comes from content encryption, which comprises an increasingly large percentage of Internet traffic (even among video providers such as YouTube and Netflix [97]). TLS encryption of large content is very taxing on CPU resources. Netflix went to great lengths, including modifying the operating system on their delivery servers, in order to support an encrypted video workload at a reasonable level of throughput [101, 102].

In order to address performance concerns, several modern key-value stores have been designed to use technologies such as remote direct memory access (RDMA) for low-latency access. However, these systems, including FaRM [26] and HERD [47], have been designed primarily to store and transfer small amounts of data. Furthermore, many of these systems use CPU polling to guarantee low latencies [26, 47, 70, 127]. Designs of this nature suffer from notable performance degradation when CPU contention is introduced, or when data being serviced is large. Polling also wastes CPU resources, resulting in unnecessary energy consumption (which translates into higher expenditures on electricity).

Scaling out is a possible method for addressing performance issues. However, the cost of purchasing additional cloud resources or, in the case of some resource-heavy services

like streaming video, physical machines can be prohibitive. These costs are exacerbated when systems use underlying hardware inefficiently, resulting in wasteful overprovisioning. Problematically, using hardware resources efficiently is no easy task. Doing so is critically dependent on workload-specific information. Implementing efficient mechanisms into existing systems can be time consuming and difficult. Even when building new systems from scratch, appropriate design decisions must be made carefully: with RDMA, for example, the differences between the best and worst approaches for a specific use case can cause a 70 times difference in terms of throughput and 3.2 times difference in terms of CPU consumption [48]. Therefore, the right mechanisms must be carefully chosen to ensure that performance does not suffer.

1.3 Goals

A key takeaway from Section 1.2 is that the important components of content delivery services suffer from performance deficiencies caused by inefficient or workload-inappropriate use of underlying hardware resources. Recognizing this, our overall goal is to create software that either augments or replaces components of existing systems in such a way that hardware efficiency (and therefore overall system performance) is increased. What exactly this means differs from project to project. For example, with Libception, we examine systems (HTTP web servers) where hard drives serve as the performance bottleneck. In this scenario, web servers are not designed to make efficient use of available disk resources for serving streaming video (primarily due to the large size of the files being served). Hard drives are a cost-effective means of storing the massive quantities of video data hosted by services such as Netflix (in other words, we cannot simply improve throughput by completely replacing hard drives with faster storage mediums like solid state drives or non-volatile RAM, as they would be too costly). Rather, we must change the way HTTP web servers interact with hard drives in order to improve overall system performance. In past research we showed how to do this for a specific web server and operating system [107], however we would like to extend this to a more general approach that works in a wider variety of scenarios with less implementation effort.

For Nessie and RocketStreams, we have more flexibility to use improved hardware than we do for Libception's primary use case. With Nessie, our goal is to create a distributed key-value store that reduces CPU utilization (and thereby also reduce energy consumption and increase resistance to performance interference from collocated computation). With RocketStreams, our goal is to provide an easy-to-use framework for efficiently disseminating live streaming video data. In both cases, many existing systems are designed around TCP

and UDP-based networking assumptions. RDMA-enabled NICs, however, offer additional options for efficient software design at cost-competitive pricing [26]. Our goal is therefore to apply RDMA to effectively meet the demands of each system. For Nessie, we want to create a system that offers an alternative design to other state-of-the-art RDMA-based storage systems and thereby provides high performance in situations these systems struggle with. For RocketStreams, we want to provide a framework for building systems for high performance, CPU-efficient live streaming video dissemination over TCP. When RDMA-enabled hardware is available, we want RocketStreams to provide even better performance, while also reducing CPU utilization further.

Ease-of-use is also an important consideration for the software we design. For existing systems, modifying them to take advantage of performance improvements can be time consuming and costly. Our software should strive to reduce the effort required for integration by eliminating the need to write new code, and through adhering to standard usability guidelines and application protocol interfaces. This helps eliminate barriers to adoption for the systems we create. For Libception, we want to do this by providing an external library that integrates seamlessly into unmodified applications (thereby making code changes unnecessary when using Libception). Nessie, as a key-value store, should support the widely used and straightforward get, put, and delete operation API that is common across many key-value stores. For RocketStreams, we want to provide a framework that eliminates the need for applications to implement their own complicated live streaming video data management and networking code. The API provided by RocketStreams should help to reduce the amount of effort required to integrate it into existing applications such as web servers (or at least be comparable to other approaches, such as Redis [87]).

Finally, for the purposes of our research, it is insufficient to merely create new software designs. Without being able to demonstrate an improvement in performance using our software over the state-of-the-art, there is no reason to adopt our software in practice. Therefore, we should fairly evaluate existing systems and alternative designs against our own to show how performance is impacted. This empirical evidence should help to demonstrate the utility of our approaches.

1.4 Contributions

The contributions of this thesis are divided among three main research projects. The first project describes the design of a system, Libception, which helps to improve the performance of disk-bound systems reading large files at high rates. The second research project, Nessie, is a key-value store that makes novel use of remote direct memory access (RDMA)

to reduce CPU utilization in its distributed protocol. The last project, RocketStreams, describes a framework that allows for easy and efficient live streaming video dissemination, including integration into existing web servers. The following subsections detail the contributions of each project.

1.4.1 Libception

Our work on Libception is described in Chapter 3. It builds on previous research that has demonstrated that serialization and aggressive prefetching can improve disk throughput for HTTP streaming video web servers [107]. Libception, in contrast with this previous work, provides a user space library that allows unmodified industry-grade web servers to benefit from serialization and aggressive prefetching. Additionally, we show through microbenchmarks that other unmodified applications may also see performance improvements by using Libception. This work was originally published in the proceedings of the International Conference on Performance Engineering (ICPE) 2017 [110] where it was nominated for a best paper award and fast-tracked for journal publication. It has since been published in an expanded form in the Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) [14]. We make the following contributions with our research with Libception:

- We present the design and implementation of Libception, a portable, application-level shim library that implements techniques (aggressive prefetching and serialization) for improving disk I/O efficiency. We demonstrate that web servers can achieve the benefits of these techniques simply by linking with Libception at runtime, without the need for source code changes. Comparing a web server that we modify to incorporate the techniques directly, and the unmodified server linked with Libception, we find essentially identical performance.
- We show that aggressive prefetching and disk I/O serialization techniques currently implemented in Libception can approximately double the peak HTTP video streaming throughput of a variety of web servers (Apache, nginx, and the userver), both on FreeBSD, and on Linux when using the default kernel parameter settings, regardless of which Linux disk scheduler is chosen. Again, notably, these performance benefits are obtained without any changes to the source code of these web servers.
- We discover that there is great scope for improving HTTP video streaming performance on Linux, when not using Libception, by tuning kernel parameters. In

particular, by tuning parameters to improve both prefetching and serialization, we find that throughput can be more than doubled, yielding peak throughput slightly higher than that obtained with Libception. In contrast, when using Libception, kernel parameter tuning yields only marginal additional improvements.

- Finally, we use a microbenchmark to demonstrate Libception's ability to improve performance for another workload. With two instances of the diff utility competing for resources, Libception is shown to reduce execution time for both instances, cutting workload completion time by over 50 %. As with our web server improvements, these benefits are achieved without source code modifications to the diff utility.

1.4.2 Nessie

Our work with Nessie is described in Chapter 4. Nessie presents an RDMA-enabled key-value store whose unique, client-driven protocol allows it to perform well under conditions that are problematic for other state-of-the-art designs. An early version of Nessie appeared in the proceedings of the Workshop on Rack-scale Computing (WRSC) 2014 [111]. A more complete version of the system, including a revised protocol, implementation, and benchmarks was published in the Transactions on Parallel and Distributed Systems (TPDS) 2017 [15]. Nessie provides the following research contributions:

- We describe the design of Nessie, an in-memory, distributed, RDMA-enabled key-value store (RKVS). We highlight Nessie's decoupled, fully client-driven design, which allows it to operate without server-side interaction.

- We identify three problematic workloads for existing RKVSes that use server-driven and hybrid designs: workloads in shared-CPU environments, workloads with large data values, and workloads where energy consumption is important. We argue that Nessie's client-driven nature makes it more suitable for accommodating these workloads.

- We evaluate an implementation of Nessie on the workloads described above. For comparison purposes, we also evaluate NessieSD, a server-driven version of Nessie that uses polling similar to HERD [47], and NessieHY, a hybrid version of Nessie that uses client-driven get operations and server-driven put operations similar to FaRM [26]. Our evaluation shows that Nessie performs better for the identified workloads than alternative approaches.

1.4.3 RocketStreams

We describe RocketStreams in Chapter 5. RocketStreams is a framework that enables easy, light-weight, and high-throughput live streaming video dissemination. This research appeared in the proceedings of the Asia-Pacific Workshop on Systems (APSys) 2019 [13]. RocketStreams makes the following contributions:

- A description of RocketStreams, a framework which provides applications with easy and efficient live streaming video dissemination, without the need to implement data management and networking code.
- An overview of RocketStreams' API, a description of our implementation and an explanation of how, with little effort and relatively few lines of code, we utilize RocketStreams to disseminate video streams to an open source web server, the userver, which in turn services large numbers of users.
- Experiments indicate that RocketStreams provides similar dissemination node performance (CPU utilization and ingest throughput) when compared to the Redis data store, while on delivery nodes it reduces relative CPU utilization by up to 54 %, leading to a 27 % increase in simultaneous viewer throughput.
- When using RDMA, RocketStreams can ingest 18 Gb/s while simultaneously delivering a total of 144 Gb/s of traffic to delivery nodes. Comparatively, Redis is capped at 10.4 Gb/s of ingest traffic and 83 Gb/s traffic to delivery nodes. On delivery nodes, RDMA-enabled RocketStreams reduces CPU utilization by up to 95 % versus Redis, allowing the web server to support up to 55 % higher viewer throughput.

1.5 Chapter Summary

Content delivery services are increasingly being used to serve content containing large amounts of data to vast, and growing, numbers of users. As an example, audio-video content currently accounts for over 70 % of fixed-line evening North American traffic [97, 19]. By 2022, it is expected that 82 % of all IP traffic will be video traffic as user adoption rates increase. Additionally, services will continue to introduce additional features such as ultra high resolution video and high dynamic range [97], which will not only increase the size of content being delivered, but will comprise a growing percentage of total delivered content [19]. Finding efficient ways to deliver content is therefore imperative for services

to scale in a cost-effective manner. Unfortunately, the components that make up content delivery services are often ill-equipped to deal with modern content delivery workloads, and their inefficient use of hardware results in suboptimal performance. In this thesis, we design software to help address these shortcomings, making better use of hardware with the end goal of improving overall system performance. We evaluate our software and find that we are able to greatly improve performance over existing state-of-the-art approaches. We believe these findings can be used to help design and implement more efficient content delivery services in the future.

Chapter 2

Background and Related Work

2.1 Background

In this section we detail some of the important hardware, software, and background concepts used throughout the research in this thesis.

2.1.1 HTTP Streaming Video

HTTP streaming video is an incredibly popular and growing modern service, and is currently the largest contributing source of IP traffic on the Internet [19]. Improving the performance of systems that deliver HTTP streaming video is the focus of two of the research projects found in this thesis, Libception in Chapter 3, and RocketStreams in Chapter 5. These projects deal with two different types of HTTP streaming video: the first type, video on demand (VOD), consists of video content, such as movies, television shows, and other videos, that is static in the sense that content exists in its final form before being selected by and delivered to users. Examples of major video on demand services include Netflix, Hulu, and YouTube. Libception focuses on improving the disk performance of web servers providing video on demand content. The second type of HTTP streaming video, live streaming video, consists of video content which is sent to users as it is created, typically from a source such as a concert, sporting event, or content creator broadcasting their activities. Examples of major live streaming video services include Twitch [119] and YouTube Live [134]. Our work with RocketStreams seeks to provide a flexible and efficient framework for live streaming video dissemination.

When improving the components of content delivery services, it is important to understand how the components fit into the services' bigger pictures. Figure 2.1 depicts an example of an HTTP streaming video on demand service. In this service, content is stored on primary servers. The content is replicated as needed (typically based on popularity or estimated future popularity) to geo-distributed content delivery network (CDN) servers, some of which may be owned by third party services (for example in Internet exchange points), and some of which may be localized within the networks of Internet service providers. The video content is then delivered from the CDNs to users. This process is similar to the delivery methods used by services such as Netflix. Libception targets disk throughput improvements on the primary and CDN servers.

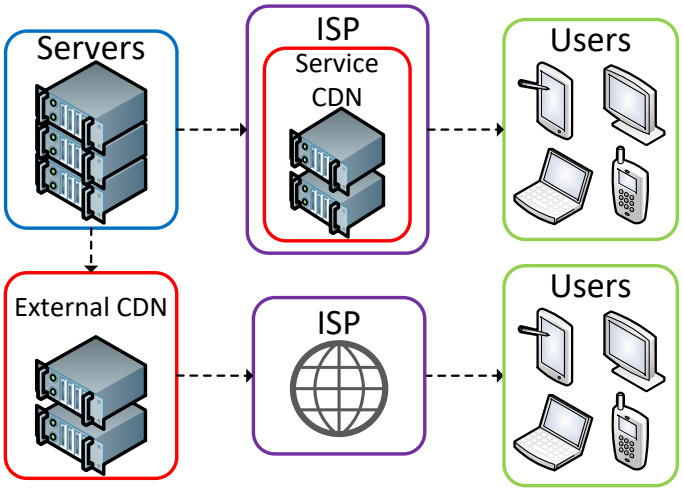

Figure 2.1: An overview of an example HTTP streaming video on demand service. The service disseminates video from primary service-owned servers to CDN servers. The CDN servers are then used to deliver video content to users.

Another example service is depicted in Figure 2.2. This workflow represents one that would be used by HTTP live streaming video services, such as Twitch. In the figure, content is generated dynamically by producers. These producers forward the video data to servers in geo-distributed data centres and IXPs (which ingest this data). The video data is further disseminated to other servers within and across data centres to meet scaling demands. Finally, the servers deliver the content to users, who consume it. RocketStreams is a framework that exposes an efficient and easy-to-use API for handling data during the dissemination phase of the given workflow.

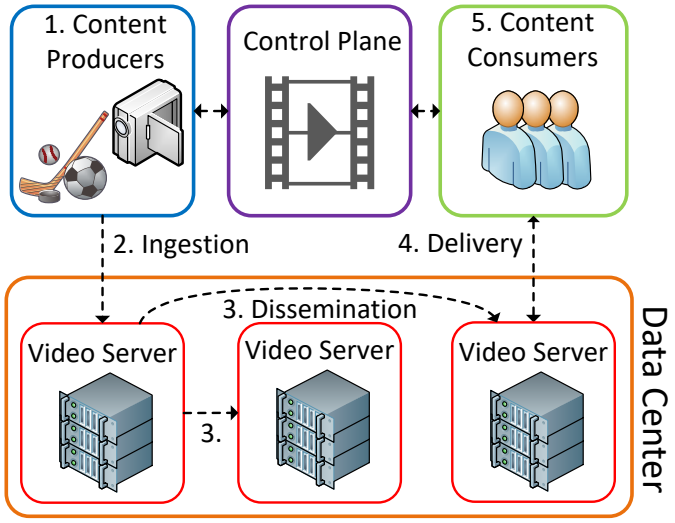

Figure 2.2: An overview of an example HTTP live streaming video service. Generated content is sent to data centres where it is replicated and forwarded to users.

2.1.2 Serialization

HTTP web servers that serve the long tail of less-popular video on demand content are frequently disk-bound [51, 106]. For Libception in Chapter 3, we examine several disk-based techniques for improving throughput. Serialization is one of these techniques. We define serialization as ensuring that only one disk request per device is presented to the operating system at a time. This means that requests to each device are serviced sequentially (one at a time) instead of in parallel (many at a time). There is an inherent trade-off in reading data at high throughput from a hard disk and ensuring responsiveness for applications across the system by multiplexing disk resources between applications. Operating systems often err on the side of fairness by default, splitting large disk requests into multiple smaller ones so they may be interleaved with disk requests from other applications. Although this can increase responsiveness for some types of applications, past research has demonstrated that the introduction of concurrent I/O streams results in severely reduced I/O throughput on Linux, across a variety of block I/O schedulers [80].

This reduction in throughput is attributable to the physical manner in which hard disks operate. When two or more disk requests are serviced by the operating system from a hard

disk in pieces (as opposed to consecutively), although the total amount of data read from the hard disk is the same, the disk head may need to move between tracks of the disk in order to service the different pieces (referred to as seek latency). Additionally, the disk head may also need to wait for the appropriate sectors to become available for reading as the disk rotates (referred to as rotational latency). These additional rotational and seek latencies translate directly into throughput losses (and therefore overall performance losses) for systems that must service many large disk requests. This is demonstrated in Figure 2.3, which shows a simplified view of read requests being serviced from different areas of a hard disk, with and without request serialization. The figure labels the seek, rotational and read latencies associated with servicing the requests. When the requests are serviced in parallel, the added seek and rotational latencies result in a much longer overall service time for the requests. When servicing the requests in a serialized manner, the requests complete in a quicker timeframe, and the throughput from the disk is thus higher.

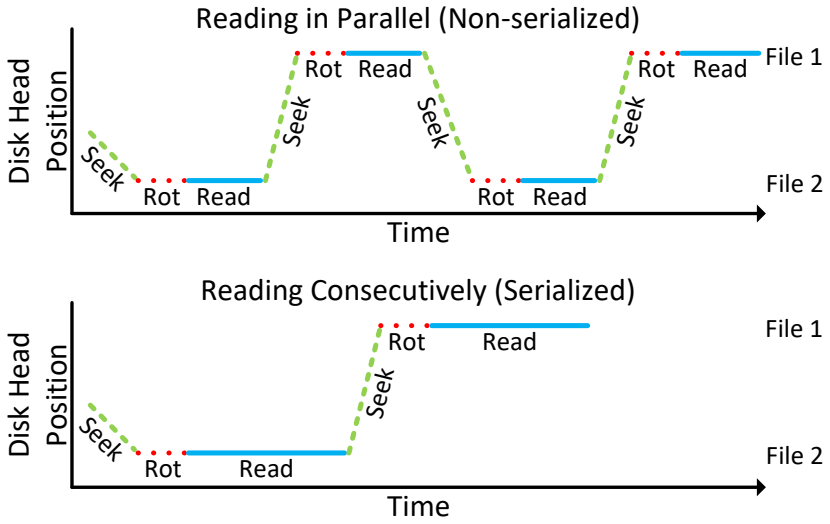

Figure 2.3: A simplified view of read requests being serviced from a hard disk with and without serialization. Serialization eliminates seeks and rotations by combining multiple small reads into single large ones, reducing the overall time needed to read data.

2.1.3 Prefetching

Prefetching is another technique employed by Libception in Chapter 3 to improve disk throughput for HTTP streaming video web servers. Prefetching is similar to serialization in that its goal is to increase disk throughput by eliminating seek and rotational latencies. The way this is done however, is different. Prefetching takes advantage of two primary concepts to achieve performance benefits: data locality and free system memory. Data locality refers to the idea that applications accessing a part of a file will likely want to access subsequent data from that file in the near future. This directly applies to HTTP streaming video workloads (a user watching a video will soon make a request for the next part of that video file), but is also useful across a variety of other workloads as well. By extending a disk request beyond what was asked for by an application, the operating system can read additional data into a file cache that resides in unused system memory. On future requests, the prefetched data may then be returned immediately from the file cache, without incurring any hard disk-related performance costs. The amount of free system memory determines how much total data can be read into the file cache at any given time. Exceeding this total results in the eviction of old data to make room for new data.

An example scenario involving prefetching is shown in Figure 2.4, which depicts read requests being serviced from a hard disk, with and without prefetching. When prefetching is enabled, initial read requests for portions of a file encounter slightly higher latencies (although this is often masked by the operating system which continues to prefetch data asynchronously in the background after any application-requested data has been retrieved). However, the prefetched data is now resident in the file cache. Requests for subsequent data will be returned immediately from RAM. This results in a significant improvement in terms of both throughput and average request latency when compared with the same reads performed without prefetching (as additional seek and rotational latency penalties are incurred).

2.1.4 Block I/O Schedulers

Libception's use of disk request serialization in Chapter 3 often leads to a discussion of block I/O schedulers as an alternative means of re-ordering disk requests (and potentially improving throughput). In most modern operating systems, block I/O schedulers are components that sit between the user I/O interface and block device drivers (software that physically accesses blocks of data from hardware). The primary role of the block I/O scheduler is to manage the execution order and size of disk requests. The actual strategies employed vary from scheduler to scheduler. Linux, for example, has had three different

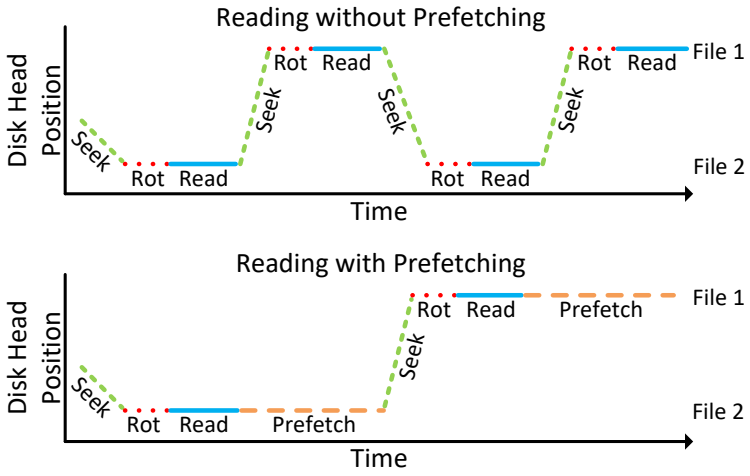

Figure 2.4: A simplified view of read requests being serviced from a hard disk with and without prefetching. Prefetching eliminates seeks and rotations by extending reads to make future ones unnecessary, resulting in more data being read during the same timespan.

block I/O schedulers since kernel version 2.6.33: NOOP, completely fair queuing (CFQ) and deadline. Each of these options takes a slightly different approach towards managing request scheduling, all with the goal of optimizing either throughput or latency for certain patterns of disk access.

The NOOP scheduler uses a first-in-first-out (FIFO) queue to schedule requests. It additionally has the ability to merge multiple requests into larger requests [7]. The deadline scheduler is more complex than NOOP, and has queues for reads and writes which are sorted based on physical sector, but also a queue which is sorted based on expiration times (from which the name deadline derives). Scheduling preference is given to expired requests, with requests otherwise being serviced from the sector-based queues. Deadline is particularly appropriate for workloads that wish to place bounds on I/O request latency [7]. Finally, CFQ attempts to provide fairness between processes, providing access to the disk in time slices to per-process request queues [7]. These time slices also help CFQ to avoid deceptive idleness (which occurs when naive block I/O schedulers switch processes during small data processing gaps between I/O requests from the same process). When we evaluate Libception, we examine the performance effects of different block I/O schedulers in conjunction with serialization and prefetching.

2.1.5 RDMA

Moving away from disk throughput, a good portion of our research deals with network-related efficiency. Remote direct memory access (RDMA) is an efficient modern alternative to traditional networking protocols like TCP and UDP, that also offers a suite of additional utility features. It is a major component of Nessie's design in Chapter 4, with Nessie's protocol design relying on the unique tools provided by RDMA. RDMA is also used (when available) to help provide performance benefits for RocketStreams in Chapter 5. An exhaustive breakdown of Mellanox's RDMA implementation (which we employ in all of our research) is available through the vendor's website [66], and we summarize the important aspects of the protocol here.

RDMA uses specialized hardware in an RDMA-enabled network interface card, referred to as an RNIC, to register regions of memory, exposing them to the RNIC. The RNIC is then able to directly interact with the registered memory, bypassing the CPU. This can be seen in Figure 2.5, which shows a simplified overview of two physical nodes communicating using RDMA.

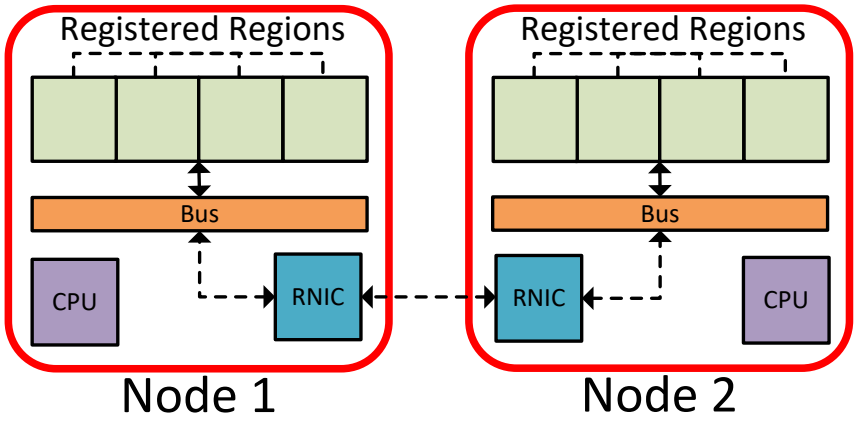

Figure 2.5: An overview of two nodes communicating using remote direct memory access (RDMA). Both nodes may interact with the registered memory regions of the other node through the RNIC, bypassing the CPU.

The RDMA protocol defines basic operations referred to as verbs. All verbs are similar in that they must operate on already-registered regions of memory. The method for

initiating an operation and completing an operation is also similar: outgoing operations are posted to the send queue in a queue pair (QP), a control data structure accessible to the RNIC. Likewise, expected incoming operations that require server-side notification are posted to queue pair's recv queue. When a requested operation posted to a send queue completes (assuming that the operation also requested a completion notification), or when an incoming operation consumes resources posted to a recv queue and completes, a work completion event is posted to another data structure, a completion queue (CQ).

RDMA supports several transport types for queue pairs, reliable connection (RC), unreliable connection (UC), and unreliable datagram (UD). Both the reliable and unreliable connection transport types require each communicating endpoint to be connected, with a unique queue pair on both sides of the connection. The unreliable datagram transport type does not require connections, meaning a single queue pair on a server could be used to communicate with any number of unreliable datagram queue pairs on other servers. This connected versus unconnected paradigm is similar to the connected and unconnected natures of TCP and UDP. Also similar, the RC transport type guarantees delivery, whereas the UC and UD transport types do not. This has further implications: although the UD transport type arguably scales better (due to consuming fewer RNIC and host resources for queue pairs) and provides multicast functionality, it supports only a subset of the RDMA protocol (RDMA send and recv verbs, described in depth later), and does not allow for message sizes that exceed the maximum transmission unit (MTU) of the network. The UC transport type supports a slightly expanded subset of operations (RDMA send, recv, and write), and also supports message sizes up to 1 GB. The RC transport type supports all RDMA verbs (send, recv, write, read, atomics), and message sizes up to 1 GB. This information is summarized in Table 2.1.

	Unreliable Datagrams	Unreliable Connections	Reliable Connections
Supported Verbs	send, recv	send, recv, write	send, recv, read, write, atomics
Multicast	Yes	No	No
QP Semantics	Any-to-any unconnected	One-to-one connected	One-to-one connected
Reliable	No	No	Yes
Max Size	Network MTU	1 GB	1 GB

Table 2.1: A summary of the features supported by the different RDMA transport types.

RDMA verbs can be categorized according to which systems are involved in processing the verb. A "two-sided" verb involves processing on both the issuing (client) end and receiving (server) end. The RDMA send and recv verbs are two-sided, and are similar to the send and recv mechanisms employed by TCP and UDP (all TCP and UDP mechanisms are two-sided), albeit that the RDMA send and recv verbs operate directly on user space

memory without involving the kernel (a recv verb pre-selects where received data will be copied to by the RNIC). An RDMA send verb posted to the send queue of a client QP results in a pre-posted recv verb being consumed from the recv queue of a server QP. Additionally, a work completion event is generated on the server. If requested, a work completion event is also generated on the client, either when the RNIC has pushed out all data (in unreliable mode), or when the RNIC receives confirmation of delivery (in reliable mode).

RDMA additionally provides "one-sided" verbs, including read and write. One-sided verbs copy memory directly to and from registered regions on remote nodes, without involving the remote node's CPU. RDMA reads and writes therefore generate no completion events on the remote end (although as with two-sided verbs, optional completion events may be requested on the initiating side). RDMA additionally provides two 64-bit atomic verbs, compare-and-swap (CAS) and fetch-and-add (FAA) which, in addition to being one-sided, are synchronized relative to other RDMA verbs on the remote RNIC (again, no completion event is generated on the remote end, although the initiating node may request an optional completion event). Other verbs also exist, notably the hybrid RDMA write-with-immediate verb which combines a one-sided RDMA write with a 32-bit data item that is delivered as a completion event to the remote end. RDMA write-with-immediate verbs therefore consume posted operations from the remote queue pair's recv queue, and are processed in the same manner as RDMA recv verbs albeit with the addition of the immediate data.

To improve performance, RDMA reduces overhead by using zero-copy data transfers and kernel-bypass. RDMA verbs can therefore reduce latency by one or more orders of magnitude when compared with traditional counterparts like TCP and UDP [26, 47], and also provide high throughput. A small RDMA read or write, for instance, can complete in as little as a few microseconds as opposed to the tens or even hundreds of microseconds required for a TCP round trip. In addition to the asynchronous event model provided for determining when RDMA verbs complete, because RDMA verbs are processed in address order, applications may poll on the last byte of a memory region to determine when some operations have completed. For example, a client requesting an RDMA read from a remote node (or a server expecting an incoming RDMA write from a remote node) can poll on the last byte of the region being updated and use that byte changing as an indication of verb completion. This acts as a trade-off between higher CPU utilization and potentially lower latency.

In addition to its many benefits, RDMA-enabled NICs have been cost-competitive since at least 2013–2014 [26]. At that time, 40 Gb/s InfiniBand networking hardware was priced similarly to 10 Gb/s Ethernet hardware [70] when comparing costs with overall device

throughput. RDMA was originally intended to be used with InfiniBand-based network in LAN settings, however the introduction of the iWARP [129] and RoCE [130] protocols enable RDMA to be used over Ethernet and even over a WAN (using TCP tunnelling).

2.1.6 Network Sidedness

An important concept when using RDMA to build systems is which nodes, clients or servers or both, are responsible for initiating and handling requests. This concept, which we refer to as "network sidedness," ties in with the differences between one-sided and two-sided networking operations described in Section 2.1.5, and allows us to classify systems as "client-driven" (CD), "server-driven" (SD), or "hybrid" (HY). The choices of network sidedness made for a system can significantly impact the system's performance. For example, in Chapter 4, Nessie presents a client-driven protocol, from which it derives most of its performance benefits. Likewise, in Chapter 5, when RDMA is available, RocketStreams carefully takes advantage of one-sided operations to minimize CPU utilization and improve performance.

We now elaborate on what differentiates applications that adhere to separate classifications of network sidedness. In a server-driven system, server nodes contain processes that wait for incoming requests made by client nodes and then service and respond to the requests using CPU resources on the server. All systems that use traditional networking protocols (like TCP) are server-driven systems, and some RDMA-enabled systems opt to use server-driven principles as well. The event-driven RDMA send and recv verbs are one example of server-driven RDMA networking calls, however instead of using the asynchronous event mechanisms associated with these verbs, some applications instead opt to poll on memory regions written to by RDMA send or write verbs. Polling strategies achieve low latency at the cost of high CPU consumption, and are also considered server-driven. In both polling and event-driven models, as the server coordinates all access to data, a server-driven design can also help to simplify synchronization of access to complex data structures. A client-driven system is one that issues exclusively one-sided network operations. For these systems, requests consume little to no CPU resources on the servers that clients communicate with. Because traditional networking protocols like TCP and UDP do not provide one-sided network operations, these designs are typically only achievable using alternate protocols such as RDMA. Finally, we also recognize hybrid systems, which contain a mixture of operations that are exclusively client-driven as well as operations that are server-driven.

2.1.7 Data Coupling

Beyond network sidedness, in Chapter 4 Nessie is also concerned with the concept of "data coupling" as it attempts to achieve high performance. We discuss this concept within the context of key-value stores (although it applies equally to other distributed storage systems).

In order to provide efficient key operations, many systems use O(1)-lookup data structures (sometimes called index tables) which map keys to locations in data storage structures. The location of this stored data relative to the location of indexing metadata can be used to classify systems. When a system stores its data and indexing metadata in separate structures, we use the term "decoupled" to refer to systems that allow metadata to reside on any physical node, and "partially coupled" to describe systems where indexing metadata must reside on the same physical node as the data it refers to. We use the term "fully coupled" to refer to designs where indexing metadata and data must be collocated within the same structure on the same physical node. A visual depiction of the different types of coupling is provided in Figure 2.6. Data coupling can impact the efficiency of data lookups for a system. Fully coupled systems may benefit from lookups that retrieve keys and data simultaneously which can be beneficial for small data, but this limits the flexibility of data placement. Decoupled systems can potentially place data on nodes to exploit locality and access patterns.

2.1.8 Cuckoo Hashing

As a distributed hash table, Nessie requires a hashing scheme in Chapter 4 to determine which keys map to which indices in its lookup data structure (index table). The hashing scheme used by Nessie in its protocol design is called cuckoo hashing. Traditional cuckoo hashing is a hashing scheme in which each key maps to two potential locations in a hash table. This mapping is typically created with a single hash function and two different seeds for each key. When a lookup (or get) operation is performed on a key, each of the key's potential locations are checked in order to find a match (if one exists). Given that the number of checks is bounded by the key's fixed number of potential locations, get operations always take constant time, even in the worst case. When an insertion (or put) operation is performed on a key, each of the key's locations are checked in order to find a match (if one exists). If a match exists, the entry is updated. If no match exists, the first available empty location is updated to contain the key (and its associated value). If no empty location exists, then one of the key's two locations is picked as a victim, and the key-value pair in the victim location is moved to its alternate location. This movement

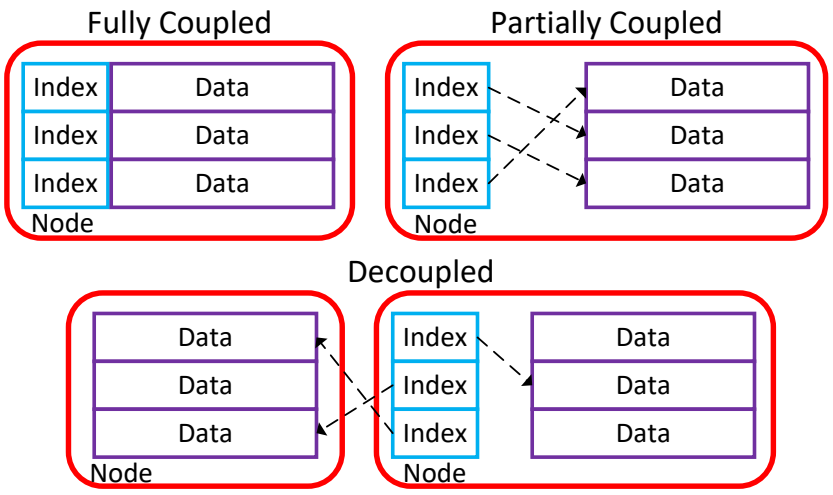

Figure 2.6: Data stores have several options for data coupling, and may store data and indexing metadata together on the same node (fully coupled), separately on the same node (partially coupled), or separately across nodes (decoupled).

is referred to as a migration. Once the migration is complete, the now-empty location is updated to contain the key-value pair that initiated the put. Migrations may result in other recursive migrations, however puts complete in expected constant time so long as the load factor of the cuckoo hash table (how full the table is) remains under 50 % [79].

The particular type of cuckoo hashing Nessie uses is called N-way cuckoo hashing. This is identical to traditional cuckoo hashing, except that each key maps to N possible locations instead of just two. Increasing the value of N results in operations that could take longer in the worst case as they may need to perform additional hashing and inspect additional entries to complete. However, in exchange, a larger value of N allows the hash table to maintain a higher load factor while still achieving expected constant time for put operations. For instance, when using 3-way cuckoo hashing, puts complete in expected constant time so long as the load factor of the table remains below 91 % [71], a substantial increase when compared with the 50 % load factor required by 2-way cuckoo hashing. A visualization of a small 3-way cuckoo hash table can be seen in Figure 2.7. In this visualization, a new key D is inserted into the table, however all of its mapped locations are full. To make room for key D, key A is migrated to an alternate location, and then D is placed in the newly emptied location.

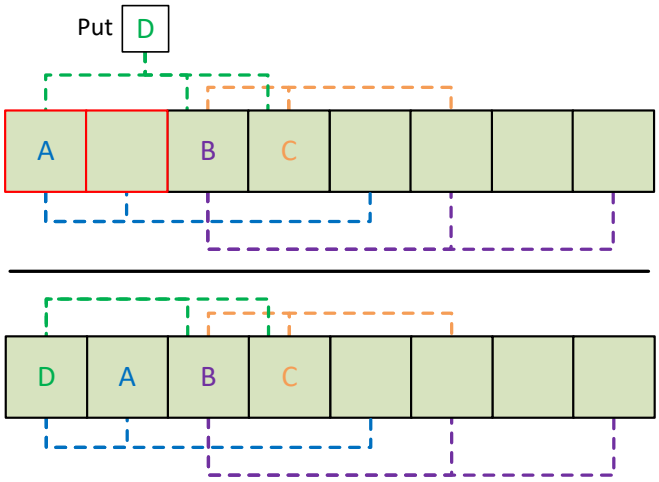

Figure 2.7: A small 3-way cuckoo hash table. When key D is inserted into the table, it will displace (migrate) key A to one of A's alternate locations.

2.2 Related Work

In this section we detail previous research related to our own.

2.2.1 Libception

For Libception, presented in Chapter 3, we identify two main categories of related research: research that has studied video on demand, and research that focuses on improving disk throughput.

Video on Demand

Video on demand's surging popularity makes it a prime choice for optimization. In 2016, Netflix alone represented 35 % of peak North American fixed downstream traffic, with Amazon video and YouTube consuming 4 % and 17 % respectively [97]. This total is expected to grow as Internet video becomes more popular and adds additional features that grow the size of delivered content [19]. Improving the efficiency of video on demand

service components can therefore have a far-reaching impact. In order to do so, however, it is important to understand the properties of the HTTP web servers being used to serve video on demand content. It is also important to understand what typical video on demand workloads look like for evaluation purposes.

HTTP web servers can be heavily disk-bound when serving streaming video content [51, 106]. This occurs due to video on demand services storing a large amount of varied content. Netflix, for example, had a 2 PB catalogue as of 2016 [106], and has added a significant amount of content since then. Although it is cost-effective to serve popular content from solid state drives, it is too costly to store and serve the large volume of long-tail content in the same way. Instead, large (and comparatively inexpensive) hard drives are used to store the long tail of content, and most requests for this content must be served from disk [106].

Previous work has demonstrated how to generate workloads for video on demand web servers, and how to benchmark these workloads [108]. It has also shown that having web servers store video on demand content as single large files is more efficient than storing the content in chunks [109]. There is also a good deal of information about other video on demand workload characteristics like watching habits, video lengths, bit rates, and popularity distributions for services such as YouTube [34, 29] and Netflix [1, 106]. We make use of this information to define our workloads when evaluating Libception.

Improving Disk Throughput

The main goal of Libception, discussed in Chapter 3, is to improve the disk throughput of HTTP streaming video web servers. To achieve this goal, Libception uses serialization, described in Section 2.1.2, and prefetching, described in Section 2.1.3. Although some disk applications are able to take advantage of file caching to improve their performance, the file cache competes with prefetching for system memory [11]. Additionally, there is reduced opportunity for HTTP streaming video web servers to benefit directly from file caching when they are serving long-tail content from disk, as the long tail of content is, by definition, infrequently accessed.

Disk prefetching was initially employed by applications to offset I/O latency, preventing CPU stalls by overlapping data reads with computation [12, 83, 121]. Some degree of disk prefetching is performed at the hardware level (which we refer to as disk readahead) and has substantial benefits [94, 95], and both this type of prefetching and the software-level prefetching used by Libception are key components of high disk throughput. Over time software disk prefetching sizes have grown substantially to accommodate modern workloads. The large prefetching that is often used by systems today to improve throughput is

referred to as aggressive prefetching [81].

As mentioned in Section 2.1.3, disk request latency can increase when prefetching data. The increased latency of disk requests is not a concern for HTTP streaming video servers, as the clients they serve are already designed to buffer data and tolerate wide area network latencies (much higher than disk latencies) and changes in available bandwidth. Rather, the improved overall disk throughput is highly beneficial for HTTP streaming video servers. As a result, other research projects have examined prefetching within the context of improving throughput. DiskSeen [44] introduces I/O request history-aware prefetching into the Linux kernel. libprefetch [120] allows applications to provide a modified kernel with application-directed prefetching hints. By comparison, Libception combines aggressive prefetching with I/O request serialization, and requires no changes to application or kernel code.

Choosing the right amount to prefetch in order to maximize disk throughput can be complex. Whereas larger prefetches help to amortize disk access overheads (such as seek latency and rotational latency), prefetching too much data can create memory pressure that evicts useful data from the file cache. Previous research with large, fixed-size prefetching showed a large improvement for system throughput when servicing sequential streams [80]. Other research has used algorithms to determine prefetch sizes based on hard drive characteristics [58]. Our work with Libception uses fixed-size prefetching on a per-experiment basis, however Libception is capable of supporting more complex prefetching strategies. This flexibility is important, as we have also demonstrated in other research that the best prefetch size depends on the amount of memory available in the system as well as the particular characteristics of a given workload [107]. Likewise, we have demonstrated that it is possible to automate the prefetch size selection process based on runtime performance in order to make efficient, live prefetching decisions [107]. Further work based on this also showed that exploiting workload-based characteristics could further improve prefetching [106]. We leave examining non-intrusive ways to integrate these techniques into Libception to future work.

Because HTTP streaming video web servers generate large, concurrent disk requests, serialization is the next important component used by Libception to help maximize disk throughput. Some systems have examined how to multiplex disk resources to establish basic performance guarantees on a per-request basis while servicing numerous concurrent requests. Argon [123], for example, uses disk-head time-slicing to provide minimum throughput guarantees to cloud-based applications, ensuring that they are treated fairly according to their service-level agreements. Within the context of serving streaming video from disk, however, there are different concerns — HTTP streaming video workloads seek to prevent clients from re-buffering. Given that a client will typically buffer between 5 to 30 seconds worth of video [9], the delivery of the video is largely insensitive to disk

latency. This means that minimum throughput guarantees on a per-request basis are less important than improving the overall throughput of the system as a whole. By forcing the system to schedule singular, large disk requests, Libception reduces the amount of time wasted waiting for the disk to seek and rotate between requests (exchanging reduced fairness between individual requests for increased overall throughput). As will be seen in Section 3.4.6, latencies under Libception are also reduced.

Finally, we consider the impact of block I/O schedulers on HTTP streaming video web servers. While the block I/O schedulers included with Linux are designed for a variety of workloads, some past studies have examined how to exploit specifically designed scheduling algorithms in order to maximize throughput of broadcasting servers that push video to clients [24, 33, 100]. Our work with Libception found that the choice of Linux block I/O scheduler has minimal impact on web server throughput when serving video on demand, especially given that Libception's use of serialization essentially acts as user-level scheduling.

2.2.2 Nessie

Nessie, described in Chapter 4, is a client-driven, RDMA-enabled key-value store protocol. Based on this, we identify, three main categories of related work: general RDMA-related research, research on RDMA-enabled storage systems, and recent research on the performance of systems with different network sidedness characteristics.

RDMA

RDMA plays a key role in the design of Nessie, and has been studied in a variety of related research. One important study explores design guidelines for RDMA-enabled systems [48]. It finds that choosing the right RDMA design techniques (which are discovered to be highly application-dependent) is critical for achieving high performance, and picking the wrong combinations can result in a factor of seventy drop in throughput, and a 3.2 times increase in CPU consumption [48]. Khrabrov et al. [54] demonstrate how to minimize the CPU impact of data serialization when working with RDMA. These types of findings are important to keep in mind when designing RDMA-enabled systems.

Some research has looked at providing RDMA in environments where it would not normally be available, such as virtual machines and hypervisors, where it continues to provide high-performance benefits [22]. LITE [116] implements a kernel-level layer that enables a flexible user space abstraction for RDMA, NIC resource scaling, and resource

sharing at the cost of giving up local kernel-bypass while performing operations (remote kernel-bypass is retained for one-sided operations). LegoOS describes a distributed operating system comprised of hardware monitors that communicate using RDMA-enabled messaging [99]. Using RDMA on virtual machines, in the kernel, and to implement new operating system techniques are interesting approaches, however these techniques do not address Nessie's primary concern which is the implementation of a strictly client-driven key-value store protocol.

Several systems have looked at making RDMA more accessible. The RaMP system, published after Nessie, provides an interface for managing loosely coupled memory regions [67]. Under RaMP, applications allocate regions of memory that are eventually shared between nodes at application-directed times (for example, during system-wide load balancing). RaMP is strongly purposed toward applications within a different space than Nessie, namely applications that can partition data and do not need immediate access to it across multiple nodes simultaneously. Nessie provides an interface under which data access is coordinated, using a distributed protocol, across all servers, and additionally implements fully client-driven operations. The Rocksteady system [55] achieves a goal similar to RaMP, but does so by constructing an RDMA-enabled migration protocol for data stored in RAMCloud [78]. Again, however, Nessie's focus is not on data migration, and its protocol is client-driven.

Another more recent system that post-dates Nessie is Remote Regions, which introduces a file system named RegionFS [2]. Processes interact with RegionFS as they would any other file system, however the "files" provided in this file system are actually composed of memory pools that are synchronized across nodes using RDMA reads and writes. In addition to having access to typical file-based API calls (such as read and write), processes may also interact with RegionFS regions using memory semantics by allocating and freeing buffers through an API. This is a highly convenient API that facilitates the development of RDMA-enabled applications, however it serves a different purpose than Nessie, which is to explore a strictly client-driven key-value store protocol in order to improve throughput for workloads that existing RDMA-enabled key-value stores are not designed to handle. Additionally, while Nessie's operations have built-in synchronization, processes must coordinate through application level logic when using Remote Regions, although this is facilitated by distributed synchronization primitives also provided by Remote Regions.

RDMA-Enabled Storage

Distributed storage systems are used to support a variety of services, and include key-value stores (like Nessie) that store data (values) associated with lookup identifiers (keys). Ex-

isting distributed storage systems like Redis [87] provide persistent storage, whereas others like memcached [31, 52] are designed to store temporary cached values. Some systems, like Bigtable [16] and Cassandra [56] may store large amounts of structured data using disk, however many modern systems such as RamCloud [78] opt instead to exclusively use main memory for storage. RDMA-enabled distributed storage systems such as Nessie can expand upon the capabilities of these systems by drawing upon RDMA's unique features to achieve low latency and high throughput.

In order to improve the performance of existing distributed storage systems, several research projects have modified existing systems to use RDMA instead of traditional networking protocols like TCP. This has been done for memcached [46, 45, 104], HBase [41] (a distributed, non-relational database), HDFS [43] (a distributed file system) and its associated remote procedure call protocol Hadoop RPC [61], Hadoop MapReduce [124], and Spark [62] (a distributed storage and computation platform). Each RDMA-modified application achieves performance improvements in terms of throughput gains and latency reductions when compared with their original versions. However, these applications were designed with the limitations of traditional networking protocols in mind, and often RDMA modifications to existing systems amount to a simple replacement of traditional networking operations with their RDMA-based equivalents.

Newer RDMA-enabled distributed storage systems are designed with RDMA in mind from the outset, allowing them to explore different avenues for improving performance by taking advantage of RDMA's unique properties. Pilaf [70] is one of the first of these systems, and achieves throughputs roughly 20 times higher than those of existing in-memory key-value stores that do not use RDMA. Pilaf is partially coupled, and uses a hybrid communication design where RDMA reads perform one-sided get operations and RDMA send operations are used to initiate put and delete operations (which are then completed server-side to facilitate synchronization). Although Pilaf demonstrates high performance compared to systems that do not employ RDMA, other systems achieve better performance by taking advantage of more efficient parts of RDMA's feature set (for example, by avoiding event-driven RDMA send and receive operations on reliable connections which are slow compared to alternative operations).

FaRM [26] implements a shared address space across a cluster. FaRM is a hybrid system that uses RDMA writes to pass messages into a server-polled circular buffer, and RDMA reads to perform lock-free data retrieval. FaRM supports ACID transactions, and uses version number metadata and operation timeouts for consistency. FaRM takes advantage of object collocation and function shipping to exploit local memory whenever possible, as the authors note this is about 23 times faster than RDMA operations. On top of their base system, the authors construct a distributed hash table that is either

fully coupled or partially coupled depending on the size of the data being stored. The authors show that their system obtains high performance, however some of this benefit is obtained from non-trivial modifications to the Windows kernel's memory management subsystem. When comparing FaRM with Nessie, Nessie provides a decoupled storage model that makes it well-suited towards flexibly storing large data objects, and is strictly client-driven, eliminating the need for polling server threads.

The authors of HERD [47] present a key-value store designed to showcase efficient RDMA use. HERD is a fully coupled system, and is exclusively server-driven, relying on polling server threads that monitor buffers to detect and respond to requests. This design trades high CPU utilization for low latency and high throughput, with HERD's evaluation demonstrating over two times the throughput and less than half the latency per request of both FaRM [26] and Pilaf [70]. HERD uses RDMA writes to initiate operations, and takes advantage of RDMA write and send operations over unreliable transports to reduce the number of network round trips made by certain RDMA verbs and improve scalability. HERD limits itself to experiments with small data (256 bytes and under) so it can take advantage of additional RDMA optimizations that reduce DMA copying and latency, and makes some trade-offs to achieve its performance. In particular, its polling server threads are CPU-heavy when compared to a system like Nessie, which is fully client-driven. This makes HERD difficult to use in environments where data storage and computation are collocated. Furthermore, systems that are not fully coupled, like Nessie, provide better opportunities for efficiently working with large data.

DrTM [127] presents a design for an RDMA-based in-memory transaction processing system. Uniquely, DrTM combines RDMA with hardware transactional memory (HTM) to provide concurrency control by ensuring that local operations are aborted if they are interfered with by a remote RDMA operation. The need to combine HTM with RDMA means that DrTM is implemented as a partially coupled system. DrTM distinguishes between two types of stored data to account for different types of access patterns, ordered (sorted) and unordered (unsorted), and also distinguishes between writes which insert new values and writes which update existing values. Reads and writes on unordered data use client-driven RDMA verbs (in conjunction with hardware transactional memory), but reads and writes on ordered data are completed using server-driven RDMA verbs. DrTM is designed to provide low latency for operations involving small values. In comparison, Nessie's decoupled nature helps it to perform well while manipulating large data. Additionally, Nessie is fully client-driven, which reduces its CPU energy footprints on server nodes during periods of system inactivity. Nessie also does not require specialized HTM hardware to provide concurrency control, but rather uses a unique distributed protocol to achieve synchronization.

The authors of RStore [115] present an RDMA-enabled distributed address space implementation. RStore uses a flat 64-bit namespace which allows applications to allocate and access data on participating servers. RStore provides applications with the ability to pre-allocate RDMA resources, allowing them to avoid some overhead on object access. Two example applications are demonstrated running on top of RStore, a key-value sorter and a graph-processing framework, both of which are shown to be several times faster than other modern equivalents. Despite its high performance, however, RStore uses a logically centralized master node to synchronize metadata, and does not provide I/O synchronization with respect to accessing data within the distributed namespace. Nessie, by comparison, does not use a master node to coordinate operations. Furthermore, Nessie provides strong synchronization guarantees when accessing data across the cluster, preventing the need for much of this logic to be implemented at the application level.

KV-Direct [57], published after Nessie, takes an alternate approach to improving the performance of RDMA-enabled key-value stores. The authors of this paper rely on programmable NIC hardware to create new one-sided RDMA primitive operations, each of which has some direct utility within a key-value store context. The new operations include get, put, delete, several vector-based atomic operations, and the vector-based operations reduce and filter. The authors focus on reducing the number of DMA requests per operation, hiding the latency incurred as RDMA verbs interact with the PCIe bus, and moving some resource load from the NIC to on-host memory in order to reduce resource pressure. Although KV-Direct exhibits good performance and provides useful new primitives, it is limited to environments where FPGAs are available. In contrast, Nessie implements RDMA-enabled key-value operations without any hardware beyond commodity RNICs.

Another recent post-Nessie work explores disaggregated persistent memory systems [117]. The authors of this work explore the impact of hosting devices without CPUs, but rather just network resources and persistent memory resources. They develop different architectures for accessing the memory on these devices, and build storage systems on top of those architectures. This can be viewed as an interesting extension of Nessie-style client-driven design, although Nessie's design was intended for more traditional environments, and it excels when its client-driven protocol saves CPU resources that can be used for other tasks (for example, computations on the data being stored). Another system, Octopus [63], builds an in-memory file system for non-volatile memory that uses RDMA writes and atomics to implement RPCs and distributed transactions. Similarly, Orion [133] explores non-volatile main memory and uses RDMA to build a distributed file system on top of it. In addition to a different interface, these systems use a mixture of client-driven and server-driven operations in contrast with Nessie's client-driven protocol.

Several papers have addressed reliability and transactions for RDMA-enabled storage

systems. A follow-up to FaRM [26] not only addresses reliability, but also shows that RDMA-based workloads can become CPU-bound at scale [27] (which means minimizing CPU use can be essential to guaranteeing high levels of performance). The authors of HERD [47] likewise created FaSTT [49] to demonstrate durable, serializable, distributed transactions. Although FaSST, like HERD, is server-driven, its authors estimated at the time that future implementations of RDMA would have efficient enough client-driven operations that they would outperform some server-driven operations (and therefore that using both will be necessary to achieve the best levels of performance). We discuss this in greater depth in Section 2.2.2. The DrTM [127] authors add additional reliability, availability, and performance improvements in two different follow-up papers [17, 126]. Tavakkol et al. [114] explore using RDMA for the construction of synchronous mirroring of persistent memory transactions through the addition of additional primitives. Tailwind [113] implements an RDMA-enabled replication recovery log for in-memory databases. NAM-DB [136] is a database that explores implementing scalable transactions using RDMA. Nessie does not address reliability and availability in our current implementation, however future research could study integrating the techniques pioneered by other systems.

Network Sidedness Performance

A recurring theme in recent post-Nessie research is studying the impacts of network sidedness, as defined in Section 2.1.6, on RDMA-enabled system design. Client-driven and server-driven operations can yield varying levels of performance, and RDMA's large feature set means there is debate about the best ways to achieve the best performance for different classes of applications. The authors of the Remote Fetching Paradigm (RFP) [105] attempt to classify the differences in performance between client-driven and server-driven systems. They note that issuing RDMA requests has about five times higher overhead than allowing the NIC to service incoming RDMA requests, raising an interesting tension between avoiding server-driven communication (which is limited by the outbound rate of replying servers) and avoiding the additional RDMA verbs required to provide synchronization for client-driven protocols. The m&m consensus protocol [3] is built on top of an RDMA framework that mixes the use of message passing (using server-driven operations) with shared-memory (using client-driven operations). The authors find that mixing both models improves fault tolerance and scalability versus strictly using one model or the other. Hemmatpour et al. [39] benchmark client-server request processing using different RDMA mechanisms, and find that client-driven RDMA writes for requests and client-driven RDMA reads for responses results in higher throughput and lower latency than alternatives as scale increases.

The authors of DrTM [127] have produced a follow-up paper [125] that examines and debates client-driven and server-driven designs. They come to the conclusion that generally client-driven designs perform better than server-driven designs (if the number of round trips is the same), that server-driven designs scale better when sending small amounts of data between large numbers of nodes, and that server-driven designs can be faster than client-driven designs if acknowledgement messages are moved off the critical path of an operation and piggybacked onto future requests. They use these lessons to build a distributed opportunistic concurrency control protocol that uses different primitives depending on the phase of the protocol (making it a hybrid design). They also note that their results depend on hardware specifics, as newer Mellanox RNICs make client-driven designs exclusively faster than server-driven designs in their testing (even when acknowledgement piggybacking is used for server-driven designs).

In recent work, the authors of HERD [47] and FaSST [49] have continued exploring server-driven designs. They introduce eRPC [50], a general purpose RPC library that they claim provides performance similar to specialized systems. This is done by providing direct access to and control over DMA-enabled buffers through the eRPC API, and optimizing for the authors' common case which consists of small messages, a congestion-free network, and short RPC handlers. While this system does not require RDMA (it works on UD RDMA transport types as well as UDP), it clearly advocates for server-driven designs, claiming that RDMA has "fundamental scalability limitation(s)" when operating in a connected manner. They attribute this to thrashing that occurs when the RNIC runs out of cache room for RDMA connection state. Although Nessie does not address RDMA's scalability issues on older hardware, it is specifically designed to work with large messages, does not require RPC handlers at all thanks to its strictly client-driven design, and also makes no assumptions about network congestion due to its use of reliable connections. This puts Nessie's target workloads outside of eRPC's common case.

Some authors also disagree with the stance advanced by eRPC [50]. A new system, Storm [77], demonstrates that RDMA reads over RC transport types on new Mellanox RNIC hardware outperform UD-based RDMA send and receive operations even when the RNIC caches are being thrashed by servicing large numbers of connections. The paper rebukes eRPC's push towards unconnected server-driven designs, and shows that the scalability of RDMA reads and writes is acceptable for rack-scale computation with up to 64 servers in a cluster. They further argue that RC transport types help performance by delegating congestion control to hardware, that their performance will only continue to improve in future hardware, and that therefore RC transport types should be used exclusively. The authors construct a hybrid system that combines client-driven RDMA reads (for simple lookups) with server-driven RPCs using RDMA writes (for more complex oper-

ations), and evaluate this system to back up their claims. The reality of the matter is that the best performance is likely derived based on application needs. Regardless of whether client-driven, server-driven, or hybrid designs end up being best for particular classes of systems, it is encouraging to know that the performance of Nessie's client-driven protocol, which uses RC transport types, will improve on current generation RNICs, and that the prognosis for its performance on future hardware looks to be even better.

2.2.3 RocketStreams

RocketStreams, outlined in Chapter 5, is a framework that allows for easy and efficient live streaming video dissemination. We identify three main categories of related work for RocketStreams: live streaming video research, research on RDMA-enabled systems, and networking frameworks that provide functionality similar to RocketStreams.

Live Streaming Video

Live streaming video is quickly growing in popularity. Over the last decade, a host of different live streaming video services have sprung up, for example Twitch [119] and YouTube Live [134]. Twitch in particular has had a major impact on HTTP streaming video, and as of 2015 was the fourth largest consumer of peak Internet traffic in the United States [30, 118], with 2.2 million unique content creators (or streamers) providing over 292 billion minutes worth of live streamed content in 2016 alone [32]. It is estimated that the amount of live streaming video delivered will grow by a factor of 15 between 2017 and 2022 [19]. By 2022, live streaming video is projected to account for 17 % of all Internet video traffic [19]. This explosive growth helps to motivate our work with RocketStreams.

In order to meet user demands for content, Twitch and similar services rely on geo-distributed data centres containing clusters of servers. A 2017 mapping of Twitch's content servers revealed around 875 unique content-providing servers, distributed across 21 regions (distinguished by airport codes) over four continents [23]. This same study found that a single live stream with only 30,000 viewers could dynamically be served by up to 150 different servers, implying that even a moderately sized stream could occupy multiple servers within the various physical locations that it is served from. These replicas are necessary to guarantee scalability for live streaming video services, which have strict real-time constraints to meet [23]. The real-time nature of live video streams makes content pre-caching at CDNs (a strategy used extensively by video on demand) impossible without impacting liveness. Furthermore, whereas a server delivering video on demand content to a

user can accommodate shifts in latency by filling a generously sized playback buffer as fast as network throughput allows, the buffering of a live streaming video is limited by the rate of the creation of the video itself (again, unless liveness is sacrificed). This is compounded by more demanding user expectations for live streaming video. Dobrian et al. [25] note that increased buffering times and low bit rates disproportionately affect live streaming video when compared with video on demand. Efficiently making use of resources so as to not affect the latency of content delivery is therefore a critical concern for live streaming video workloads.

In addition to providing basic live streaming video playback, many services offer other functionality. YouTube live streaming video, for example, supports trick mode during playback, allowing users to pause and rewind instantly. Twitch does not provide trick mode for live streams, although it does convert live streams to static video files which become available once a stream has ended. Until that time, however, users only have access to the latest, short-lived data in a live stream. For some live streams, this data may be available to users from Twitch at multiple different bit and frame rate encodings [20]. It is important to note that content creators do not upload their data at multiple bit and frame rates to take advantage of this feature — rather Twitch transcodes their live stream using dedicated transcode servers. Given that video transcoding is very CPU-intensive, and therefore transcode servers are expensive to provide, Twitch limits the availability of transcoding to higher traffic streamers and registered partners [112]. A final functionality provided by many modern services is encrypted content. Over 60 % of Internet traffic is now encrypted [97], and this percentage is growing.

Delivering HTTP live streaming video, most of which is done using TCP-based networking, has high CPU overheads in terms of data copying and protocol processing. One previous study found that, for their particular hardware configuration, the TCP/IP protocol stack consumed significant amounts of CPU and managed to saturate a computer's memory bus without using even 35 % of peak network bandwidth [8]. The same study showed that RDMA generated about one fourth the memory traffic, and RNICs have improved dramatically in the time since that study was conducted. Even when TCP offload is used to move the processing of the kernel stack directly onto the NIC's hardware, other networking technologies like RDMA still provide better resource utilization and performance, while also allowing for additional utility like atomic and one-sided network operations. This is partly due to the fact that TCP offload engines are constrained by the operating system's programming interface, which renders certain parts of the sockets layer difficult to make zero-copy [8]. One of the benefits of RocketStreams is that it enables the use of RDMA to reduce CPU utilization.

Several studies have looked at other techniques for improving the efficiency of systems

serving live streaming video. Pires et al. [84] examine strategies for selecting Twitch streams to be bit rate transcoded in order to reduce overall service resource consumption. Although this strategy is not directly intended to improve CPU utilization (it focuses primarily on network bandwidth), a reduction in bandwidth also reduces the amount of CPU required to copy video data into socket buffers when sending it to clients. Similarly, He et al. [36] focus on transcode server allocation, attempting to find the most efficient way to maximize viewer satisfaction while minimizing the costs associated with transcoding. A follow-up to this work [37] suggests offloading transcoding from the service itself to the edge, taking advantage of viewer hardware to help reduce CPU utilization of the service. Netflix has examined several ways to reduce CPU utilization per client connection on HTTP streaming video web servers by moving TLS encryption into the FreeBSD kernel [101, 102]. Although this was done in the context of video on demand, similar benefits would also apply to live streaming video. We believe that the techniques proposed by existing work are valuable and complementary to RocketStreams' goal of making live streaming video services more CPU-efficient.

There have been multiple studies to characterize the workload details of live streaming video services. Pires et al. [85] provide information on both YouTube Live and Twitch, including the amounts of bandwidth consumed by both services, diurnal viewership analysis, and access patterns for content. They find, for example, that both services exceed traffic peaks above 1 Tb/s, and that both services have fewer diurnal changes in traffic than other services that provide user generated content. Similarly, Zhang et al. [137] break down some of the characteristics of Twitch broadcasting, showing that Twitch live streams can last much longer than video on demand content and that there are extreme skews in stream popularity, with 0.5 % of broadcasters sometimes contributing more than 70 % of total views (and sometimes even more than 90 % of total views). These forms of extreme skew are confirmed by Kaytoue et al. [53], and make sense given the overwhelming tendency of viewers to gravitate towards large events. Claypool et al. [20] provide comprehensive details on stream lengths, frame rates, and bit rates for Twitch, and Stohr et al. [103] provide a similar analysis of traffic patterns for the YouNow live streaming service. We use this information to help evaluate our RocketStreams implementation.

RDMA

RDMA can be used by RocketStreams to implement efficient data dissemination. The body of relevant RDMA research related to RocketStreams is predominantly similar to the body of RDMA research related to Nessie, albeit the components of the related work that RocketStreams compares to differs from Nessie. A detailed breakdown of RDMA-related

research is provided in Section 2.2.2.

For RocketStreams' purposes, it is primarily comparable to systems that use RDMA to move data for more complex interfaces. In this respect, RocketStreams is comparable to systems like RaMP [67] and Rocksteady [55], or RDMA-enabled Spark [62]. It would also be comparable to systems that implement RDMA underneath a file system-like interface, such as Remote Regions [2], Octopus [63], or Orion [133]. A final good comparison would be to systems that enable a key-value store or distributed memory interface including RDMA-enabled Nessie [15], memcached [46, 45, 104], Pilaf [70], FaRM [26] (and follow-ups), DrTM [127] (and follow-ups), RStore [115], HERD [47] (and follow-ups), or KV-Direct [57].

The previously mentioned systems do not fit the exact use case for RocketStreams. For example, data migration interfaces like RaMP and Rocksteady do not directly support a dissemination abstraction, and may not provide networking alternatives such as TCP (and plans to incorporate additional features such as multicast support and different RDMA mechanisms in the future). Key-value store interfaces may not allow for granular control over data placement, and even when they do allow for such placement they do not account for the type of data dissemination required by RocketStreams (which requires live streaming video data to be replicated among potentially many nodes). Several of these systems, such as FaRM, Octopus, DrTM, and Orion, require operating system modifications or hardware outside of the scope of system that RocketStreams is designed for. In general, while all of these systems contain important lessons and techniques that could be leveraged to improve RocketStreams' RDMA-based performance, the contribution of RocketStreams is enabling easy and efficient live streaming video dissemination, and it achieves this using networking techniques tailored towards high performance for live streaming video dissemination.

Networking Frameworks

Because RocketStreams uses an underlying network component to send data, it draws some comparison to frameworks such as libfabric [35] and libevent [64]. The libevent [64] library provides an interface for access to event queues for file descriptors. The main benefit of this is that some high-performance mechanisms used to access file descriptors (for example, epoll and kqueue) can be platform-dependent. Not only does libevent provide a unified interface for accessing file descriptors (and therefore network data) across operating systems (and different versions of the same operating system), but it also reduces the amount of boilerplate code required by applications that wish to use scalable and high-performance file descriptor event queues.

In a similar manner, libfabric [35] provides a platform-independent abstraction for networking. The libfabric library allows applications to discover the types of communication fabrics available to them, establish connections between logical nodes over communication channels, transfer data over connections (and between connectionless endpoints), and finally to learn about transfer completion through events. Its APIs are intended to be efficient and to simplify application use of networking hardware. Although libraries like libevent and libfabric are useful, they do not operate at the same level as RocketStreams. They act as an abstraction over top of the networking layer (or in libevent's case, file descriptors in general), and do not provide the automatic buffer management and dissemination that RocketStreams enables for live streaming video. Whereas an application using libfabric or libevent must still determine the best mechanisms for providing live streaming video dissemination (and manually employ them), this process is purposefully hidden from applications and managed by RocketStreams to reduce the amount of code required by the application. RocketStreams could use either library in the implementation of its own networking layer, but is currently built on top of infrastructure developed initially for Nessie, due to our familiarity with this code base.

Finally, the WILSP platform [91] provides a framework for disseminating and delivering live streaming video. Under WILSP, producer nodes forward data to a Redis [87] dissemination layer. Data is then forwarded to web server delivery nodes, and in turn to users. WILSP is built primarily for interactive video, however (for example, this would include video streams for online learning environments). This means that they are most concerned with low-latency, and small numbers of users (relative to the scale at which RocketStreams wishes to operate). The WILSP study demonstrates web servers that serve video to a maximum of 50 users, representing a total of about 230 Mb/s of delivery throughput. RocketStreams is designed for efficient and highly scalable live streaming video, and is benchmarked delivering to user numbers orders of magnitude higher than this.

2.3 Chapter Summary

In this section we provide background elements related to our research projects. We discuss HTTP streaming video and differentiate between video on demand and live streaming video, the primary workloads for Libception and RocketStreams respectively. We discuss the important disk-based techniques that Libception uses to improve disk throughput for web servers delivering HTTP video on demand, namely serialization and prefetching. We discuss disk block I/O schedulers and their relationship to disk request scheduling. We describe RDMA, a networking technology used by Nessie and RocketStreams to achieve

high performance and reduce CPU utilization. We then elaborate on network sidedness and data coupling. Finally, we explain the hashing algorithm used by Nessie, cuckoo hashing.

We next provide an overview of research related to our own. We begin by discussing research related to Libception, namely characterization of video on demand and techniques for improving disk throughput. We discuss workload details for benchmarking Libception, and explain that Libception provides a unique opportunity to combine and study multiple performance-improving techniques without the need to modify code. Next, we discuss work related to our implementation and evaluation of Nessie. We examine existing research on RDMA, RDMA-enabled storage systems, and recent (post-Nessie) research into network sidedness and its effects on performance. We find no work, however, that evaluates the same type of strictly client-driven key-value store protocol provided by Nessie. Finally, we discuss work related to the design of RocketStreams. Research into live streaming video motivates the design of RocketStreams, and gives us indications of workload properties for benchmarking our framework. A review of RDMA-enabled systems and other existing networking frameworks reveals no systems designed specifically to address the dissemination of live streaming video that RocketStreams wishes to facilitate.

Chapter 3

Libception

3.1 Introduction

Video streaming over HTTP is now the largest contributor to Internet traffic, having represented 75 % of all traffic in 2017 [19]. This total is expected to rise to 82 % by 2022 [19], as user adoption rates increase and the catalogue of available content from popular video streaming services grows. Although memory caching and SSDs can be effective for storing the most popular content, many HTTP streaming video services contain large amounts of less-popular content that are only cost-effective to serve from hard drives. Serving the long tail of content is therefore often disk-bound [51, 106]. Techniques for improving disk throughput in such systems are therefore of considerable interest.

There has been much past work on improving disk access efficiency. In the context of HTTP video streaming, however, there are two complicating factors. First, unlike early video streaming systems, HTTP-based video streaming is pull-based: the server responds to client requests for video chunks, rather than pushing video data to the client at some server-determined rate [9]. Second, contemporary servers may be highly concurrent, responding to video chunk requests from hundreds or thousands of clients concurrently [106].

A well-known approach to making disk access more efficient for applications that access files sequentially is to perform larger reads from disk to prefetch data before it has been requested. Prior work observed that serializing reads may also be important in some contexts [108, 109, 107]. By modifying a research web server, the userver, to both aggressively prefetch and serialize its reads, performance was significantly improved on FreeBSD. This prior work requires the use of the FreeBSD-exclusive `SF_NODISKIO` flag for the `sendfile`

system call, alongside non-trivial code changes. As such, it did not investigate whether the approach of combining aggressive prefetching with serialization could yield similarly large benefits for more widely used web servers (for example, nginx and Apache) or on other operating systems such as Linux.

Implementing techniques for improving disk access efficiency inside the application requires detailed knowledge of the application code, and must be repeated for each application of interest. This could be quite difficult for applications with large code bases such as Apache [4] and nginx [76]. On the other hand, a kernel implementation is operating system-specific, requires detailed knowledge of the relevant pieces of kernel code, and has the additional problem of potential adverse impacts on other types of applications.

In this chapter, we address the problem of improving disk access efficiency in popular HTTP video streaming servers (nginx and Apache) and a custom web server (the userver) by implementing a portable library shim, Libception, that can introduce serialization and prefetching to unmodified applications on multiple operating systems (FreeBSD, Linux, and macOS). Using Libception, we explore the impact of aggressive prefetching and serialization, together and in isolation, on all three web servers on Linux. We show that Libception allows the same types of performance improvements as previous research [107], albeit for other unmodified applications, and without being restricted to FreeBSD.

3.2 Design

The Libception library is portable, operates in user space, and has been tested on FreeBSD, Linux and macOS. It is comprised of two components: the first component is Deception, a dynamically linked shared object which inserts itself between the application and C standard library calls. Deception intercepts I/O requests from the application and forwards them to the system's second component, Reception. Reception is a server process that runs separately from applications using Deception. Reception receives, services, and responds to requests generated by applications using Deception (including prefetching data when necessary). Figure 3.1 shows a high-level overview of the components of Libception.

3.2.1 Deception

Deception is the primary interface for communication between user applications and the Libception library. It is implemented as a shared object that is dynamically linked at launch time by the application that wishes to make use of it. On Unix-based operating

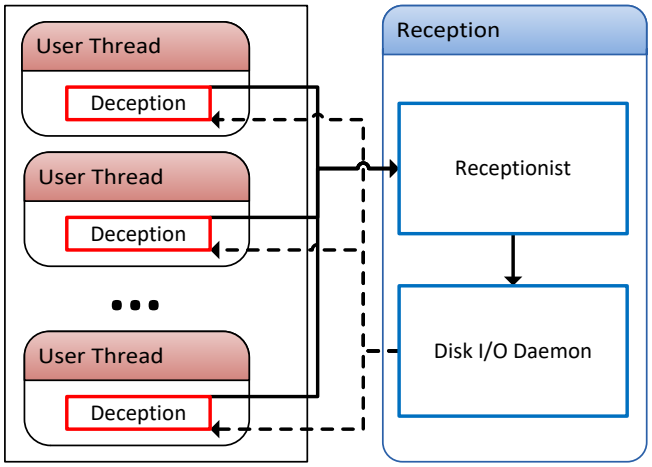

Figure 3.1: An overview of the components of Libception. I/O calls are redirected by the Deception shim to the external Reception process. DiskIODaemons then service and respond to the request.

systems, including Linux and FreeBSD, this is done by setting the `LD_PRELOAD` environment variable when launching an application. Likewise, on macOS this is accomplished by setting the `DYLD_INSERT_LIBRARIES` environment variable. Once loaded, Deception begins silently intercepting calls to a variety of I/O-related functions including `open`, `read`, and `sendfile`. This technique allows applications to take advantage of the Libception library's benefits without requiring any source code modifications. Furthermore, because Deception is implemented in user space, it requires no changes to the underlying operating system.

Most calls to Deception perform several validity checks, and then determine whether or not the application's I/O request is already resident in the system's file cache. This is done by using the functions `mmap` and `mincore` to check all memory blocks of the request (excluding blocks that would be prefetched if the request went to disk). If a request is found to be contained entirely in memory, Deception passes the call to the original function (which returns without needing to go to disk). Otherwise, disk I/O is required, and a Libception I/O request is constructed and sent to Reception using Unix domain sockets. Unix domain sockets allow receiving processes to obtain file descriptors from sending processes, and Reception does this in order to get access to the file descriptor that will be used for prefetching and serialization purposes.

Should a disk read be necessary, Deception waits on a response from Reception before it proceeds. To ensure transparency, Deception always finishes by executing the application's original C standard library call, and returning the resultant return code (even if a parameter input to Libception fails sanity checks). This allows Deception to run invisibly, without affecting the guarantees of the API for the associated C library call. This in turn means that the application can expect the same control and user-functionality for I/O calls that it would obtain if Libception was not being employed.

Some additional non-I/O C standard library functions are also intercepted by Deception (in particular `fork`). These functions are intercepted for functionality purposes: in order to support processes that use Libception from multiple thread contexts, Deception uses locks to synchronize access to its internal data structures. Intercepting `fork` allows Deception to guarantee that locks are not owned in other thread contexts when the `fork` system call is issued (as this would result in undefined behaviour). These functions do not communicate with the Reception layer, and are also invisible to the user (as they complete by making the original C standard library calls as well). All initialization for Deception is handled transparently at application launch time by constructor attribute functions that are invoked prior to `main`, and all cleanup is likewise handled unobtrusively at termination by destructor attribute functions which are invoked after `main` returns.

The basic design principles for Libception are also applicable to other operating systems, such as Microsoft Windows. Although the process for implementing these techniques on Windows are more complicated than setting an environment variable, there are several options for re-routing the WinAPI calls that deal with disk I/O. For example, a Windows-based version of Libception could use Microsoft Detours [68] to forward I/O calls to an external Reception process. We leave the details of such an implementation to future work.

3.2.2 Reception

In our experiments, Reception is launched as a separate process in user space, prior to running applications with the Deception shim. It could also be launched as a daemon process in a production environment. Reception is primarily responsible for accepting, serializing and servicing incoming I/O requests from one or more Deception shims. Reception furthermore modifies requests as necessary (for example, by enlarging read sizes to introduce prefetching), executes the disk request itself, and finally responds to Deception. These tasks are divided between a single Receptionist thread, and one or more DiskIODaemon threads.

The Receptionist thread acts as a server, accepting requests over Unix domain sockets.

I/O requests received by the Receptionist thread are sorted by device, and are placed in one of multiple queues to be serviced by the appropriate DiskIODaemon thread. As Deception and Reception both keep track of statistics for performance and diagnostics purposes, the Receptionist also has the ability to push messages containing statistics to a separate thread that performs data aggregation.

DiskIODaemon threads are responsible for performing I/O for different devices. By default, Reception runs in a single DiskIODaemon mode, with all requests being serviced sequentially in the system by one thread. Alternatively, the user may set Reception to create individual DiskIODaemon threads for individual devices in the system. A single DiskIODaemon thread is used per device to ensure that requests to the device are serialized. In this case, the Receptionist assigns each incoming request to the appropriate DiskIODaemon thread based on the underlying device for the request's file descriptor.

Regardless of which mode is selected, DiskIODaemon threads each use their own lock-protected request queue, which is filled by the Receptionist, and drained by the DiskIODaemon thread. Only one request is removed from this queue at a time by the DiskIODaemon thread, ensuring that I/O to whichever device it is servicing is serialized. As requests are pulled from this queue, they are expanded to include any necessary prefetch information, and are then sent to the disk. Once finished servicing the request, the DiskIODaemon thread sends a message to the Deception shim that made the request, allowing the calling process to unblock and proceed.

Libception additionally contains options that allow it to perform prefetch-free serialization, serialization-free prefetching, and simple request tracking without either serialization or prefetching (useful, for instance, in latency profiling or other statistics gathering). Serialization-only mode indicates to Libception not to extend reads provided by the user and is identical to prefetching with a prefetch size of zero. Prefetch-only mode transfers the burden of extending and performing requests from Reception to Deception. In this mode, instead of communicating with Reception, Deception shims immediately extend their requests and perform application-side positional `read` calls before completing the original I/O request.

3.3 Methodology

Here we briefly describe the methodology we use for generating experimental workloads and benchmarks. The values used in this section are taken from several studies, and are described in detail in previous work that combined them to create this experimental

methodology [108]. Our workload represents a large number of HTTP streaming video clients requesting videos with characteristics similar to requests for YouTube videos in 2011 [29]. We use a small number of client machines to generate traffic simulating thousands of sessions. Each session represents an end user viewing video. The video selected for each session is chosen using a Zipf distribution [34] with an α value of 0.8 [108]. This video is watched for some fraction of its duration. It is an important characteristic of video workloads that users do not typically watch to the end of a video, so this property is reflected in our workloads: the percentage of a video that any given user requests is in line with the watching patterns of a typical YouTube-like workload [29].

Another important characteristic of our workload is that the network is not the primary bottleneck. Netflix, for example, uses Open Connect Appliance (OCA) servers that have operational throughputs between 9 Gb/s and 36 Gb/s [73], but are provisioned with up to 40 Gb/s of network capacity [75]. Instead, the workload is heavily disk-bound. This can be seen in the findings of recent work characterizing the properties of a Netflix video workload [106]. Video providers maintain a large catalogue of content in order to appeal to a broad audience. Netflix, for instance, has a catalogue which is approximately 2 PB in size [106]. Because much of this is long-tail content that is infrequently viewed, cost-effective storage is achieved by storing these videos on large, inexpensive hard disk drives from which they are typically served.

Videos are stored on the server hard drives by storing each video in a separate file. We stored data for a video in a single file rather than multiple chunks because this approach is shown to be more efficient in prior work [109]. Furthermore, this is the approach used by video services like Netflix (with individual files used for separate bit rates) [106]. The disk is populated with 20,000 video files which have an average duration of 265 seconds, with a similar distribution as YouTube videos [29]. The video files have a fixed bit rate of 420 kb/s. This bit rate was chosen based on information available at the time of creation of the benchmark [34]. With these duration and bit rate characteristics, our average file size is 13 MB.

Each client session consists of a sequence of requests for 10 second intervals of video data, which are 0.5 MB in size. The first three requests in a session are made as quickly as the server can deliver the results, then subsequent requests are made in fixed 10 second intervals. This represents the filling of a playout buffer at the beginning of a session, followed by requests to refill the buffer as it is consumed at the bit rate of the video. This is a simplified model of a pull-based video client; it does not attempt to represent user actions like pausing the video, skipping to different points in the playback, or changing the quality level of the video. These actions were rare in the YouTube workload we modelled, but our methodology is flexible enough that we could represent these actions for workloads

where they are significant.

The average duration of a video session is 160 seconds, using a distribution derived from real-world measurements. During experiments, video sessions are started at a chosen rate, using a Poisson distribution for session initiation. Experiments consist of 14,400 sessions, with a maximum of about 650 concurrent sessions. A total of 118 GB of video data is requested from 6,581 different videos. Each video is viewed 2.2 times on average and 67.8 % of videos are requested a single time during the experiment.

The clients monitor the service time for each request, and if it takes longer than 10 seconds to completely receive the data from a request, the client terminates the session and stops making further requests. This mimics a client depleting its playout buffer, and the viewer stopping their session. For our experiments, we are interested in determining the highest aggregate client request rate (the total number of viewers) that can be serviced, so that we can compare different web servers and configurations. To determine this rate, the *maximum failure-free rate*, we conduct a number of benchmark runs with a range of different aggregate rates of requests. For each configuration, we determine the highest rate that results in fewer than 0.3 % session failures. This percentage was chosen to permit clients to perform a non-zero (but very limited) amount of re-buffering.

The clients are connected to the server over a local-area network with high bandwidth and low delay. To better represent the conditions available to real-world users, we use dummynet [90], which allows us to simulate different network types. We throttle 50 % of client sessions in the workload to 3.5 Mb/s, and the other 50 % of client sessions to 10 Mb/s, in order to represent a mix of end-user cable and DSL access speeds. Furthermore, we add 50 ms of delay to the network in each direction in order to model wide-area network conditions (approximating transmit time for coast-to-coast traffic in North America).

The equipment and environments we use to conduct our experiments were selected to ensure that network and processor resources are not a limiting factor in the experiments. We use two server machines, one for FreeBSD experiments and the other for Linux experiments. Both are HP DL380 G5 systems which contain two four-core Intel E5400 2.8 GHz processors and 8 GB of RAM. The Linux system uses Ubuntu 12.04 with a Linux 3.2.0 kernel, and a Western Digital Red (WDC WD10EFRX) 1 TB 5,400 rpm 3.5 inch SATA3 disk to store video files (chosen for its combination of relatively high throughput and low power consumption). The FreeBSD system uses FreeBSD 8.0 and stores videos on an HP 146 GB 10,000 rpm 2.5 inch SAS disk. Table 3.1 shows the raw throughput and seek times for the two different drives, obtained using the diskinfo command. Note that these are raw throughput numbers that do not include file system overhead so actual application throughput numbers will be lower. We use FreeBSD 8.0 so we can try to match the per-

formance obtained previously with a modified web server (the userver) [108, 107]. Video files accessed during experiments are stored on a separate disk from the operating system.

Drive	Throughput (MB/s)			Seek Time (ms)		
	Outer	Middle	Inner	Full	1/2	1/4
HP	122	107	73	10.5	8.0	6.8
WD Red	133	115	67	32.6	23.1	18.9

Table 3.1: Raw disk throughput and seek times obtained using the FreeBSD diskinfo command.

In all of our experiments four client machines are used to generate load mimicking thousands of viewers. Each of the client systems contains either dual 2.4 GHz or dual 2.8 GHz Xeon processors and 2 GB or 3 GB of memory. Client machines run Ubuntu 10.04 and version 2.6.32-30 of the Linux kernel. They also use a version of httperf [72] that was modified locally to support new features in a workload generation module named wsesslog. These modifications also allow the clients to track additional statistics. All clients are connected to the server via multiple 1 Gb/s network links through multiple 24-port switches, helping to further ensure that the network is not a bottleneck.

3.4 Evaluation

We now evaluate the maximum failure-free throughput (also referred to henceforth as just throughput) of the Apache, nginx, and userver web servers while utilizing Libception on FreeBSD and Linux. In all cases, we have tuned the web servers to the best of our ability so that they provide their maximum failure-free throughput. Unless otherwise specified, the prefetch size used by Libception is 2 MB (we examine other sizes in Section 3.4.4 and Section 3.4.5).

3.4.1 FreeBSD

Previous work [108, 109] demonstrated how modifications to the userver web server to perform asynchronous serialization and aggressive prefetching (ASAP) within the application significantly increased server throughput when servicing streaming video workloads. Unfortunately, these benefits rely on modifying the web server to use the `SF_NODISKIO` option [93] to the `sendfile` system call which is only available on FreeBSD. This flag causes `sendfile` calls that would block on disk I/O to instead return `EBUSY`.

The basic architecture of the userver using ASAP is to have a separate thread which performs large disk reads (thus implementing asynchronous, serialized, aggressive prefetching). This was relatively straightforward in the userver because it integrates well with its event-driven architecture and its code base is relatively small, making it easier to work with and modify.

In this section we are interested in providing similar benefits to the more widely used Apache and nginx web servers without modifying the source code of either application. Note that nginx running on FreeBSD is of interest because the servers in the Netflix Open Connect Content Delivery Network use nginx on FreeBSD [74]. This is particularly relevant because Netflix currently accounts for a large fraction (35.2 %) of peak Internet traffic in North America [97].

We avoid making code modifications to the web servers because they each have a much larger code base than the userver and because Apache uses a significantly different software architecture (thread-per connection) [82]. Additionally, we want to determine if Libception can improve web server performance without using the non-portable `SF_NODISKIO` option to the `sendfile` system call.

Figure 3.2 shows the maximum failure-free throughput obtained when using each of the Apache (labelled "A"), nginx (labelled "N") and userver (labelled "U") web servers. Throughput is shown without Libception (labelled "Vanilla"), when using Libception (labelled "Libception"), and for the modified version of the userver that uses the `SF_NODISKIO` option (labelled "ASAP"). Furthermore, this graph shows both the disk throughput and the web server throughput as observed by all of the client machines.

These results show that Libception is able to more than double disk throughput and total server throughput for all three web servers. As well, when using Libception, the maximum failure-free throughput obtained by each server is equal to that obtained by the modified ASAP userver. This is despite Libception's use of `mincore` prior to each call to `sendfile` (rather than relying on the non-portable `SF_NODISKIO` option) to determine whether or not data needs to be prefetched before calling `sendfile`. Although previous work reports that `mincore` call overhead can be expensive [92], these video server workloads are disk-bound and can therefore easily tolerate the relatively minor increase in CPU overhead.

It is worth pointing out that, in these experiments, not all of the data that is requested needs to be read from disk. For example, two clients requesting the same video content in quick succession will only require the data to be read from disk one time. Therefore, the difference between the total server throughput and the disk throughput is due to file system cache hits.

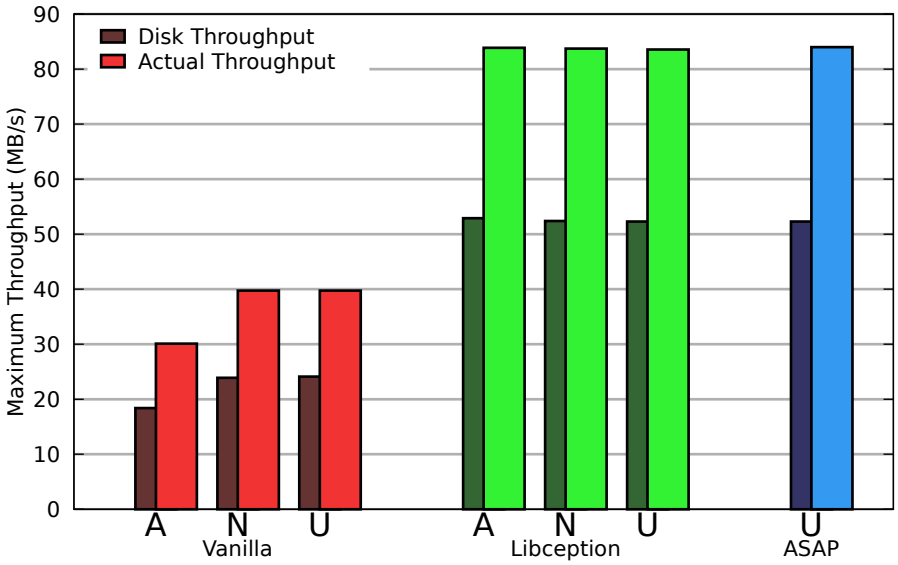

Figure 3.2: Web server throughput on FreeBSD without modifications, with Libception, and with ASAP (HP drive).

To the best of our knowledge, the version of FreeBSD used for these experiments does not provide any options to control the block I/O scheduler. There were also relatively few options available to influence kernel prefetching decisions and we were not able to significantly improve throughput using kernel parameters.

3.4.2 Linux

As noted previously, one of the key goals of Libception is to provide improved throughput for HTTP video web servers using techniques that are portable across different Unix-based operating systems. As a result, we now examine the performance of Apache, nginx and the userver on Linux. Recall that the disks used on the FreeBSD and Linux systems are different so we cannot directly compare performance across the different operating systems.

Figure 3.3 shows the disk throughput and maximum failure-free throughput obtained using each of the different web servers on Linux running with and without Libception. As was the case for FreeBSD, Libception again provides significant improvements in disk and server throughput. On Linux, server throughput is increased by a factor of about 2.5 times when using Libception.

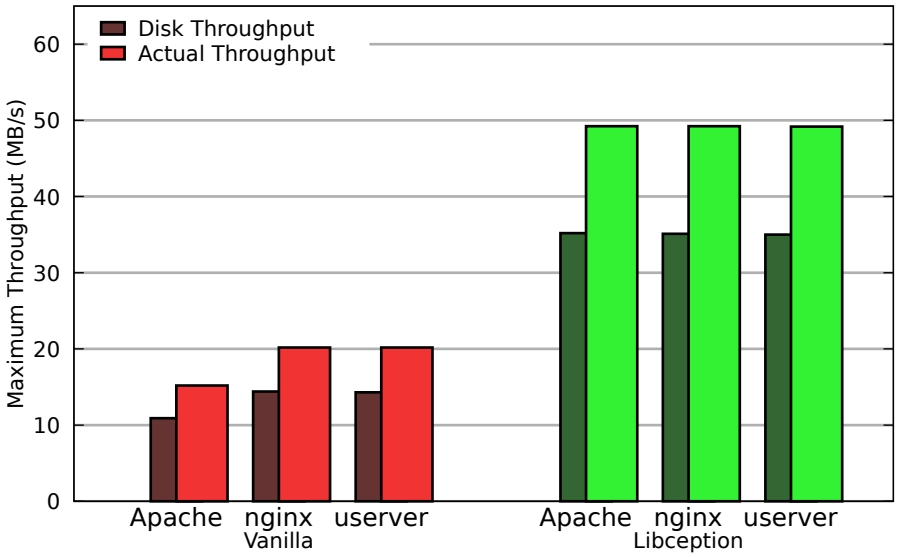

Figure 3.3: Web server throughput on Linux without Libception and with Libception (WD drive).

To our surprise, despite years of research on prefetching techniques in operating systems, these web servers and workloads do not appear to perform well on FreeBSD and Linux.

3.4.3 Block I/O Schedulers

The default Linux configuration on our system uses the CFQ block I/O scheduler [7]. We expected that the anticipatory nature of the Linux CFQ scheduler might be well-suited to this workload. Because web servers simultaneously process requests from thousands of clients, we hypothesized that blocks from different requests might provide reordering opportunities that could be exploited by CFQ to improve disk and server throughput. For completeness we now examine the performance of all web servers with each of the three block I/O schedulers available in Linux. Figure 3.4 shows the throughput obtained without Libception while using the CFQ (labelled "C"), deadline (labelled "D") and NOOP (labelled "N") schedulers. Figure 3.5 shows the results obtained using the same schedulers but this time while using Libception.

Interestingly, Figure 3.4 shows that without Libception the server throughput is slightly higher with the deadline and NOOP schedulers than with the CFQ scheduler. On the other

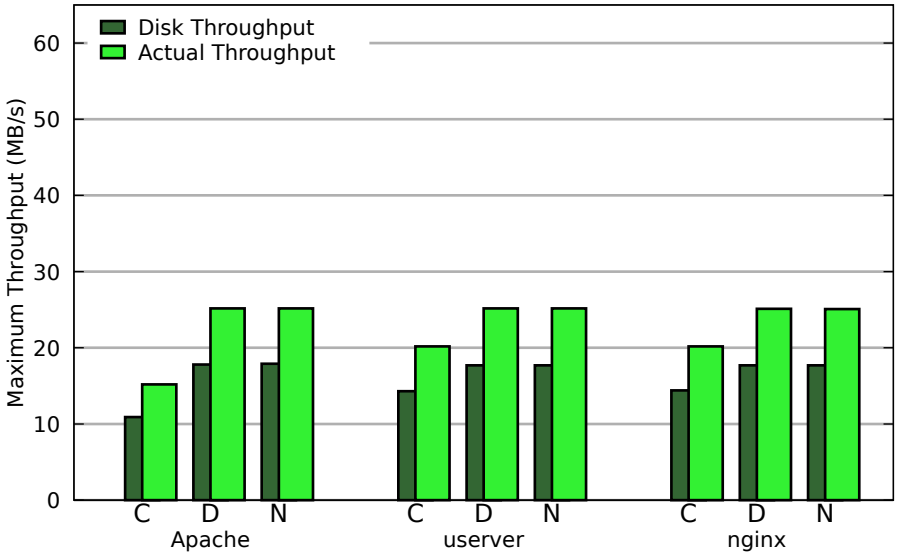

Figure 3.4: Throughput of web servers without Libception using different Linux block I/O schedulers (WD drive).

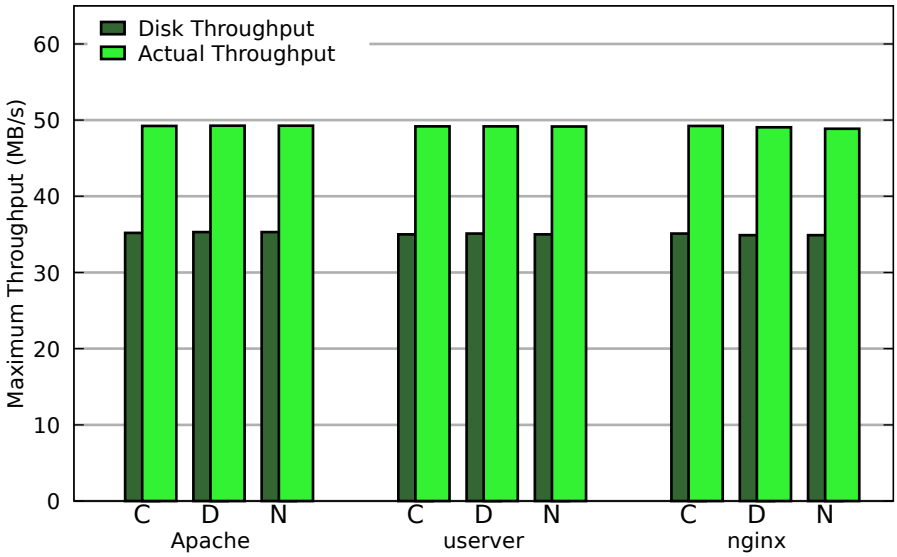

Figure 3.5: Throughput of web servers with Libception using different Linux block I/O schedulers (WD drive).

hand, when using Libception (see Figure 3.5) all web servers obtain the same maximum failure-free throughput of nearly 50 MB/s regardless of the block I/O scheduler used. We believe that this is because Libception is serializing all of the reads that go to disk and as a result the schedulers only ever see a single outstanding request, which leaves no room for scheduling policies to make a difference. Note that we have spent some time attempting to adjust the parameters designed to control the behaviour of the deadline and CFQ block I/O schedulers. We did not see throughput improvements when compared with the default values.

In summary, the different Linux block I/O schedulers do not significantly improve web server throughput for this workload. More importantly, Libception provides significant increases in throughput that are not possible using the block I/O scheduling algorithms.

3.4.4 Insights

In this section we first conduct a sequence of experiments designed to understand the relative importance of the prefetching and serialization components of Libception. For the remainder of the section we focus solely on the nginx server and the CFQ block I/O scheduler. We chose nginx because it is used by Netflix for serving HTTP video streaming workloads. We chose CFQ because it is the default block I/O scheduler on our Linux server and because the schedulers did not significantly affect performance when using Libception (see Figure 3.5).

Figure 3.6 shows the maximum failure-free throughput obtained using nginx without Libception (labelled "Vanilla"), with Libception using only serialization (labelled "Serialized"), with Libception using only prefetching (labelled "Prefetching"), and with Libception using both serialization and prefetching (labelled "Libception"). As can be seen in this figure, serialization alone actually reduces throughput. We believe that this is primarily due to the relatively small size of reads that the application performs. These small reads, in conjunction with serialization, result in small requests being issued one at a time, causing very poor performance. On the other hand, using prefetching without serialization significantly increases both disk and server throughput when compared with the "Vanilla" server. Finally, by combining both aggressive prefetching and serialization, a further increase of approximately 25 % beyond that of prefetching alone is achieved. These experiments demonstrate that, while aggressive prefetches are essential, the full potential of Libception is not realized unless the requests to the disk are serialized.

In previous work [108, 109] and in all experiments in this chapter up to this point, the prefetch size was set to 2 MB. We now examine a range of prefetch sizes and study the

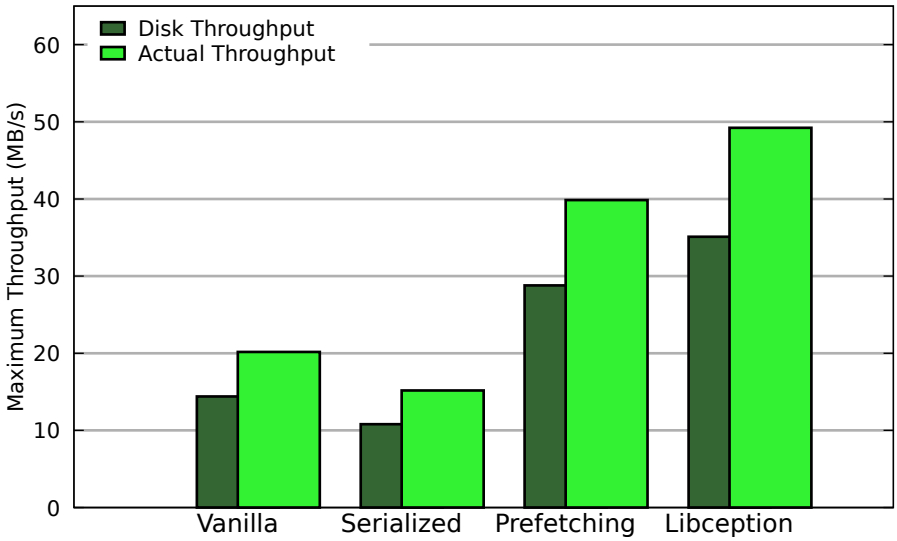

Figure 3.6: Summary of nginx throughput with different configurations on Linux (WD drive).

impact on the throughput of nginx while using Libception with prefetching but without serialization (see Figure 3.7) and with both prefetching and serialization (see Figure 3.8).

Figure 3.7 shows that, without serialization, throughput does not improve until a prefetch size of 2 MB or greater is used. It also demonstrates that prefetch sizes of 3 MB and 4 MB provide slightly better throughput than a prefetch size of 2 MB.

Figure 3.8 shows that when using Libception with both serialization and prefetching, small prefetch reads actually slightly reduce server throughput. However a prefetch size of 1 MB significantly improves server throughput, which is not the case when prefetching is used without serialization. When using both serialization and prefetching, throughput peaks with prefetch sizes of 2 MB to 4 MB. Additionally, these results show that serialization provides additional benefits (about 10 %) when compared with prefetching alone.

3.4.5 Tuning Linux

We now use the insights obtained from Section 3.4.4 to modify Linux kernel parameters in an attempt to improve the performance of nginx when running on Linux without the use of Libception. The question being examined is: can we find and tune appropriate

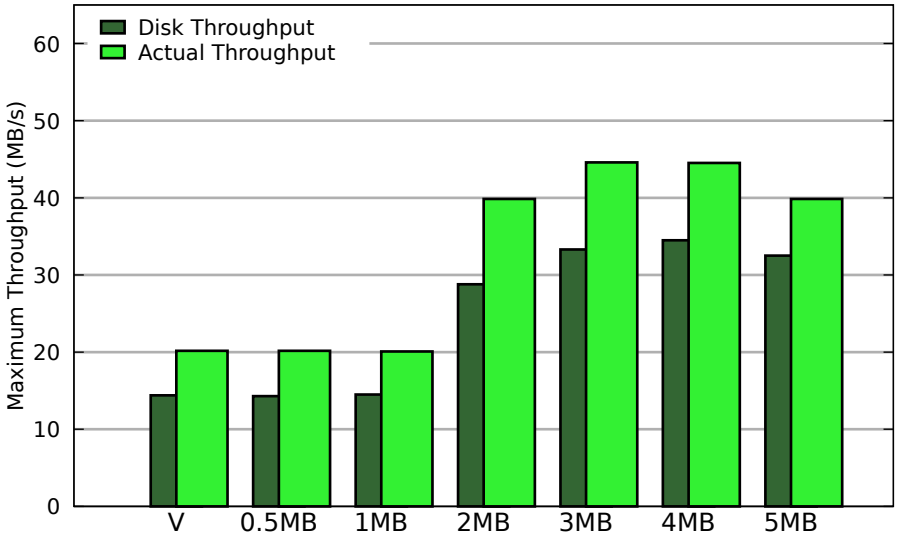

Figure 3.7: Throughput of nginx on Linux with Libception using various prefetch sizes and no serialization (WD drive).

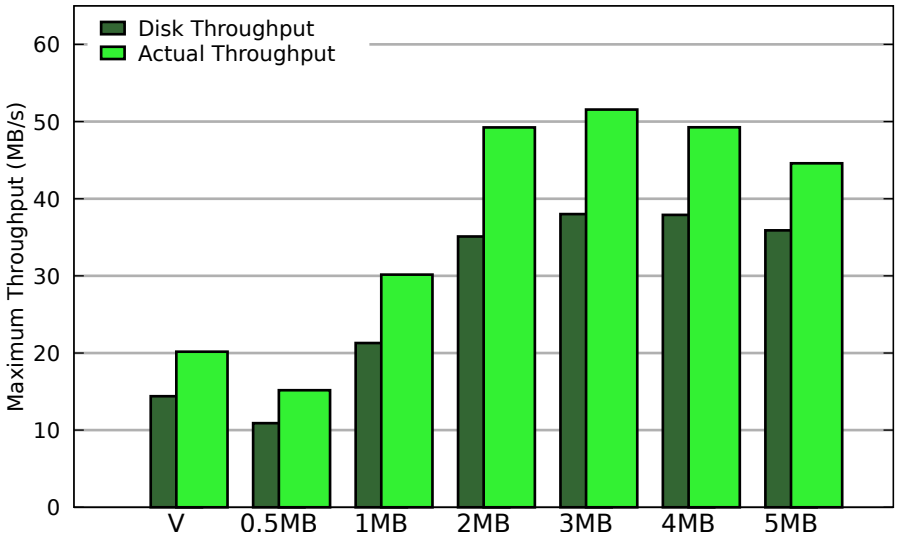

Figure 3.8: Throughput of nginx on Linux with Libception using various prefetch sizes and serialization (WD drive).

Linux kernel parameters in order to obtain throughput that equals that obtained using Libception?

It was not too difficult to find a Linux kernel parameter that we were able to adjust to increase the amount of data being read from disk by the kernel. When obtaining data from disk the Linux kernel may extend the read beyond what the user has requested (depending on several factors, the details of which are not relevant to this discussion). Roughly speaking, a readahead size is tracked per file and under proper conditions grows for each successive read. However, the maximum readahead size for all files on a device is limited by the kernel parameter `read_ahead_kb`, which can be set to a different value for each device. The default setting for this parameter in the version of Linux used in our experiments is 128 kB. This is quite small compared to the aggressive prefetches we use in Libception (2 MB in many cases) in order to obtain significant increases in throughput.

Figure 3.9 demonstrates how the throughput of nginx changes as the value of the readahead parameter is increased. This is done using the blockdev utility with the `--setra flag`. For example, to set the readahead size to 1,024 kB on the disk drive (`/dev/sdb1`) used to store video files in our experiments, we use the command `blockdev --setra 1024 /dev/sdb1`. As can be seen in the graph, `read_ahead_kb` values of 0.5 MB and 1 MB provide significant improvements over the default value of 128 kB (labelled "V" for Vanilla). Larger values of `read_ahead_kb` do not perform as well as 1 MB. While the throughput obtained with 1 MB readahead is about 45 MB/s, it is not as high as that obtained using Libception (50 MB/s). Understanding why this is the case involved significantly more work.

Although the sizes of the readaheads for files were actually reaching the values we set using `read_ahead_kb`, using the Linux blktrace utility we were able to determine that physical requests to the disk were being fragmented, with a size limited to 512 kB per fragment. We believe that, as a result, some requests for reads to different files were being interleaved (this is similar to the Libception case where prefetching is used but serialization is not).

By examining the Linux kernel source we were eventually able to determine that another kernel parameter was placing further limits on the size of reads. This value, `/sys/block/sdb/queue/max_sectors_kb` (for the disk `/dev/sdb`), has a default size of 512 kB. We expect that the default values for these two limits (`read_ahead_kb` and `max_sectors_kb`) are chosen in order to ensure fairness across different disk requests and to keep latency low.

To ensure that `max_sectors_kb` does not limit the size of disk reads we set its value to 16 MB. We then adjust `read_ahead_kb` and examine the throughput obtained by nginx. Figure 3.10 shows the results of these experiments and demonstrates the importance of

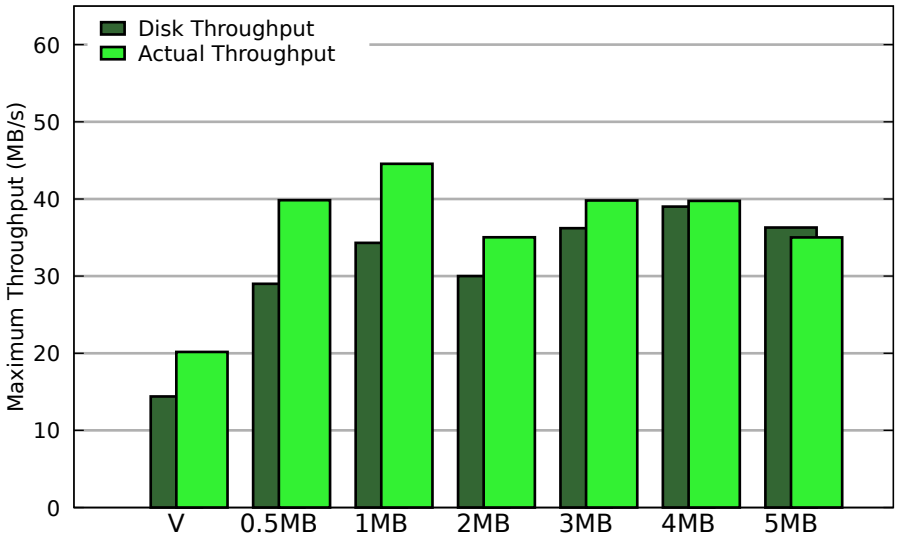

Figure 3.9: Throughput of nginx on Linux without Libception as disk readahead is modified using various `read_ahead_kb` sizes (WD drive).

setting both kernel parameters to proper values in order to obtain high throughput on this workload. In this case a `read_ahead_kb` size of 1 MB or 2 MB results in throughput of about 58 MB/s, which is slightly better than that obtained using Libception. For one configuration, disk throughput actually exceeds web server throughput. We discuss why this is the case at the end of this section.

Figure 3.11 shows results obtained using Libception in conjunction with modified Linux kernel parameters. For this experiment `max_sectors_kb` is again set to 16 MB to ensure that it does not limit physical disk read sizes. The Libception prefetch size and `read_ahead_kb` are adjusted together using the values shown along the x-axis of the graph. The column labelled "V" shows the "Vanilla" case where nginx runs without Libception and the default kernel parameter values are used. Although these results show a slight improvement in maximum failure-free throughput when compared with just using Libception, they are still lower than adjusting the kernel parameters alone and not using Libception.

We observe that the disk throughput exceeds the total server throughput for some configurations of the experiments conducted in this section. In these cases, prefetched data is either being evicted before it is requested by and sent to the client (due to memory pressure), or is simply never requested by the client at all (which can occur, for example, if a client's session ends). We refer to these as "evicted data" and "wasted data", respectively.

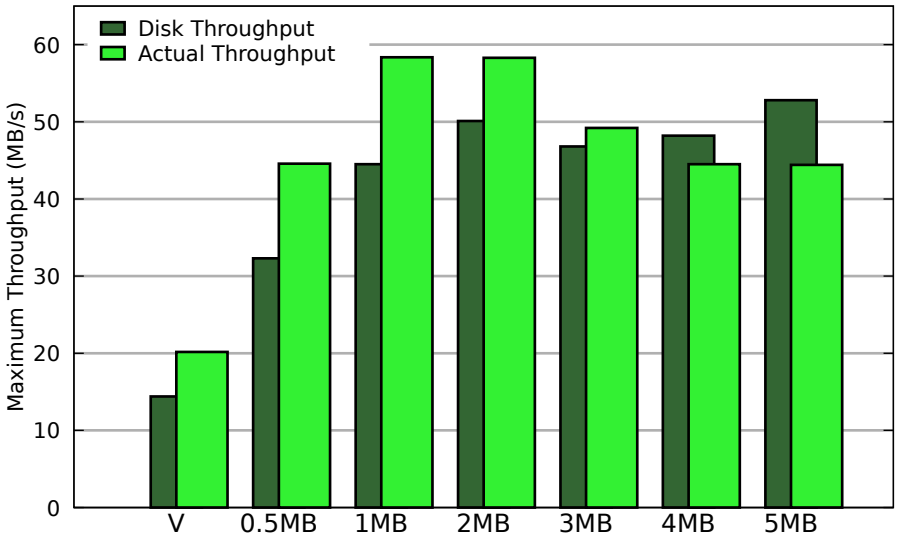

Figure 3.10: Throughput of nginx on Linux without Libception as disk readahead is modified using various `read_ahead_kb` sizes and with max sectors set using `max_sectors_kb` = 16 MB (WD drive).

Previous research on a similar workload has shown that, while overly aggressive prefetches do result in wasted data, they result in significantly more evicted data [107]. This same research showed that expanding the amount of memory in the system helps to increase total server throughput, as doing so decreases the memory pressure caused by aggressive prefetching. It seems that, in HTTP video streaming workloads, it may be more important to have memory acting as a buffer for large aggressive prefetching (in order to obtain high disk throughput) than as a file system cache.

3.4.6 Latency

When servicing HTTP streaming video on demand workloads, server throughput is strongly influenced by disk throughput. This is because video on demand services like YouTube and Netflix have large numbers of videos that are viewed infrequently (in other words, the popularity distribution of videos has a long tail). In order to improve disk throughput Libception prefetches relatively large amounts of data and serializes access to each disk. This naturally increases disk throughput while potentially increasing latency for some requests. The potential for increased latency should be relatively harmless when servicing

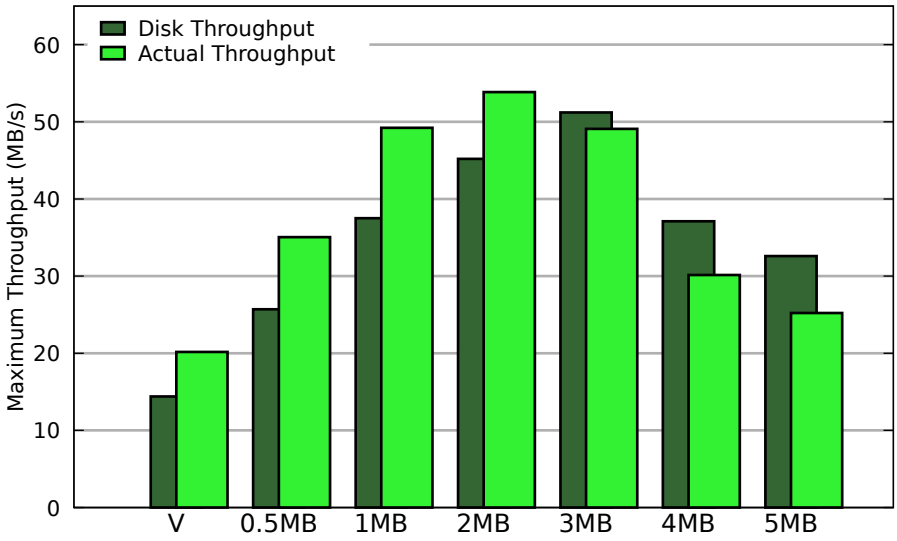

Figure 3.11: Throughput of nginx on Linux with Libception as disk readahead is modified using various `read_ahead_kb` sizes and with max sectors set using `max_sectors_kb` = 16 MB (WD drive).

video server workloads because clients are designed to be able to tolerate fairly significant WAN latency by using a playout buffer. The playout buffer is filled before playback begins and is used to seamlessly continue playing the video even during periods where the client experiences latency due to the network or HTTP server. For example, a client with a 10 second playout buffer can tolerate nearly 10 seconds of latency for some requests. As long as the data being requested arrives and can be decoded within 10 seconds of when it is requested, the user will not experience any service interruptions.

Before delving into further empirical analysis, it is important to note that all of our results place an implicit bound on latency. Client timeouts occur when a request has not received a corresponding response after 10 seconds. This latency includes the time to transmit the request, service the request at the server, transmit the response, and receive the response on the client. Therefore, if a client completes a session without errors, all of the latencies must have been acceptable.

In order to better understand the impact on latency of using Libception or the aggressive tunings of Linux kernel parameters that support high server throughput, we now examine the latency experienced by the web server processes. We use an existing system call tracing facility that exists in the userver to record the time required for every call to

`sendfile` in memory and print those times to disk after the server has finished servicing all requests. While client-side latency measurement would give an indication of end-user quality of experience, latency measurement at the server simplifies the collection process. Furthermore, as noted in Section 3.3, in our workload the network does not serve as a bottleneck. Therefore, server-side latency should be reflective of client-side latency, minus network transfer time.

Figure 3.12 shows the cumulative distribution function of `sendfile` call latency without Libception and with default kernel parameters (labelled "Vanilla"), with Libception, and with aggressive kernel parameter tunings. The "Vanilla" and "Libception" lines are created using data from one run with the userver using the configuration that obtains the highest error free rate (i.e., the two userver configurations depicted in Figure 3.3). The "Aggressive Kernel Params" line was created by setting `read_ahead_kb` to 2 MB, setting `max_sectors_kb` to 16 MB, and using the same request rate as used for the configuration of nginx shown in Figure 3.10.

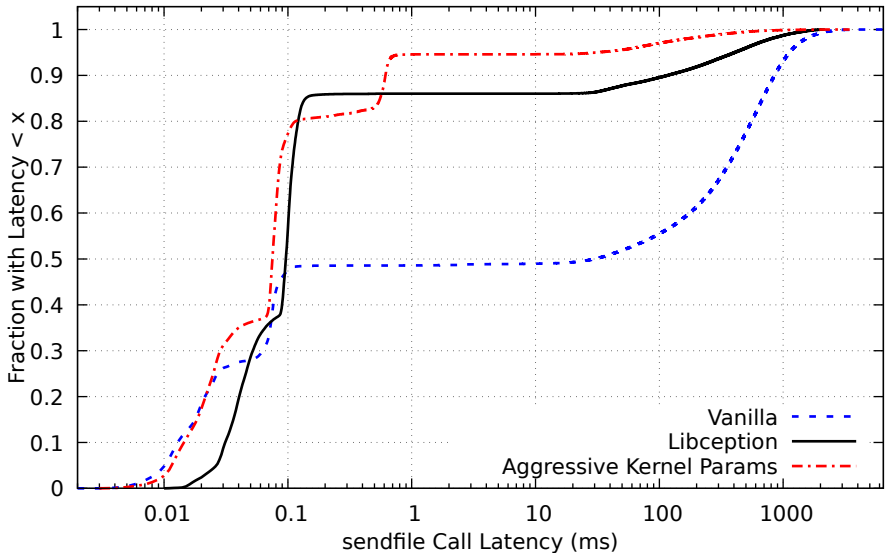

Figure 3.12: CDF of userver `sendfile` latency on Linux (WD drive).

We can see in Figure 3.12 that Libception has the smallest density of requests that are serviced in the lowest range of latency (i.e., below 0.1 ms). This is due to the extra overhead incurred from the mincore system call, which is used to determine if the data being requested is in memory or if it should be prefetched from disk. In cases where the data is already in memory, the lowest latencies are achieved by calling `sendfile` directly.

For requests where the desired data is not in memory at the time of the `sendfile` call, the "Vanilla" configuration results in a blocking call to `sendfile` because the data being requested needs to be read from disk. When a disk read occurs using either Libception or aggressively tuned kernel parameters, the application-initiated read is serialized and increased in size (to 2 MB in this case) to implement prefetching. Although this incurs some overhead for that individual request, subsequent reads are often served directly from the file system cache with comparatively low latency. The net result is a significant reduction in latency for a large number of `sendfile` calls. While about 49 % of `sendfile` calls take less than 1 ms under the "Vanilla" configuration, about 85 % of `sendfile` calls take less than 1 ms for the Libception configuration, and about 95 % of `sendfile` calls take less than 1 ms for the aggressive kernel parameters configuration.

The key observation from this experiment is that, although one might expect latency to increase because of large serialized prefetches, for this workload latency actually decreases for a large number of data requests.

3.4.7 Video Summary

Table 3.2 summarizes the main results from our experiments. Entries marked with an asterisk are not possible to obtain (for example, the ASAP userver requires an option to `sendfile` that is only available on FreeBSD so it cannot be run on Linux). Entries marked with "--" are not included primarily because they are unlikely to provide new insights. Note once again that the Linux and FreeBSD results use different hardware, and are therefore not directly comparable to each other. On FreeBSD, the use of Libception more than doubled the peak server throughput for Apache, nginx and the userver. We were not able to improve the poor performance achieved by these web servers without Libception by tuning FreeBSD kernel parameters. Comparing these results with results obtained by modifying the userver to incorporate aggressive prefetching and serialization shows that the overhead of implementing these techniques in a shim library, rather than directly, is negligible for our HTTP video streaming workload (on FreeBSD the throughput obtained with Libception for all servers matches that obtained with ASAP).

These FreeBSD results raise the question of whether they might reflect some deficiency in the block I/O scheduler in that system. This prompted us to take a careful look at performance on Linux, which supports three different block I/O schedulers. We found that the use of Libception approximately doubled the peak server throughput on Linux, for all three web servers, regardless of the choice of block I/O scheduler. However, using our insights from Libception as a guide (and in some cases examining the Linux kernel source

	Vanilla	Libception	Kernel Tuning	Libception and Kernel Tuning
FreeBSD				
nginx	39.75	83.71	*	*
Apache	30.10	83.88	*	*
userver	39.74	83.56	*	*
ASAP	83.99	--	*	*
Linux				
nginx	25.11	49.06	58.29	53.85
Apache	25.18	49.26	--	--
userver	25.16	49.18	--	--
ASAP	*	*	*	*

Table 3.2: Summary of web server throughput results: throughput in MB/s for web servers with various techniques on different systems. Results on FreeBSD are not directly comparable with those on Linux as they were obtained on different hardware.

code) we were able to discover kernel parameters that we could tune to obtain slightly better performance than that with Libception. In contrast to Libception, which is highly portable, we could not run our modified version of the userver on Linux since it makes use of a system call option that is only available on FreeBSD. Finally, a potential concern might be that these throughput improvements have significant cost with respect to latency, but as shown in Section 3.4.6 this does not appear to be the case.

3.4.8 Multiple Disks

We have conducted some experiments using FreeBSD and the HP disk drives where HTTP streaming video workloads are serviced using two disks. In this case a file set was created on one disk using the same methodology as was used for the single-disk experiments. To emulate the performance benefits of spreading content across disks, a copy of the file set was made on a second disk. Each video request was then serviced from either the first or second disk, based on static partitioning that randomly assigned each file to one of the two disks. This guarantees that the file cache is not polluted with two copies of the same file. Note that this means that the workload differs from the workload used in previous experiments in this chapter because the same number of clients are used but requests are now spread across both disks. As a result, one cannot directly compare the results

from previous experiments using only one disk with these experiments (for example, the throughput from the dual disk experiments may not necessarily be expected to be double the throughput of the single disk experiments). To drive an appropriate amount of load across two disks request rates were increased dramatically for these experiments.

When servicing this workload without Libception the average throughput on the two disks (observed by using the iostat utility) was 20 MB/s and 21 MB/s and the actual server throughput obtained was 65 MB/s. When using Libception with a prefetch size of 2 MB, the average observed disk throughput improved to 46 MB/s and 45 MB/s and the actual server throughput increased to 134 MB/s. This provides some evidence for our claim that Libception can be used with multiple disks and that serialization should be done on a per-disk drive basis.

3.4.9 Other Workloads

Other disk I/O-bound workloads with concurrently issued requests could also benefit from Libception. To demonstrate this, we employ a microbenchmark using the diff utility that compares the corresponding files in two versions of the Linux kernel (version 3.7.1 and version 3.8-rc2). As we do not care about the final result of the diff utility, but are rather just concerned with the disk performance metrics, we send output to `/dev/null`. We measure the completion time for a workload consisting of a single instance of this microbenchmark, as well as that for a workload with two instances running in parallel (using two copies of each kernel), with Libception (labelled "Libception") and without Libception (labelled "Vanilla"). Experiments are carried out on Linux with default kernel parameter settings using each of the three available block I/O schedulers. The results are shown in Figure 3.13. Note that these plots show total runtime on the y-axis, so a smaller bar is better.

As can be seen from the three pairs of bars on the left-hand side of Figure 3.13, when running a single instance of the diff utility, Libception does not significantly impact completion time. However, with two instances running in parallel generating multiple concurrent disk I/O requests, Libception reduces the workload completion time by over 50 % compared to results obtained without Libception. The choice of block I/O scheduler has little impact on these results.

Our understanding from examining the Linux kernel source code is that, when the first read from a file occurs, an initial readahead size is calculated based on the requested amount of data. In the case of the diff utility, the requested read size is 4 kB and Linux determines that up to 4 kB will be read synchronously to satisfy the read request, which

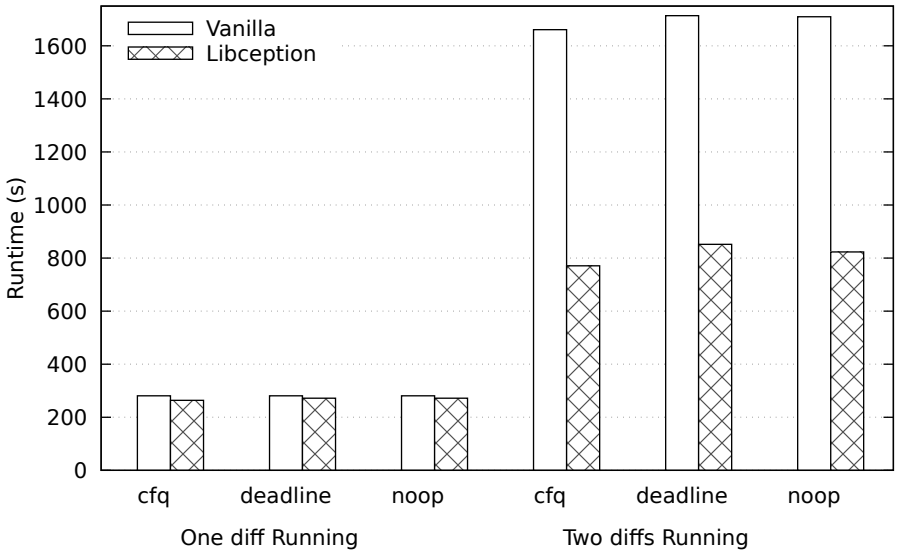

Figure 3.13: Runtimes for one and two copies of the diff utility, with and without Libception using different Linux block I/O schedulers (WD drive).

unblocks the process. Another 12 kB is also read asynchronously in anticipation of future sequential reads.

For files larger than 16 kB, additional reads will be needed. Figure 3.14 shows a CDF of the number of files of different sizes for version 3.7.1 of the Linux kernel (labelled "Linux 3.7.1 Files"). It also shows the proportion of total overall bytes that are represented by files of different sizes (labelled "Linux 3.7.1 Total Bytes"). Plots for version 3.8-rc2 of the Linux kernel show results that are mostly indistinguishable from the results for Linux kernel version 3.7.1 and are not included. The curve for total bytes shows that files larger than 16 kB account for about 50 % of all of the bytes in the Linux kernel. This is perhaps surprising, given that the curve displaying the number of files of different sizes shows that files larger than 16 kB in size constitute less than 10 % of all files. Note that even though the files are substantially smaller than in our HTTP streaming video workload, multiple reads to access all the data in a file are still common.

When two instances of our diff utility microbenchmark are run in parallel on Linux using the CFQ scheduler without Libception, disk I/O becomes much less efficient due to the interleaving of disk I/O requests from the two instances. This reduction in disk I/O efficiency is quantified in Table 3.3 by the average disk I/O service and wait times obtained from the iostat utility. Comparing the average disk I/O service and wait times in

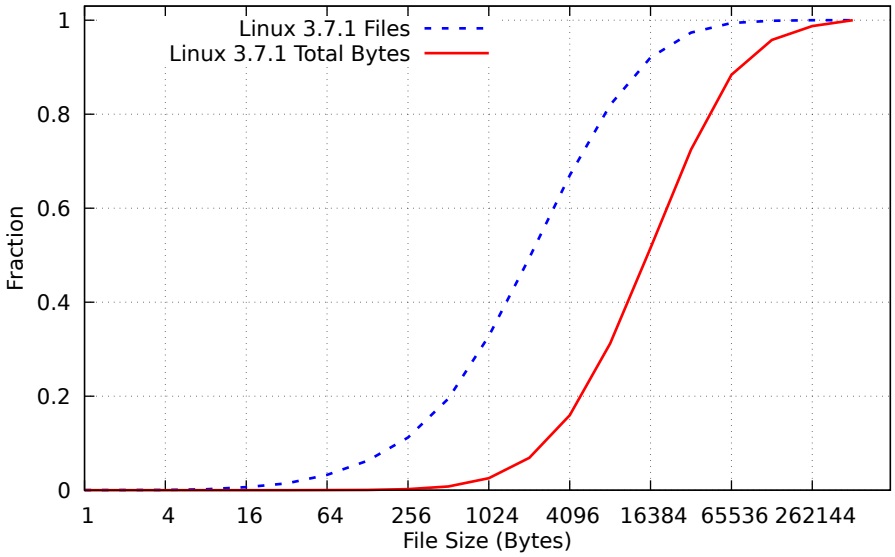

Figure 3.14: A CDF showing the fraction of files of different sizes in Linux kernel version 3.7.1, as well as the total proportion of bytes represented by those files.

Table 3.3 for two copies of the diff utility without Libception (labelled "Vanilla") and two copies of the diff utility using Libception, it can be seen that Libception greatly improves disk I/O efficiency for this workload. Libception also reduces the total I/O count, as can be seen from the combination of the average reads per second values in Table 3.3 and the respective completion times shown in Figure 3.13. The net result is a large reduction in total disk busy time, and therefore workload completion time. These results suggest that other workloads may also benefit from using Libception.

Workload	Utilization (%)	Avg Service Time (ms)	Avg Tput. (MB/s)	Avg Reads/s	Avg Wait (ms)
Vanilla One diff	96.6	3.2	3.9	387.2	19.6
Libception One diff	94.3	3.5	4.2	339.7	19.4
Vanilla Two diffs	99.7	8.2	1.3	130.2	45.0
Libception Two diffs	98.6	5.0	2.9	231.3	17.3

Table 3.3: Summary of disk performance statistics from iostat when running diff microbenchmarks.

3.5 Discussion

Perhaps surprisingly our results show that, without Libception, none of the web servers we investigated yield good performance for HTTP video streaming workloads, on either FreeBSD or Linux with default kernel parameter settings. We believe it is possible that operating systems and web servers may still be optimized for more traditional workloads, despite the rapid growth of HTTP video streaming. Also, developers might assume that, after many years of research on prefetching (or readahead) in operating systems, that efficiently using a magnetic disk is a "solved problem," but evidently this is not the case. Although with Linux it was eventually possible to achieve good performance after kernel parameter tuning, it is noteworthy that the initial effort required to determine which parameters existed, exactly how they worked, and which settings yielded the best performance was non-trivial. For services like Netflix and YouTube, due to the large amounts of video available, the wide variety of bit rates at which each video is encoded, and because of the large number of videos that are viewed infrequently, it is not economically viable to store it all on solid state drives (SSDs). As a result, obtaining good performance when servicing video from disks is still important in these settings. We have found Libception to be a relatively simple and portable platform for implementing and evaluating techniques for improving disk I/O efficiency. In particular, using Libception we were able to readily evaluate the benefits of aggressive prefetching and I/O serialization for HTTP video streaming workloads, using multiple web servers and operating systems.

As demonstrated in Section 3.4.4 and Section 3.4.5, selecting a prefetch size which is too small results in low disk throughput. Likewise, selecting a prefetch size which is too large leads to a drop in system throughput. As a result obtaining peak server throughput requires choosing the most appropriate prefetch size. In previous work we explored how the best prefetch size and the benefits obtained from prefetching are sensitive to workload and system properties [107]. We show how the best prefetch size can be affected by the amount of available system memory, the distribution of the popularity of videos requested, hard drive characteristics, and the bit rates of files being served. For example, we demonstrate that increasing the prefetch size can increase disk throughput if sufficient system memory is available. However, if there is insufficient system memory available then prefetched data may be evicted from the file cache before it can be used, reducing overall system throughput. Likewise, we demonstrate that the physical properties of the disk, including transfer times and seek times, significantly impact the benefits conferred by prefetching.

In order to avoid having to exhaustively and repeatedly determine the best prefetch size when workload or system characteristics change, this same previous work introduced an algorithm for dynamically and automatically adjusting the prefetch size with the goal

of obtaining peak server throughput [107]. We demonstrate that a gradient descent algorithm, which minimizes a score based on a combination of disk transfer times and file system cache miss ratio, is effective at selecting prefetching sizes that result in high server throughput. This adaptive algorithm results in throughput rivalling that obtained by exhaustive manual tuning across a variety of different workload and system characteristics. We believe that such a strategy could be added to Libception, allowing it to continue providing high throughput while also eliminating the need to manually select a prefetch size.

More recent work [106] has analyzed log files of HTTP requests to characterize the workload of production Netflix storage servers. This work shows, through simulation, that workload-specific adjustments can be used to increase server throughput. However, some of these improvements require knowledge of request streams and other information about the workload that are difficult to infer in Libception. An interesting question is how much information can be provided to Libception without requiring changes to web server code and whether or not such changes can compete with the performance that can be obtained by directly modifying the web server. This line of questioning would be interesting to explore in future work.

We additionally believe it would be useful to continue exploring Libception's utility for improving other workloads, for example a workload combining multiple applications with different I/O needs sharing the same disks. When all involved applications request large amounts of data, Libception can continue to provide aggressive prefetching and request serialization. However, applications that perform small, random reads will not benefit from these techniques. For these applications, large prefetching is wasteful as data cached during readahead may never be accessed or it may not be accessed before it is evicted from memory. Rather, prefetching should be minimized for these applications ensuring that only useful data is read. Similarly, serialization would not help to improve read performance for such applications since their reads are small enough that they can be handled with one disk request (and the operating system will not interleave requests from other processes with these requests). Therefore, in order to satisfy both types of applications, Libception would need to balance the trade-off between the performance of the throughput-dependent applications making large sequential reads and the latency needs of applications performing small random reads, since small reads may be delayed by the large prefetching reads.

Our work has focused on improving the throughput of streaming video content that is serviced from hard disk drives (HDDs). Despite the fact that SSDs are not cost-effective for storing entire content libraries, they do have utility for storing and quickly serving popular content. For example, SSDs are being used by Netflix to store hot content and increase overall system throughput. Examining techniques designed for improving the throughput

of systems using SSDs is a possible topic for future research.

3.6 Chapter Summary

HTTP streaming video delivery has become an important class of web server workloads. Such workloads have significantly different characteristics than the web server workloads that have been the focus of most prior work on web server performance. In particular, HTTP video streaming servers that store the long tail of content receive many parallel requests for large chunks of data which must be served from disk, meaning these servers are frequently disk-bound.

In this chapter, we have a explored method for making aggressive prefetching and serialization more widely available. We have designed, implemented, and evaluated Libception, an application-level shim library that incorporates these techniques to improve disk access efficiency. HTTP video streaming servers can achieve the benefits of these techniques simply by using `LD_PRELOAD` to dynamically link with Libception at runtime, preventing the need for any source code modification. Experiments with three web servers (Apache, nginx and the userver) and two operating systems (FreeBSD and Linux) show that with the aggressive prefetching and disk I/O serialization techniques currently implemented in Libception, peak server throughput can be doubled. Only after studying the Linux kernel source code and adjusting the appropriate tunable kernel parameters was it possible to achieve performance competitive with Libception. We believe that Libception could be applied to investigate other techniques for improving HTTP video streaming performance, and possibly for improving the performance of other disk-intensive applications.

Chapter 4

Nessie

4.1 Introduction

Distributed in-memory storage systems are an important class of applications used by content delivery services to provide quick access to stored data. By eliminating slow disk and SSD accesses, these storage systems, such as memcached [31, 52], Redis [87], and Tachyon [59], can reduce an application's request service time by more than an order of magnitude [78]. In the absence of disks and SSDs, however, networking often becomes a significant source of application latency, in particular network interface cards and operating system protocol stacks [96]. Traditional network interface cards and operating protocol stacks have been shown to hinder network performance [96], and as network operations approach tens or hundreds of microseconds (or even milliseconds) to complete, the need for better networking solutions has become apparent.

The drive for increasing performance has led recent in-memory storage systems [70, 26, 47, 127] to use remote direct memory access (RDMA). RDMA enables direct access to memory on a remote node, and provides additional performance benefits such as kernel bypass and zero-copy data transfers. These RDMA-based systems are able to provide lower latencies and higher throughput than systems using TCP or UDP.

Although existing RDMA-enabled key-value stores (RKVSes) perform well on workloads involving small data sizes and dedicated servers, we have identified three environments for which these designs are not well-suited. The first environment includes those where CPUs cannot be dedicated for exclusive use by the RKVS. This includes applications where it is advantageous for distributed computation and storage to be integrated

into the same node, or where physical CPUs are shared by multiple virtual machines. Existing RKVSes are designed to run in environments with dedicated hardware. They achieve low latency by using server threads to perform CPU-intensive polling for incoming requests [26, 47]. We refer to these systems, in which client requests are forwarded to a dedicated server thread for processing, as server-driven. This approach is susceptible to performance degradation in environments with multiplexed CPU resources, as an increase in CPU usage from a co-resident application or VM can cause context switching that delays the processing of requests, and significantly reduces the throughput of the RKVS. This delay is amplified when a server-thread is pre-empted, preventing progress for all clients communicating with that server thread.

The second environment that current designs have not focused on are those where the RKVS is required to store large data values. Existing RKVSes use designs that couple indexing and data storage, resulting in rigid data placement and unavoidable data transfers for put operations [18]. As data value sizes grow, so does the amount of network bandwidth required to service key-value operations. This presents issues for a variety of workloads, for example when using an RKVS as a memory manager for a distributed computation framework such as Apache Spark [135] which manages data units that can be hundreds of megabytes in size. When indexing and data storage are coupled, writing new data of this size results in significant overhead as the data is sent over the network to the node responsible for its key.

The third environment includes those where reducing energy consumption is important. As previously explained, server-driven RKVSes use polling threads alongside RDMA to decrease operation latency. However, with commodity network interfaces achieving bandwidths of 10 Gb/s, 40 Gb/s, and higher, a single polling thread per machine is often insufficient to saturate the link. Instead, these RKVSes must dedicate multiple polling threads to handling requests. While this is useful when operating at peak load, polling during off-peak periods is energy-inefficient. In some parallelized data centre applications, it is possible to consolidate resources by shutting down VMs during periods of inactivity. For an in-memory storage system, however, this means frequently reorganizing hundreds or possibly thousands of gigabytes of data.

In this chapter, we present the design of Nessie, a new RKVS that addresses some of the limitations of current RKVSes. Nessie is fully client-driven, meaning requests are satisfied entirely by clients using one-sided RDMA verbs that do not involve the server process' CPU. One-sided reads and writes allow Nessie to eliminate the polling which is essential for providing high throughput in server-driven RKVSes. This makes Nessie less susceptible to performance degradation caused by CPU interference in scenarios where CPUs are shared, such as cloud computing environments. Furthermore, Nessie only consumes

CPU resources when clients are actively making requests in the system, reducing Nessie's power consumption during non-peak loads when compared with server-driven systems that poll. Finally, Nessie decouples indexing and data storage by storing data values separately from indexing metadata. This allows indices to refer to data stored on different nodes in the system, enabling more flexible data placement (for instance local writes), and can benefit write-heavy workloads by exploiting locality and reducing the volume of remote operations.

4.2 Design Space

For the purposes of this chapter, we describe four existing RDMA-enabled storage systems (each implementing a key-value store) related to Nessie. These systems represent a selection from the state of the art research at the time the Nessie protocol was developed, with each system making different design decisions. The selected systems, which we describe fully in Section 2.2.2, are Pilaf [70], FaRM [26], HERD [47], and DrTM [127]. Table 4.1 shows a summary of these systems' design choices in terms of their network sidedness (as described in Section 2.1.6) and data coupling (as discussed in Section 2.1.7). The table additionally shows whether or not the systems require additional specialized hardware (beyond RNICs) to operate, and also provides an estimate of relative CPU consumption (based primarily upon network sidedness). Nessie is also included as a point of comparison.

	Network Sidedness	Data Coupling	Extra Hardware	Estimated CPU Use
Pilaf	Hybrid	Partially	No	Low–Medium
FaRM	Hybrid	Fully or Partially	No	Medium
HERD	Server-Driven	Fully	No	High
DrTM	Client-Driven (Ordered) or Server-Driven (Unordered)	Partially	Hardware Transactional Memory	Data-Dependent
Nessie	Client-Driven	Decoupled	No	Low

Table 4.1: An overview of systems within the RDMA key-value store design space at the time the Nessie protocol was developed.

Our evaluation in Section 4.6 focuses on comparisons of Nessie with a FaRM-like system and a HERD-like system. We do not include a comparison of Nessie against Pilaf as it is

a hybrid system like FaRM. We also do not include a comparison of Nessie against DrTM, as DrTM requires hardware transactional memory to implement its protocol.

4.3 Motivation

Existing RKVSes use server-driven operations for at least some aspect of their put operations, and none are decoupled (separating indexing metadata from data values). In Section 4.1 we highlighted three classes of workloads that are problematic for existing RKVS designs (workloads in shared CPU environments, workloads with large data values, and workloads that experience periods of non-peak load and could benefit from reduced energy consumption), and can be better addressed by a client-driven, decoupled system. In this section, we touch on the high-level design principles of a client-driven, decoupled system, and outline how it better addresses the properties of the given workloads.

4.3.1 Design Principles

Deployment Model: A client-driven, decoupled system is designed to operate in a distributed environment, with each machine acting as a request-issuing client and as a storage node. This provides for easy and efficient integration with distributed computation frameworks such as Apache Spark [135]. More generally, any workload containing collocated CPU-bound and memory-bound tasks is well-suited to this design (and therefore to Nessie). This includes web servers, which often consist of a CPU-bound process serving small amounts of pages from memory, as well as an in-memory cache or storage system, for example memcached [31, 52], that require little in terms of CPU utilization but consumes large amounts of memory.

Client-Driven Operations: Operations in a client-driven, decoupled system are exclusively one-sided, comprised of RDMA read verbs and RDMA compare-and-swap verbs. While such a system may use one-way RDMA write verbs as well, no active polling is involved on the server-side. This adds complexity to the protocol, but provides unique opportunities for performance improvements, which we elaborate on in Section 4.3.2. We provide a comprehensive breakdown of our client-driven protocol in Section 4.4.

Decoupled Data Structures: Although RDMA network operations are quite fast relative to traditional network operations, they are still up to 23 times slower than a local memory access [26]. Therefore, using local memory (whenever possible) remains an important part of maximizing performance in a distributed environment. However, fully and

partially coupled systems are limited in their ability to place data in order to take advantage of local memory. Data in these systems must be physically collocated with indexing metadata (which is typically statically partitioned), reducing opportunities for local memory optimizations. By using decoupled data structures, Nessie avoids a static partitioning scheme for its keys and values and is free to dynamically partition data at runtime, allowing it to better exploit local memory for performance purposes. Furthermore, Nessie can implement other optimizations such as a local cache to store recently accessed remote data table entries, and bits from the key hash, which we call filter bits, in index table entries to reduce unnecessary remote memory accesses. These techniques and optimizations are described in detail in Section 4.4.

4.3.2 Targeted Environments

Shared-CPU Environments: For modern computing workloads, it is often impossible to fully control the environment in which a KVS runs. This is especially common when using a cloud or shared cluster deployment. In the cloud, a physical machine is typically shared between multiple VMs, and its resources are oversubscribed. As we will demonstrate in Section 4.6, introducing sources of CPU contention leads to a reduction in throughput for systems that rely on polling server worker threads.

A common strategy for dealing with CPU contention is to isolate processes through tuning or pinning processes to particular CPUs, but this is difficult or impossible to do in a shared-CPU environment such as a VM in the cloud. Every time a polling server thread is pre-empted in order to share the CPU with another thread, all clients that have outstanding requests for that server to process experience a substantial delay. The magnitude of this delay can be quite severe given that RDMA verbs typically complete on the order of microseconds, whereas process scheduling timeslices are often in the millisecond range. As a result, if polling servers do not execute on dedicated CPUs, the throughput of the system can be dramatically reduced. Nessie's client-driven protocol eliminates server worker threads as potential sources of latency for operations. This leads to a greater degree of resilience against the effects of CPU contention.

Large Data Values: Previous RKVSes were designed primarily to provide high performance for operations involving small data values. FaRM [26] and HERD [47] in particular demonstrate significantly better performance when data values are small. Small data values allow FaRM, for instance, to place data values directly inside its indexing data structure, causing it to be fully coupled but also allowing it to efficiently return both data and indexing metadata with a single network operation. HERD, on the other hand, is explicitly

designed with only small values in mind, and does not consider workloads with larger data values.

Many applications require storage for, and low latency access to, larger objects, including web pages, photos, and distributed data. Apache Spark [135], for example, operates on objects divided into units called partitions. Unless instructed otherwise, when creating a dataset from a file, Spark uses one partition per block in the file. The Spark programming guide notes that, when used in conjunction with the Hadoop Distributed File System (a common deployment), this implies a default size of 128 MB per partition [5]. Another system, Redis [87], is an in-memory data store popular with web server operators for its ability to work with complex objects such as lists and sorted sets. For example, Pinterest, a content sharing service, uses Redis to store social data structures, including lists of users and pages followed by other users [10]. These data structures can become large, especially when they are used by popular services. Nessie's use of decoupled indexing and storage structures, together with other optimizations that reduce data transfer during get and put operations by enabling the use of local memory, helps to improve performance for workloads involving large data values when compared with systems that have partially or fully coupled data structures. This has been put to use in practice by the R3S system [132], which integrated Nessie into Spark [135] as a back-end data partition memory manager. In this instance, Nessie helped to provide high-performance storage and movement for thousands of 1 MB data partitions.

Energy Consumption: The polling nature of server-driven RKVSes implies that, for these systems, certain CPU cores of a node will be 100 % busy. This is appropriate for workloads that constantly saturate the system, but many workloads, including those found in data centres, exhibit bursty traffic patterns, with occasional busy peaks but also periods of little or even no traffic. Heller et al. [38] showed, for instance, that traffic exhibited diurnal behaviour in both a large e-commerce application and a production Google data centre. Atikoglu et al. [6] likewise showed diurnal and weekly traffic patterns for requests to Facebook's memcached deployment. During any period of reduced activity for a bursty or light workload, a server-driven RKVS would continue using relatively large amounts of CPU, wasting electricity. The client-driven design used by Nessie, which precludes the need for polling server threads, results in lower power consumption for these same types of workloads.

4.4 Design

4.4.1 System Components

Nessie is an RDMA-enabled key-value store that decouples indexing metadata, which is stored in data structures we refer to as index tables, and key-value pairs, which are stored in data structures we refer to as data tables. Each instance of Nessie contains an index table, a data table, and a small local cache of remote key-value pairs. This is depicted at a high level in Figure 4.1. Throughout this section we elaborate further on the details of each of these components.

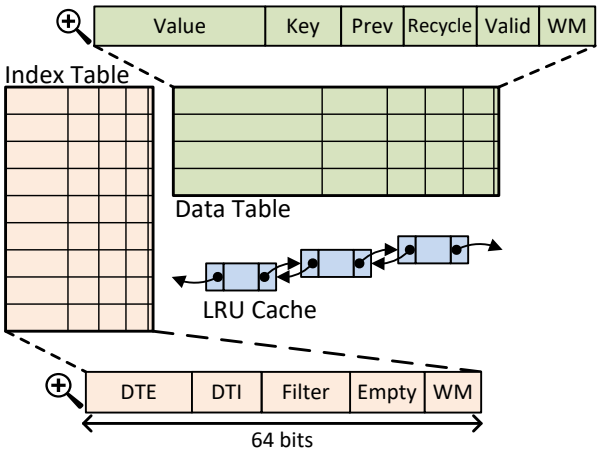

Figure 4.1: An overview of the components of Nessie. Each instance of Nessie contains an index table, data table, and local cache.

Index Table

Index tables are implemented as a distributed N-way cuckoo hash table [79]. In N-way cuckoo hashing, each key potentially hashes to one of N different entries within index tables across the deployment, and inserted entries that overlap with others displace them through an action known as migration. We describe cuckoo hashing in depth in Section 2.1.8. Cuckoo hashing was chosen because it needs to examine a maximum of N entries on each read operation, and is simple to implement without requiring coordination between nodes.

Other approaches would yield different performance properties. For instance, hopscotch hashing (which is used by FaRM [26]) may require fewer network operations under certain scenarios. Exploring other hashing alternatives would be an interesting possibility for future work.

Each index table entry (ITE) is a 64-bit integer, allowing ITEs to be operated on by atomic RDMA compare-and-swap verbs. This atomic access eliminates inconsistencies that could result from simultaneous writes to the same ITE. Because RDMA compare-and-swap verbs are only atomic relative to other RDMA verbs on the same RNIC, all ITE accesses (even for local ITEs) are managed using RDMA. This means that each RNIC may be responsible for one or more index tables, but no index table may have entries that are being managed by more than one RNIC. The small size of the index table entries allows Nessie's index tables to have a very large number of entries without using much memory. This allows Nessie to maintain a low load factor for its index tables (which is important for high performance with cuckoo hashing), without consuming large amounts of resources.

A possible concern with using RDMA atomic verbs is that they can negatively affect performance [48]. Nessie reduces the performance impact of using these operations by applying them judiciously. RDMA compare-and-swap verbs are only used during put and delete operations, and these operations will typically only require a single RDMA compare-and-swap verb. Additionally, because Nessie is concerned primarily with large values, the overhead of working with these large values tends to dominate workloads. Moreover, on some RDMA-enabled hardware (for example, the Mellanox Connect-IB, which we did not have access to), proper placement of data can significantly improve performance when working with atomic operations. Previous work [48] demonstrates that the Connect-IB uses buckets to slot atomic operations by address. Nessie's ITEs are stored in memory that is completely separate from data table entries (DTEs), meaning that on such hardware, memory could be allocated in such a way that RDMA compare-and-swaps on ITEs would not cause performance penalties for concurrent access to DTEs. Index tables could also be intelligently allocated to spread groups of ITEs across different RNIC address buckets, increasing parallelism. Thanks to Nessie's small ITEs which result in a low load factor for index tables of even a moderate size, contention caused by atomics would further be reduced. Even without special hardware support, Nessie has the ability to use multiple RNICs on the same node by creating multiple index tables and data tables and partitioning them between RNICs. This can further reduce contention for the RNIC by different requests, ensuring that RNIC resources do not bottleneck the system.

The 64 bits of an ITE are flexibly divided between several fields, as illustrated in Figure 4.1. The first field is an identifier that determines a particular DTE in a given data table. The next field is a data table identifier (DTI), which uniquely identifies a data table

in the deployment. Taken together, a DTI and DTE identify a spatially unique portion of memory in the cluster. This spatial uniqueness is combined with an expiration-based system in order to resolve concurrency and caching issues (see Section 4.4.3). A third field, called the empty bit, is a single bit wide and is used to represent whether or not a particular ITE is empty. Because RDMA verbs update their bytes in address order, the last bit of an ITE is used as a watermark which clients poll on to determine if an operation has completed [26]. This provides a lower-latency alternative to RDMA's event-raising model [26, 47]. Finally, any remaining bits which are not needed for the DTI and DTE fields may be used for a field that we call the filter bits, which are used to probabilistically avoid DTE lookups over the network. Filter bits are discussed in detail in Section 4.4.3.

Data Table

Nessie's data tables are simple arrays of key-value pairs and metadata. Key and value sizes are configurable at launch time, and fixed across a given deployment for its runtime. DTEs contain a valid bit, denoting whether or not a DTE belongs to an in-progress put operation (see Section 4.4.2). Like ITEs, DTEs also contain a watermark bit on which clients poll to determine whether or not an operation has completed. The fields of a DTE are arranged such that keys and metadata may be read without also needing to retrieve the value field. By default, the Nessie protocol retrieves entire DTEs when reading them, however the optional multiple reads optimization (described in Section 4.4.3) can change this behaviour, which is useful for large values. Furthermore, DTEs also contain an eight-byte timestamp and a recycle bit, which are used to implement a simple expiration-based system for reclaiming DTEs that are no longer in use (see Section 4.4.3), and an eight-byte previous version index, which is used to reduce the abort rate of get operations (see Section 4.4.3).

Once a DTE is assigned to an ITE, that ITE has ownership over the DTE. Apart from DTEs changing state from invalid to valid, DTEs are not modified after being written. Nessie's protocol ensures that it is impossible for operations to read DTEs which are incomplete in transition between valid and invalid (they will always be one state or the other). DTEs may therefore be accessed remotely using RDMA read verbs, and locally using simple memory reads (which can offer significant advantages for large data values).

Local Cache

The immutable nature of valid DTEs allows them to be locally cached, using their globally unique ITE as a cache identifier. This caching reduces network load, particularly for

workloads with skewed popularity distributions. Our current implementation of Nessie uses a small least recently used (LRU) cache on each node. When a key's associated entries are modified, its newly assigned ITE will be spatially unique from the key's previous ITE, and the key's old cache entry will be evicted over time as more entries are accessed. The same expiration-based strategy used to guarantee correctness when accessing DTEs remotely is also used with the cache. This prevents stale entries from being returned in the case where an application waits for a long duration in between cache updates and retrievals.

4.4.2 Protocol

In this section, we present a protocol overview of each Nessie operation. We later introduce additional mechanisms that are used to ensure correctness and improve performance. For additional details about these operations, we direct readers to Appendix A.1, which provides step-by-step breakdowns of each operation in addition to detailing how they interact with each other in terms of consistency.

Get

At a high level, a Nessie get operation for a particular key consists of a forward pass searching for a DTE containing the requested key (performed for cuckoo hashing) and a reverse pass (performed to verify consistency).

During the forward pass, Nessie iterates over a list of possible ITEs for the requested key using its cuckoo hashing functions. For simplicity, in this section and Appendix A.1, we describe Nessie as computing all the ITEs with its hashing functions at once, however in practice Nessie can delay the computation of an ITE to the relevant iteration (this way, unnecessary hashes are not computed if an operation can complete without needing them). Once an ITE is computed, Nessie retrieves it using an RDMA read verb. If the ITE is empty, or its filter bits do not match those of the requested key (which means the DTE it refers to cannot contain the key, as discussed further in Section 4.4.3), then the ITE is skipped. ITEs that are not skipped are used to retrieve candidate DTEs. First, if the DTE is in a local data table, its memory is inspected directly. Otherwise, the ITE is used to check for a DTE in the local cache. Finally, if the DTE is not local and is not cache-resident, it is retrieved from its remote location using an RDMA read verb. The key in the retrieved DTE is compared against the requested key, and if they do not match then Nessie continues to look for a match by iterating over any remaining ITEs.

Pending put operations set the valid flag in DTEs to false. To avoid returning inconsistent data, Nessie checks this flag when retrieving DTEs and returns a concurrent operation error if the DTE is invalid. A get operation that fails in this manner may retry after a back-off period. The use of a previous version index, an optimization described in Section 4.4.3, ensures that back-offs for get operations happen infrequently. If a valid DTE is found containing a matching key, it is cached (if it was retrieved from a remote node) and the value is returned. Figure 4.2 illustrates the basic steps involved in an example successful get operation. The labelled steps in the figure are explained in the figure's caption.

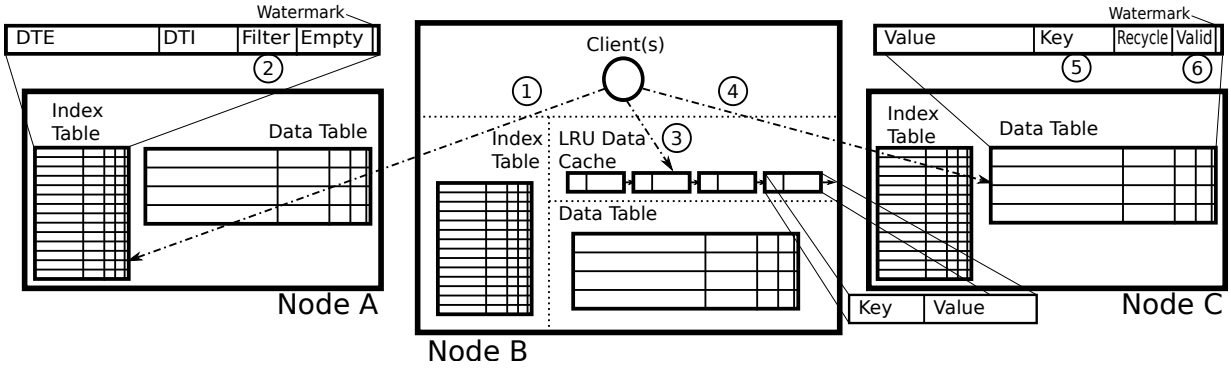

Figure 4.2: Example Nessie get operation initiated on node B. Each node contains a Nessie instance with an index table, data table (for simplicity, the previous version index is omitted), and LRU cache. (1) A node and ITE is determined using the key's hash and is read using RDMA from node A's index table. (2) The ITE's filter bits match the key's filter bits, meaning the DTE associated with this ITE could contain the requested key. (3) No entry is found for the ITE in node B's local cache. (4) The ITE's DTI and DTE are used for an RDMA read from node C's data table. (5) The DTE's key matches the requested key. (6) The DTE is valid, and the value is returned to the application.

If the requested key is not found in any of the candidate DTEs, another pass is made over the ITEs to account for an edge case. During the forward pass, a concurrent migrate operation could have moved an ITE for the requested key into an already-examined ITE slot (apart from the very last one examined) in such a way that it is not seen during the forward pass. Therefore, a get operation re-examines the ITEs (apart from the very last one examined during the forward pass) to determine if any have changed since they were last read. We describe and implement this second pass (and the second pass for other operations) as being done in reverse order, as it helps to distinguish the second pass from the first. From an algorithmic standpoint, however, it is likely that performing the second

pass in forward order would also work. During the reverse pass, if a change is detected, a concurrent operation error is returned, and the get operation is re-attempted after an application-determined back-off period. Otherwise, Nessie returns a value indicating that the requested key is not in the system. Given that a key-value store is typically used to access elements that are known to be present, performing the reverse pass is rare in practice. Finally, additional precautions are taken to prevent Nessie from returning recycled DTEs and stale cache values. These preventive measures are described in Section 4.4.3. An exhaustive description of Nessie's get protocol is provided in Appendix A.1.2. The other sections of Appendix A also discuss how other operations might interfere with get operations, and how these cases are handled by the protocol.

Put

Abstractly, Nessie's put operation protocol performs a forward pass over a requested key's possible ITEs to insert a new entry. As Nessie uses cuckoo hashing, it is possible that, after inserting an entry, another entry for the same key exists in a yet-unexamined ITE. Therefore, Nessie continues the forward pass and iterates over any remaining ITEs checking for conflicting entries to remove. For consistency, the put operation then performs a reverse pass and aborts if any conflicting put operations or migrate operations are detected. The combination of the forward and reverse iterations ensures that only one operation in a set of conflicting operations completes successfully.

During the forward pass, Nessie iterates over a requested key's possible ITEs, searching for a candidate ITE that is empty or refers to a DTE whose key matches the requested key (and can therefore be overwritten). This iteration is identical to that used by get operations except that values are not retrieved as they are not needed. If the forward pass finds that all ITEs for a requested key are non-empty and do not refer to matching DTEs, then a migrate operation is performed to free one of the ITEs.

With a candidate ITE chosen, Nessie must allocate a DTE for the put operation's key-value pair. Nessie's decoupled design allows this DTE to be placed on any node. In our implementation, the DTE is placed on the same node as the client making the request. This placement scheme is useful for a variety of workloads, as it avoids network round trips and provides locality for subsequent operations on the key from the same node. We discuss alternate placement schemes in Section 4.7, and we discuss the selection of empty local DTEs and their re-use in Section 4.4.2 and Section 4.4.3. The DTE contains a valid bit that is initially set to false, signifying that it belongs to an in-progress operation.

Once a new DTE has been written, the candidate ITE is atomically updated using

an RDMA compare-and-swap verb to contain the data table identifier and index of the new DTE. If the compare-and-swap on the candidate ITE fails, an error is returned and the caller may re-attempt the put operation. If the compare-and-swap succeeds and the candidate was initially non-empty, the candidate's original DTE is staged for recycling once the put operation completes. It is important to note that, at this point, the original DTE is still valid, and will be until after its expiration period (which is not set until after the put operation completes and changes its new DTE to valid). We take advantage of this for our previous version index optimization, described in Section 4.4.3. After updating the candidate ITE, the put operation continues its forward pass, iterating over any unchecked ITEs to ensure there are no duplicate entries for the key. Any duplicate entries that are found are marked as empty using an RDMA compare-and-swap verb, and their DTEs are staged for recycling once the put operation completes.

After the forward pass, a reverse pass is used to check for interference from concurrent operations. The put operation iterates backwards over the key's possible ITEs and verifies that none of them have changed since they were last retrieved. If a discrepancy is detected, a concurrent operation error is returned, and the caller must retry the put operation. Otherwise, the new DTE's valid bit is flipped to true. Any values that have been staged for recycling are marked with expiration times using RDMA write verbs, although for latency purposes Nessie may also delay this step until the put operation is complete and the system is not busy. DTE expiration is further discussed in Section 4.4.3. The put operation then returns successfully.

An example of a put operation can be seen in Figure 4.3 (details for each numbered step are described in the figure's caption). For simplicity, the figure uses two cuckoo hashing functions, ignores filter bits, and assumes that no interference occurs. In the example, the first checked ITE refers to a DTE with a key mismatch, and the second ITE is found to be empty. A complete breakdown of put operations in Nessie is provided in Appendix A.1.3, and includes a discussion of the conflict cases that put operations may cause for other operations and how these are resolved by the protocol.

Migrate

While trying to insert a new key during a put operation, it is possible that all of the new key's candidate ITEs could point to DTEs containing other keys. This is an expected behaviour in cuckoo hashing that simply results from different keys having overlapping candidate ITEs. To solve this issue, one of the candidate ITEs must be freed by migrating its contents to one of its alternate locations. A candidate ITE is selected as a source. In our implementation, we choose the first candidate ITE as the source. The migrate operation

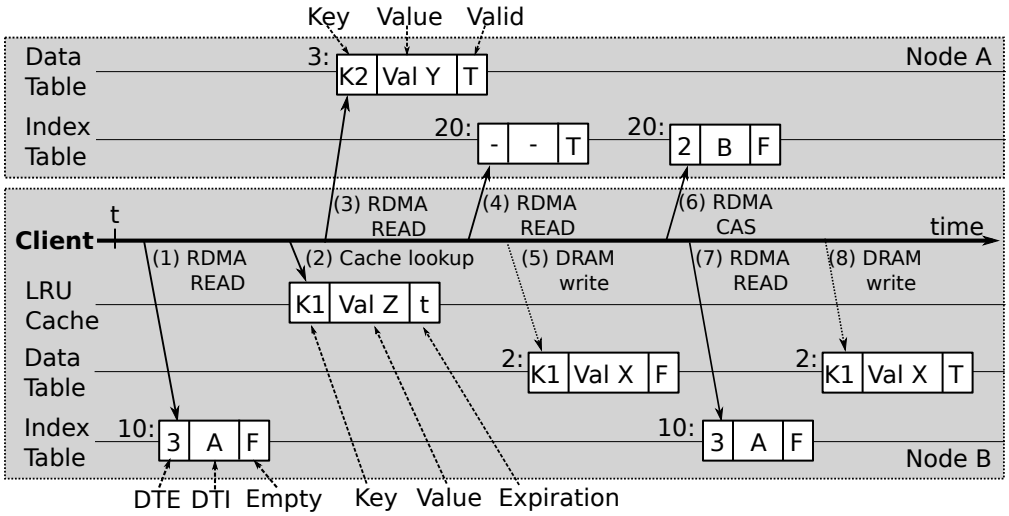

Figure 4.3: Example Nessie put operation on key "K1," value "Val X" by a client on node B: (1) Starting at time t, RDMA read the ITE determined by the first hash function on "K1." The ITE is non-empty, and points to DTE 3 on node A. (2) Because the ITE points to a remote DTE, it is looked up in the local cache. A match is found, but it expired at time t, and is therefore evicted from the cache. (3) RDMA read the DTE referenced in the index table lookup (DTE 3 on node A). The retrieved DTE contains key "K2" and is not a match. (4) RDMA read the ITE determined by the second hash function on "K1" (ITE 20 on node A). The ITE is empty, and may therefore be used for the put operation. (5) Create a new local DTE, DTE 2 on node B, with the valid flag set to false. (6) RDMA compare-and-swap ITE 20 on node A to refer to the new DTE. (7) For consistency, perform the reverse pass and RDMA read ITE 10 on node B to verify that it has not changed (it has not). (8) Set the valid flag in the new DTE to true.

is aborted if the source refers to an invalid DTE (to avoid interfering with an in-progress operation). Otherwise, the DTE referred to by the source is copied to a new local DTE, and the new DTE's valid bit is set to false. The list of candidate ITEs is computed for the key in the newly copied DTE, and one of these candidates is selected as a destination. Our current implementation of Nessie chooses the first candidate that is different from the source as the destination.

If the destination is not empty, Nessie frees it by performing a recursive migrate operation using the destination as a source. Once the destination is empty, an RDMA compare-and-swap verb overwrites it to point to the newly copied DTE, and the source ITE is marked as empty using another RDMA compare-and-swap verb. The newly copied DTE's valid bit is set to true, and the ITE originally referred to by the source is marked for recycling. The use of a new DTE for each operation, in combination with the DTE expiration times and operation timeouts discussed in Section 4.4.3, ensures that it is impossible for an operation, such as a get operation, to miss an existing value due to other concurrent operations. For example, if a get operation reads an ITE during its forwards pass, and a migrate operation subsequently moves an existing value into that ITE, the ITE now contains indexing data that is different from what was recorded by the get operation. Even if further operations change the ITE, the get operation will detect a mismatch on its backwards pass and return a concurrent operation error. Figure 4.4 illustrates an example migrate operation (step by step descriptions are provided in the figure caption).

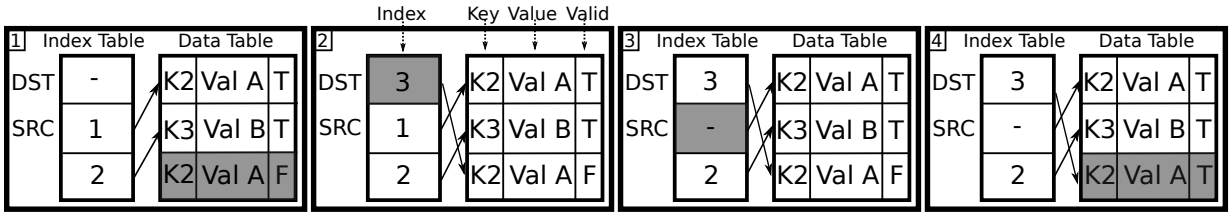

Figure 4.4: Example Nessie migrate operation from ITE "SRC" to ITE "DST." For simplicity, the index table's DTI and DTE are combined into a single "Index" field. The shading highlights the component being acted on in each step: (1) Create an invalid copy of SRC's DTE. (2) Change DST to refer to the new DTE. (3) Clear SRC. (4) Make the new DTE valid.

In theory, multiple recursive migrate operations may be needed to free a destination ITE. Therefore, at a configurable recursive depth the put operation that incurred the migrate operation is aborted and an error is returned, informing the caller that the system's index tables must be made larger or data must be deleted before re-attempting the put operation. In practice, because Nessie is able to maintain a low load factor for its index

tables without consuming much memory, it is easy and inexpensive (in terms of memory) to size index tables in such a way that even non-recursive migrate operations are rare. A more comprehensive examination of migrate operations is provided in Appendix A.1.4, which provides an in-depth breakdown of the concurrency conflicts that migration can cause, and how the Nessie protocol deals with these conflicts.

Delete

Delete operations in Nessie are identical to put operations, but instead of inserting a DTE with a specified value, a delete operation inserts a DTE with an empty value. Just as with put operations, any operations running concurrently to a delete operation on the same key will encounter an invalid DTE, allowing them to abort, back-off, and retry. Unlike a put operation, a delete operation never marks its inserted DTE as valid. Instead, the delete operation performs one last RDMA compare-and-swap verb on the ITE referring to the DTE, marking the ITE as empty. The delete operation then marks the DTE for recycling and returns. Appendix A.1.5 contains a more formal breakdown of Nessie's delete protocol.

Data Table Management

Each node maintains a list of data table entries which may be allocated during put operations or migrate operations. A DTE is in-use if an ITE contains a reference to it. When an ITE becomes empty or is changed to reference a different DTE, the original DTE it referred to can be re-used after an expiration period. Because ITEs are updated atomically using RDMA compare-and-swap verbs, and only the thread that clears an ITE is allowed to recycle a DTE, it is impossible for multiple operations to simultaneously mark a DTE for recycling. Therefore, direct memory writes and RDMA write verbs (depending on locality) are sufficient to set the recycle bit and expiration timestamp of a DTE.

When a node runs out of available DTEs, it scans its DTEs for entries that have been marked for recycling, and adds them back into its list of available entries. This scan is performed when available DTEs are exhausted, but to reduce the frequency of this occurring on the critical path, it may be performed lazily by a background thread at scheduled intervals or when the CPU is idle. Likewise, the act of marking DTEs for recycling can also be moved into a background thread, removing the need to do so from an operation's critical path. To avoid the case where a particularly slow get operation or put operation reads an ITE, waits for a long duration, and then erroneously reads a DTE which has been recycled, we use a simple time-based expiration system. This is explained in greater detail in Section 4.4.3.

4.4.3 Protocol Modifications

In addition to the basic components of the Nessie protocol described in Section 4.4.2, several other techniques are used by Nessie to achieve better performance for certain workloads, and to avoid consistency issues when recycling memory. In this section, we examine these additional mechanisms.

Correctness

Aborting migrate operations: Migrate operations may be interrupted by encountering an invalid source ITE, or by failing an RDMA compare-and-swap verb on the destination or the source ITE. If a migrate operation encounters an invalid source, or is unable to compare-and-swap the destination ITE, it may safely abort as it has made no changes to the system. If a migrate operation fails to compare-and-swap the source ITE then it must mark the destination copy it has created as empty. This is done using an RDMA compare-and-swap verb. If marking the destination as empty is successful, the DTE referred to by the destination is marked for recycling and the migrate operation returns a concurrent operation error. If marking the destination as empty fails, this means that another operation has already marked the destination as empty. The migrate operation may then safely return a concurrent operation error.

DTE timestamping and expiration: Nessie does not allow DTEs to be immediately recycled when they are replaced by put, migrate, or delete operations. This is to prevent a case where a slow operation reads an ITE and then delays long enough before reading the associated DTE that the DTE has been recycled and re-used by concurrent operations. To enforce this, DTEs contain a recycle flag, denoting whether or not they are in the process of being recycled, and an 8-byte expiration timestamp. When the recycle flag is set, the timestamp is updated to a time in the future after a configurable expiration period. A Nessie client attempting to re-use freed DTEs cannot re-use a DTE until it has fully expired. Furthermore, a DTE is only timestamped for recycling after the completion of an event that has marked its ITE as empty. Thus, any node that reads a valid ITE is guaranteed to receive a consistent value if they subsequently read the associated DTE within one expiration period. We therefore enforce the condition that every Nessie operation returns an error once a single expiration period of time has elapsed, and must be re-attempted. Although a timeout at the operation level is more strict than is necessary to maintain consistency, it is easy to implement and is unlikely to be exceeded in practice.

Configuring the DTE expiration period: The DTE expiration period defines how long a DTE lives in memory before it is allowed to be reclaimed and used again by the

system, but it also defines the upper bound of time that an operation is allowed to run for before it must be aborted. Therefore, it is important to strike a balance when choosing an expiration period: it must be long enough that it does not unnecessarily cause operations to abort, but it must also be short enough that the system does not run out of available DTEs during periods of heavy update activity. It is possible to determine a safe upper bound on the expiration period using the rate of put operations, and the amount of overflow capacity available in the data table. For example, if the system is expected to handle 600,000 put operations per second, which is more than the highest rate of put operations we reached in any experiment, and has 600,000 data table entries of overflow capacity, which in our experimental setup would be a data table operating at 80 % of capacity, then the expiration period should be no longer than one second. In less put operation-heavy workloads, a longer expiration period would suffice. For example, with 100,000 put operations per second, an expiration period of six seconds could be used. As is shown in Section 4.6.1, the latency of get operations and put operations are on the order of microseconds. Even when operations have tail latencies in the millisecond range, this would still be an order of magnitude smaller than an expiration period on the order of seconds. Thus, a one second expiration period would ensure DTE availability without causing unnecessary aborts.

Avoiding the ABA problem: If a slow-running Nessie client were to read an ITE and then stall while other nodes continued to run, it could be possible for the other nodes to write over the ITE many times in sequence. If the original DTE referred to by the ITE were to be recycled and then used again for the same ITE, the slow-running Nessie client might re-read the ITE and (wrongly) assume that no intermediate changes occurred. This type of behaviour, where a value is overwritten with new data and is then updated once again to contain its original data, can cause concurrency issues. This is known as the ABA problem [128]. Nessie avoids ABA-style issues by relying on its DTE expiration times in conjunction with operation timeouts. When an ITE is updated, it will point to a DTE that is spatially unique from its previous update. In a sequence of ITE updates, in order for an ITE to point away from a particular DTE, and then back to the same DTE again, a full expiration period would need to elapse (as otherwise the DTE would not have been recycled). Thus, any operation that reads the ITE prior to its update and finds the DTE in question, and then reads the ITE again after its sequence of updates and finds the same DTE, must have taken longer than the operation timeout period. Therefore, the operation would be aborted as potentially stale.

Cache consistency: Nessie stores cached DTEs using their associated ITEs as keys, and looks up cached entries using ITEs retrieved during the course of an operation. As discussed previously, a DTE retrieved using a valid ITE is also guaranteed to be valid for a full expiration period from the time the ITE was read. To maintain cache consistency,

Nessie therefore stores items together with their expiration times in the local cache. If a cache lookup yields an expired ITE-DTE pair, Nessie treats it as a cache miss, and evicts the entry from the cache. When an ITE results in an unexpired hit in the cache, the DTE in the entry is again guaranteed to be consistent for a full expiration period from the time at which the ITE was read. This is because reading the DTE from the cache, so long as it has not expired, is no different than fetching it from a remote node. This allows Nessie to extend the expiration time of entries on a cache hit, and it does so by immediately overwriting the cache entry's expiration time with the newly computed one. Because Nessie is a big memory system, the size of the dataset implies that the local cache primarily benefits workloads with highly skewed access patterns. Therefore, Nessie continues to receive most of its caching performance benefits even with a relatively short expiration time.

Synchronizing DTE timestamps: Using timestamps for expiration times requires the clocks of nodes in the cluster to be loosely synchronized. The configurable expiration time used by a Nessie cluster will typically be on the order of seconds, whereas operations in the system occur on the order of microseconds. This means that the simple use of the Network Time Protocol (NTP), which is able to achieve millisecond-level (or better) accuracy [69], provides sufficient synchronization accuracy for Nessie's purposes. Any variance between clocks on different nodes after NTP is applied is therefore minor, and can be accounted for by adding in a small safety factor when checking whether or not a DTE is considered expired.

Availability during node failure: Nessie does not provide replication, and as a result a node failure leads to the loss of indices and data. Additionally, portions of the keyspace for which the lost node shared responsibility or was operating on may be unavailable until the node is restored. There are potential challenges involved in adding fault tolerance to Nessie. This could involve the implementation of a client-driven version of a consensus protocol, the use of external failure detection, or integration with a hardware solution such as non-volatile memory. Exploring how to provide a fault-tolerant, client-driven system would be an interesting avenue for future research.

Consistency model: Each thread that performs operations in Nessie (referred to as a client worker) completes operations sequentially, one at a time. Therefore, when a client worker completes an operation, any changes it has made are visible to the next operation performed by the same client worker. If any operation (on any client worker) sees changes from a completed operation, so will any other operation that begins after this point in time. It is possible for operation ordering to vary between client workers in other circumstances, however. For example, consider the case of a put operation that initiates slightly before a

get operation on a different client worker, with both operations working on the same key. In this case, the get operation could see an in-progress put operation and return a value using the previous version index (described later), completing before the put operation despite having been initiated after it. Based on these properties, Nessie satisfies the criteria for supporting a sequential consistency model [131].

Performance

Filter bits: Nessie uses filter bits in ITEs to reduce the frequency of cuckoo hash misses, which occur when a DTE is retrieved that does not match the requested key of an operation. Filter bits are similar to hash table optimizations used by MemC3 [28], MICA [60], and MemcachedGPU [40]. This optimization is particularly important for workloads with densely populated index tables, and workloads with large DTEs. When a new DTE is being inserted, the trailing bits of its key hash are stored in the filter bits of the ITE that references the DTE. Upon reading an ITE, the filter bits may be compared against the filter bits of the requested key. If these bits do not match, then it is guaranteed that the ITE's referenced DTE does not contain an entry for the requested key.

Previous version index: If one or more clients attempt get operations on a key for which a put operation is simultaneously occurring, the clients performing the get operations are unable to complete as they see a DTE with its valid bit set to false. This can lead to large latencies for get operations in put operation-heavy workloads with a small subset of popular keys. However, recall that when put operations (or equivalently, delete and migrate operations) atomically overwrite an ITE, the associated DTE still exists in the system, and is usable for the duration of the operation in question plus an expiration period afterwards (as the DTE is only marked for recycling once the operation has completed). If a client had access to the previous DTE, then it could return it, instead of having to wait. To provide clients with this access, before an operation modifies a non-empty ITE, it copies the ITE being replaced into the previous version index of the new DTE. When a client retrieves a DTE and determines its valid bit is not set (and is therefore associated with an in-progress operation), it uses the previous version index to retrieve the DTE that was replaced by the in-progress operation. If the DTE retrieved with the previous version index is valid, and matches the requested key, then the get operation returns it successfully. In Section 4.6.2, we show that the previous version index significantly improves throughput for a put operation-heavy workload with key accesses that follow a Zipfian distribution.

Back-offs and avoiding retries: The use of previous versions reduces occurrences of Nessie's put operations from interfering with its get operations. However, it is still possible

for put operations on colliding keys to interfere, and in these circumstances, one of the put operations must be aborted and retried after a back-off period. To avoid a scenario where newly arriving put operations continually interrupt other in-progress put operations (thereby preventing forward progress), Nessie aborts put operations that encounter an invalid DTE while choosing a candidate ITE, as this implies another put operation is already occurring. We use a standard exponential back-off for retrying put operations.

Multiple reads: As data values become very large, there is a significant overhead associated with cuckoo hash misses during get operations, which occur when DTEs are read that do not contain the key being searched for (put, delete, and migrate operations do not retrieve the value field of a DTE, so their overhead is the same regardless of how large values become). Despite the use of filter bits to help mitigate cuckoo hash misses, when values are large enough, the cost of retrieving unnecessary data during get operations, however infrequent, has a major deleterious effect on performance. In these cases, we allow get operations to split remote DTE retrieval across two RDMA read verbs, one to fetch all non-value fields, and another to fetch the DTE including the value (if the key matches and the DTE is valid). Despite incurring an additional round trip, multiple reads decrease unnecessary bandwidth, and result in a performance increase for large values (as we show in 4.6.2).

4.5 Implementation

We have implemented Nessie with about 4,400 lines of C++, which sits on top of networking, configuration and benchmarking infrastructure consisting of another 8,500 lines of C++. Nessie allows for quick and easy deployment across a cluster of RDMA-enabled nodes. It supports configurable numbers of cuckoo hashing functions, networking parameters, data table sizes, index table sizes, and more. Our implementation supports a key size and data size configurable at launch time (which remain fixed thereafter). Future implementations could be made to partition data tables into entries of different sizes in order to accommodate more efficient storage of variably sized data. This is similar to the approach used by memcached [31, 52], which divides slabs of memory into pieces using configurable chunk sizes. Our current implementation of Nessie supports partitioning across a fixed number of machines. For a discussion of supporting dynamic node join and leave, please refer to Section 4.7. Nessie uses the RDMA reliable connection (RC) transport type to provide network connectivity between nodes in the system, which is required in order to support Nessie's use of RDMA read verbs and RDMA compare-and-swap verbs. The RC

transport type also guarantees transmission, and therefore prevents the need to implement an extra, external layer of reliability for Nessie's networking protocol.

In order to compare our fully decoupled, client-driven Nessie implementation against similar approaches that use other communication mechanisms, we have also implemented a server-driven version of Nessie, called NessieSD, and a hybrid version of Nessie, called NessieHY. Similar to HERD [47], NessieSD uses RDMA write verbs to insert requests into a server's request memory region. These regions are polled by server worker threads, which service both get and put operation requests, and respond to them using RDMA write verbs. In the case where a request is made by and serviced by the same node, the request and response are optimized to be written directly, without using RDMA. Although HERD is described and evaluated within the context of a single server node, we expand the design of NessieSD to run with multiple nodes. Like Nessie, NessieSD's index tables are implemented as cuckoo hash tables. Unlike Nessie, however, an ITE in NessieSD only ever refers to DTEs on the same node (making it partially coupled instead of decoupled). This design decision also emulates HERD, limiting the number of RDMA verbs necessary for NessieSD to complete a get operation or put operation, but potentially limiting opportunities for locality-based optimizations.

Inspired by the RKVS evaluated in FaRM [26], NessieHY employs a hybrid communication system that uses client-driven get operations and server-driven put operations. For its protocol, NessieHY combines techniques from both Nessie and NessieSD. As in NessieSD, NessieHY uses RDMA write verbs to push put operation requests to remote servers. These servers use polling server worker threads to service and respond to the put operation requests using RDMA write verbs. NessieHY's get operation protocol is largely similar to that of Nessie, described in Section 4.4.2. For comparison with Nessie and NessieSD, index tables for NessieHY are implemented as cuckoo hash tables. Similar to the index tables of NessieSD, NessieHY's index tables are partially coupled. This allows NessieHY to minimize the number of round trip operations required to service a put operation. Additionally, NessieHY is optimized to use direct memory accesses instead of RDMA verbs whenever possible when dealing with same-node data, for example when reading DTEs. This helps to reduce unnecessary consumption of RNIC resources.

Despite NessieSD being inspired by HERD, we have eschewed some of the low-level RDMA optimizations utilized by HERD [47], including using RDMA unreliable connection (UC) transport types and data inlining (which allows small messages to avoid a copy over the PCIe bus). While we do understand and value the performance benefits of UC transport types and inlining demonstrated in the HERD paper, we avoid using UC transport types to focus on environments that require guaranteed reliable delivery. Additionally, our goal is to build a storage system that is not designed specifically for very small value sizes and

as a result we have not been concerned with obtaining the benefits of inlining. Likewise, although NessieHY draws inspiration from the hybrid design of FaRM, it does not include other features that FaRM's hash tables do, for example the use of chained associative hopscotch hashing, the ability to perform multi-item transactions, and the use of data replication. Rather, we have provided systems that closely resemble each other apart from communication mechanisms, allowing for easy comparison between the three. Furthermore, we have not implemented several of Nessie's optimizations in NessieHY, including filter bits, caching, previous versions, and multiple reads. This is primarily because these feature are not present in other hybrid systems. Furthermore implementing these features in a NessieHY requires significant engineering effort which is outside the scope of our work.

The NessieSD and NessieHY implementations are about 3,600 lines of C++ and 4,200 lines of C++ respectively, and employ the same underlying infrastructure, configuration and deployment options as Nessie.

4.6 Evaluation

To evaluate Nessie, we use a cluster of 15 nodes, where each node is a Supermicro SYS-6017R-TDF server containing one Mellanox 10 Gb/s ConnectX-3 RNIC, 64 GB of RAM, and two Intel E5-2620v2 CPUs, each containing six cores with a base frequency of 2.1 GHz and a turbo frequency of 2.6 GHz. To simplify experiments and to ensure repeatability we have disabled hyperthreading. Each node is connected to a Mellanox SX1012 switch. All nodes run an Ubuntu 14.04.1 server distribution with Linux kernel version 3.13.0.

All experiments use an index table load factor of 0.4. This load factor is small enough that cuckoo hash collisions occur infrequently. We believe it is reasonable to use a low load factor because each index table entry is 64 bits, allowing the index table to be sparse without using much memory. For example, with 1.2 billion keys and a load factor of 0.4, only 1.5 GB are required per node to store the index table. Index table entries in all Nessie experiments are configured to use seven filter bits. Due to the low load factor used in our experiments, these filter bits do not play a major role in the results that we present. Similar to the results presented by Pilaf [70], we empirically determined that using three cuckoo hash functions strikes a good balance between memory efficiency and performance. We therefore use this number for our experiments.

Experiments run for a total of 120 seconds, and use a static key size of 128 bytes. We vary the size of values from 256 bytes to 128 kB (with one fixed size per experiment). The number of keys used per experiment is chosen such that, when all key-value pairs

are present in the system, each node should contain 20 GB worth of data on average. As a result, experiments with smaller data values have a larger number of keys than experiments with larger data values. Since each update requires a new DTE, for workloads with enough put operations, all DTEs will eventually become full and further additions will require recycling. To prevent excessive recycling we use data table sizes of 30 GB per node. Additionally we use a cache size that is able to accommodate data for 0.5 % of the total number of keys on each node. The cache provides negligible benefits for experiments with uniformly distributed access patterns. Therefore, the cache size was determined experimentally by its impact on Zipf-distributed workloads. The nature of the Zipf distribution means that the cache provides diminishing benefits as it continues to grow. Before an experiment begins, the system spends 30 seconds warming the cache to ensure that the experiment's interaction with the cache is more realistic.

Unless otherwise stated in an experiment, we express our throughput results in terms of application-level goodput, which is defined as the number of successful operations per second completed over the course of an experiment, multiplied by the size of data being transferred for each operation. This gives a more accurate determination of relative performance, as it does not artificially reward strategies that incidentally move large quantities of overhead data, demonstrating only how much useful data is transferred over the course of an experiment. Furthermore, for simplicity, data sizes are expressed in terms of powers of two, such that 1 kB equals 1,024 bytes, 1 MB equals 1,024 kB, and 1 GB equals 1,024 MB. The results recorded in our experiments are typically the averages across multiple identical runs and include confidence intervals, however our results were found to be relatively stable, and thus outside of our experiments with shared-CPU environments (where interference and instability is expected), the confidence intervals are either small (such as in Figure 4.14) or not visible (such as in Figure 4.6).

Our Nessie experiments drive load by performing operations in parallel using 12 client worker threads per node (one for each core on our nodes). We found that exceeding one worker per core caused contention which reduced throughput. Similarly, NessieSD and NessieHY were tuned to have one worker thread per core, however these are divided between server workers handling requests and client workers making requests. For these systems, tuning the number of client and server workers presents a trade-off between throughput and latency. In our experiments, we adjust these values for maximum system throughput. Through manual tuning, we determined that NessieSD's peak performance is obtained by having three server workers and nine client workers per machine when value sizes are less than 1 kB. If value sizes are greater than or equal to 1 kB, we found that two server workers and ten client workers per machine obtained the peak performance for NessieSD. With NessieSD, server worker threads service all operations, and therefore tuning their number

is a simple matter of trying to accommodate the load being generated by the system. With NessieHY, however, client workers service get operations, and server workers service put operations. Tuning the number of server workers versus client workers depends on additional factors such as the ratio of an experiment's get operations to put operations. Through manual tuning, we discovered that NessieHY's best throughput for 50 % get operation workloads was achieved with three server workers and nine client workers for data sizes less than or equal to 32 kB, and two server workers and ten client workers for larger data sizes. For 90 % and 99 % get operation workloads, NessieHY performed best with one server worker and 11 client workers for most data sizes, with results at two server workers and ten client workers providing comparable throughput as individual operations are serviced with lower latency but less load is driven by the system overall.

While it is possible to create server-driven mechanisms that reduce CPU usage by alternating between polling and blocking modes during periods of high and low activity, blocking communication mechanisms have been avoided by systems such as FaRM and HERD due to the dramatic increase in operation latency that they cause. We do not compare against such an approach, as the results would be difficult to interpret meaningfully due to the unacceptably high service latency when serving requests using blocking communication mechanisms.

4.6.1 Shared-CPU Environments

Our first set of experiments emulate a shared-CPU environment as might be found in a cloud data centre. In such environments, external processes compete for the CPU, causing contention. In these experiments, each server worker thread is pinned to its own core, and all client worker threads are pinned to a separate set of cores that does not include those being used by the server worker threads. Background threads are designed to cause CPU contention, and are not pinned to any CPUs. The background threads mimic the behaviour expected in a cloud computing environment where external processes cannot be controlled. Figure 4.5 shows goodput results for Nessie (labelled "CD"), NessieSD (labelled "SD"), and NessieHY (labelled "HY") on a 15 node cluster with an increasing number of background CPU-consuming processes. In this experiment, key accesses are randomly distributed in a uniform fashion across the entire keyspace and the size of a data value is 16 kB. Repeating this experiment with smaller data sizes yielded results demonstrating similar trends. The ratio of get operations to put operations in the experiment is 90 to 10. The results show that, with no background processes (in other words, no CPU contention), Nessie and NessieSD both perform equivalently well, while the goodput for NessieHY is slightly lower. However, as CPU contention is introduced in the form of external processes,

NessieSD's goodput declines dramatically, NessieHY's goodput declines somewhat but not nearly as dramatically, while Nessie remains relatively unaffected.

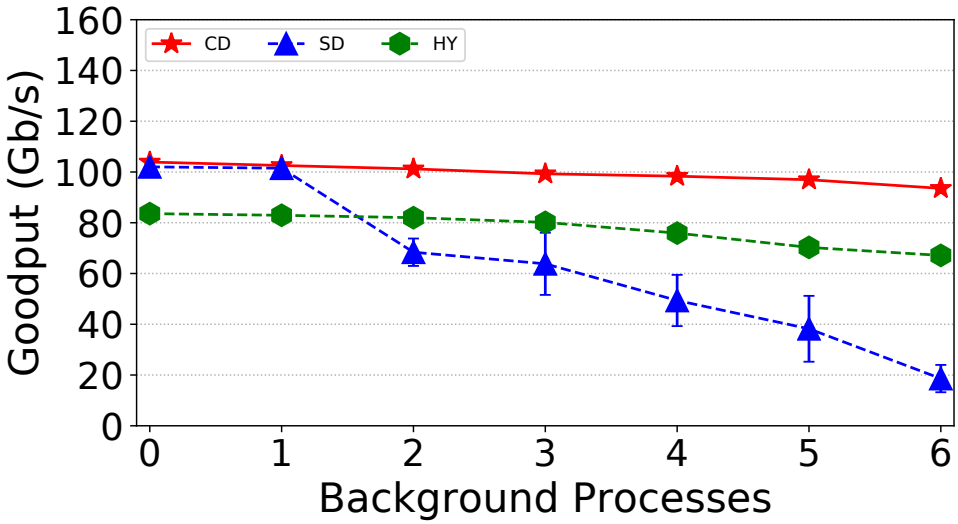

Figure 4.5: Goodput for client-driven Nessie (CD), server-driven NessieSD (SD), and hybrid NessieHY (HY) as other processes are introduced, using 16 kB values and 90 % get operations.

NessieSD's performance decline occurs because its server workers are unable to maintain the same level of high performance when faced with CPU contention. In NessieSD, clients must wait for a server worker to respond when a server worker thread is unable to access the CPU due to contention. When this occurs, all clients sending requests are unable to make any progress. In contrast, Nessie clients operate independently so if a single client worker thread is unable to access the CPU it only impacts the operation being serviced by that worker. NessieHY's performance similarly declines as CPU workers are introduced, however this rate of decline is lower than that of NessieSD as NessieHY only uses its server worker threads for put operations. Therefore, CPU contention has less impact on NessieHY than on NessieSD, and more impact on NessieHY than on Nessie. It is worth noting that the impact of competing background threads would decrease for NessieHY with a lower percentage of put operations, and increase for a higher percentage of put operations.

4.6.2 Large Data Values

Unlike existing RKVSes, Nessie is designed to efficiently support large data values. We now evaluate Nessie's performance using a range of data sizes and get operation to put operation ratios under both uniform and Zipf distributions. Unless otherwise stated, all experiments use 15 nodes.

Figure 4.6, Figure 4.7, and Figure 4.8 show the goodput of Nessie (labelled "CD"), NessieSD (labelled "SD"), and NessieHY (labelled "HY") using random, uniformly distributed keys with 50 %, 90 %, and 99 % get operations, respectively. We first note that, while a workload with a small amount of highly popular keys (such as the Zipf-distributed workload we examine later in this section) would see performance benefits from the local cache for get operation-heavy access (especially as values grow larger), this is not the case for these experiments due to their uniformly distributed key access patterns. We next note that, regardless of the ratio of get to put operations, NessieSD performs better than Nessie for small data sizes. In all cases, the opposite becomes true as data sizes increase, and the difference in performance closes quicker for put operation-heavy workloads. For example, with a 2 kB data size and 50 % get operations, Nessie performs 30 % worse than NessieSD. However, for the 50 % get operation workload with 8 kB data values and beyond, the performance of Nessie begins to outpace that of NessieSD. For data values of 128 kB and 50 % get operations, Nessie's goodput is about 70 % higher than NessieSD's goodput. As we will see later, this is attributable to Nessie's ability to take advantage of local data placement during put operations. For both the 90 % and 99 % get operation workloads, both Nessie and NessieSD perform equally well with 16 kB values. In both cases, Nessie makes small relative gains versus NessieSD as the data size continues to grow, with the gap in the 90 % get operation results being slightly more apparent, likely due to the presence of addition put operations (for which Nessie performs very well). The goodput of NessieHY is consistently lower than both Nessie and NessieSD, primarily due to its lack of filter bits, which we discuss later in this section.

To better explain the differences in performance for each system, we need additional information about how the systems operate. Figure 4.9 shows a CDF of the average number of RDMA verbs required by get operations and Figure 4.10 shows a CDF of the average number of RDMA verbs required by put operations for Nessie (labelled "CD"), NessieSD (labelled "SD"), and NessieHY (labelled "HY"). Similarly, Figure 4.11 and Figure 4.12 show CDFs for the average number of bytes sent over the network by both systems during individual get and put operations, respectively. These values include overhead bytes in addition to goodput bytes. All four figures are presented using data from the 50 % get operations, 50 % put operations workload using 16 kB DTEs and uniform key access across

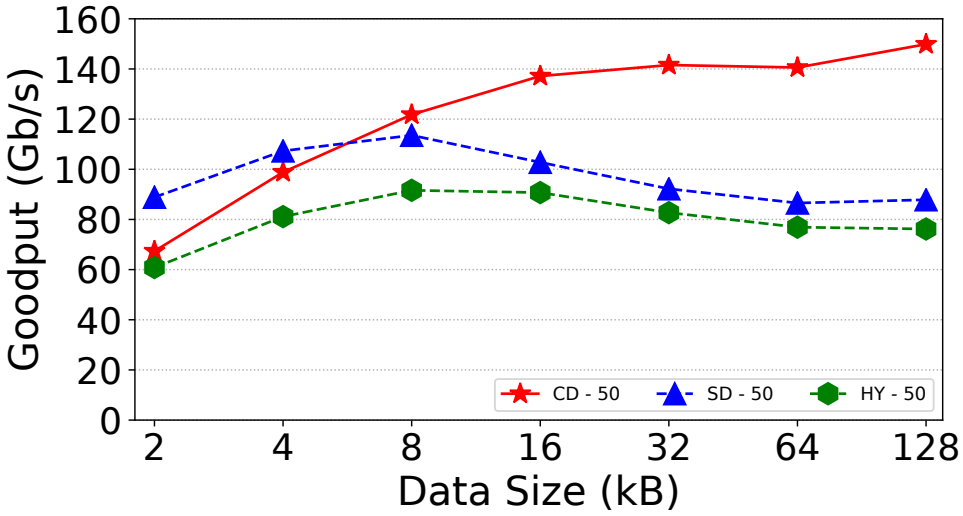

Figure 4.6: Goodput for Nessie (CD), NessieSD (SD), and NessieHY (HY) as data sizes increase. 15 nodes perform 50 % get operations and 50 % put operations with uniform random key access.

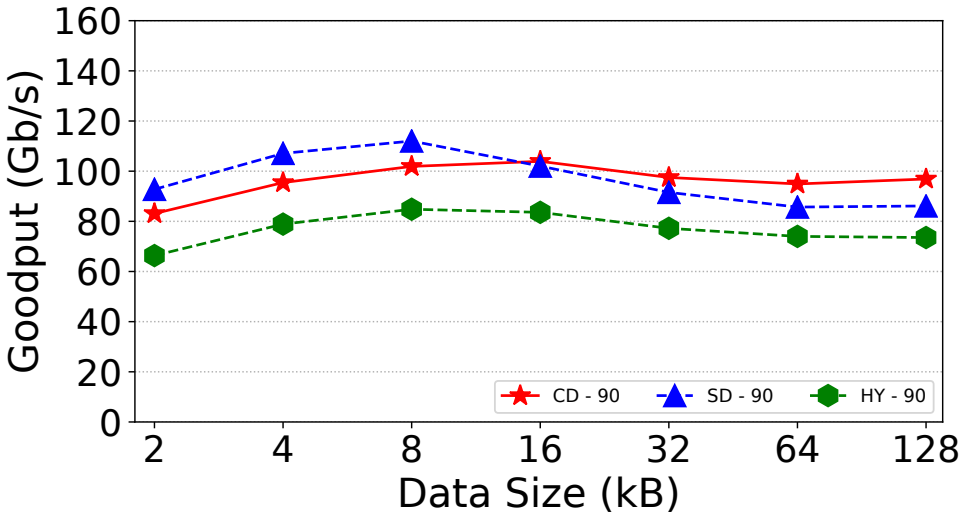

Figure 4.7: Goodput for Nessie (CD), NessieSD (SD), and NessieHY (HY) as data sizes increase. 15 nodes perform 90 % get operations and 10 % put operations with uniform random key access.

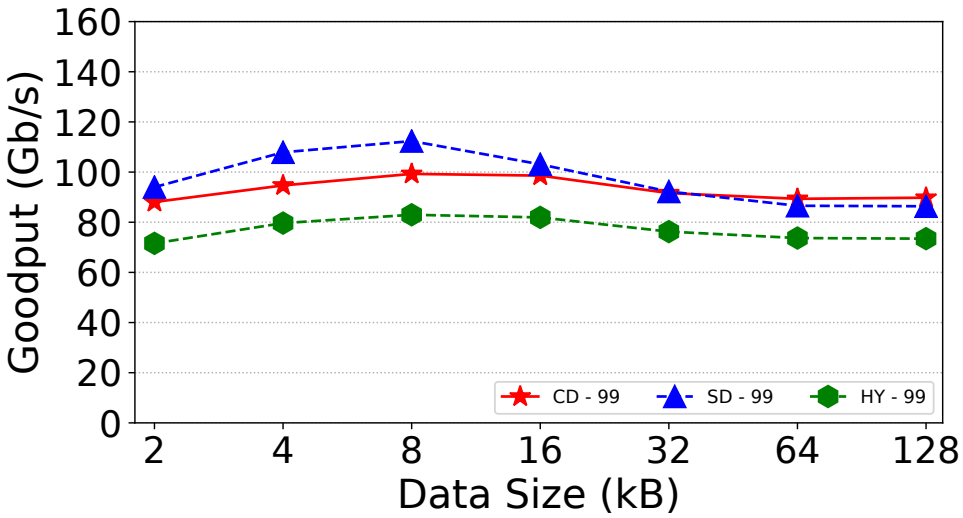

Figure 4.8: Goodput for Nessie (CD), NessieSD (SD), and NessieHY (HY) as data sizes increase. 15 nodes perform 99 % get operations and 1 % put operations with uniform random key access.

15 nodes. 16 kB DTEs are used because at this point in Figure 4.6 Nessie begins to increase in performance relative to NessieSD and NessieHY.

Figure 4.9 and Figure 4.10 show that the server-driven aspects of NessieSD and NessieHY allow them to complete a small number of operations without using RDMA. This is because about 6 % of ITE and DTE lookups are hosted on the node making the request. This benefit would shrink as the size of a cluster grows. Non-local NessieSD operations and non-local NessieHY put operations use two RDMA write verbs. Comparatively, Nessie get and put operations must always read at least one ITE using RDMA. A Nessie get operation uses no other RDMA verbs if data is local or can be serviced from the cache, but otherwise data is serviced remotely using RDMA. When the load factor of the system's hash tables increases, a small number of Nessie and NessieHY get operations require more than two RDMA verbs as they iterate over cuckoo hash indices. In these cases, Nessie is often able to avoid DTE lookups using filter bits. NessieHY, which does not use filter bits, incurs additional RDMA verbs on each cuckoo hash iteration. Nessie's put operations require a minimum of 5 RDMA read verbs to ITEs and an RDMA compare-and-swap verb on an ITE. Despite using more RDMA verbs, as values grow larger Nessie performs increasingly better than NessieSD and NessieHY. This can be attributed to Nessie's ability to exploit data locality and prevent network usage through local writes. We believe that introducing

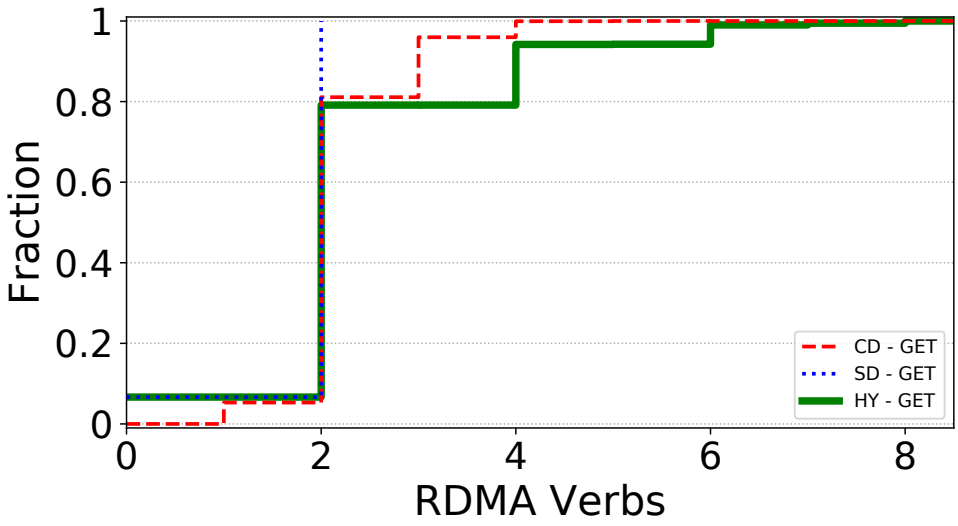

Figure 4.9: CDF of RDMA verbs required to complete a get operation for Nessie (CD), NessieSD (SD), and NessieHY (HY). 15 nodes perform 50 % get operations and 50 % put operations with uniform random key access on 16 kB data values.

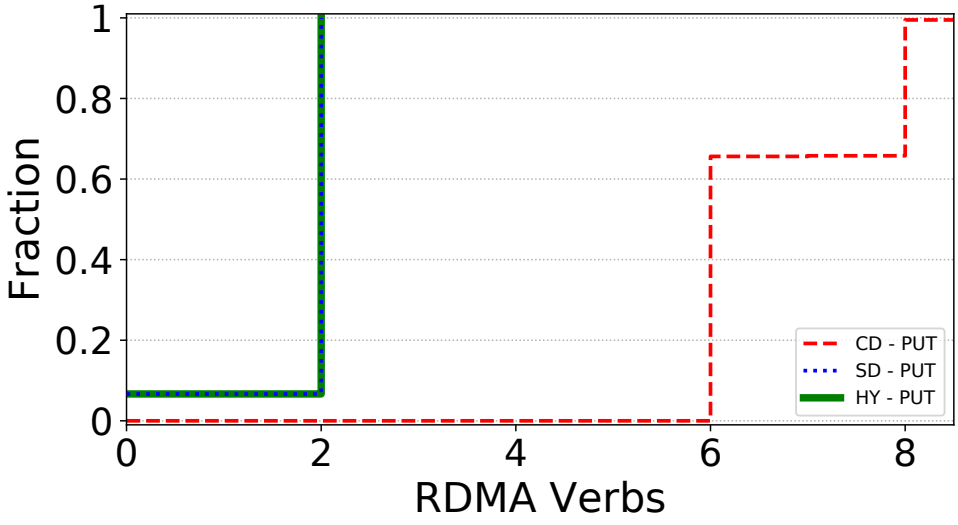

Figure 4.10: CDF of RDMA verbs required to complete a put operation for Nessie (CD), NessieSD (SD), and NessieHY (HY). 15 nodes perform 50 % get operations and 50 % put operations with uniform random key access on 16 kB data values.

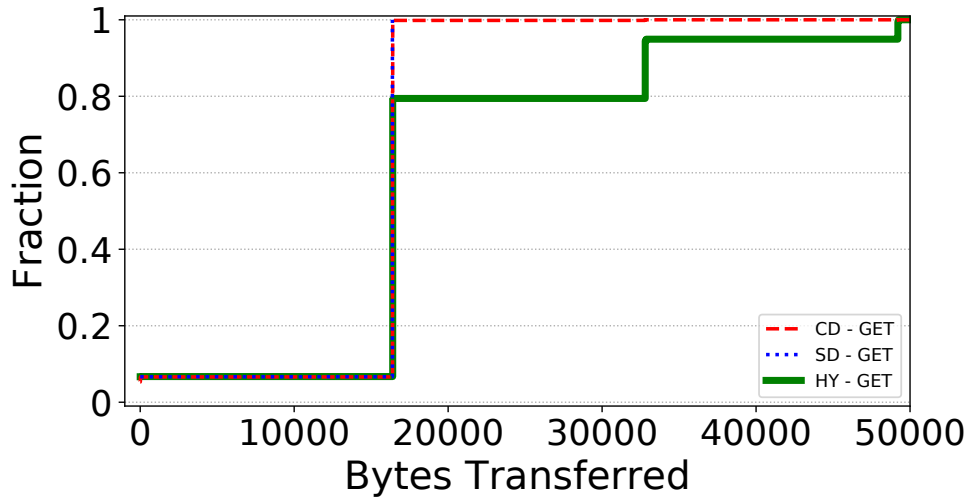

Figure 4.11: CDF of transferred bytes (overhead and goodput) required to complete a get operation for Nessie (CD), NessieSD (SD), and NessieHY (HY). 15 nodes perform 50 % get operations and 50 % put operations with uniform random key access on 16 kB data values.

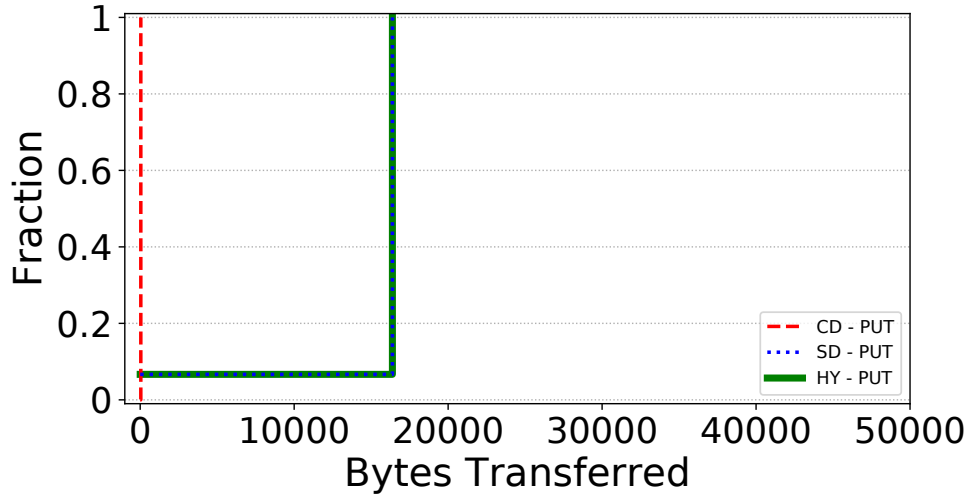

Figure 4.12: CDF of transferred bytes (overhead and goodput) required to complete a put operation for Nessie (CD), NessieSD (SD), and NessieHY (HY). 15 nodes perform 50 % get operations and 50 % put operations with uniform random key access on 16 kB data values. SD and HY are overlapping.

filter bits into NessieHY would bring its throughput in line with some Nessie results at small data sizes and NessieSD results at large data sizes.

Examining the number of bytes transferred during get and put operations in Figure 4.11 and Figure 4.12 reveals that, although Nessie and NessieSD transfer similar amounts of bytes on average during get operations, Nessie transfers far fewer bytes than NessieSD and NessieHY during put operations due to its local DTE placement scheme. For put operations, NessieHY on average transfers the same number of bytes as NessieSD. However, for get operations a significant fraction of the NessieHY 16 kB get operations require two or three cuckoo hash iterations before successfully retrieving the data. Unlike Nessie, which uses filter bits to prevent data transfer in this specific case, NessieHY requires a data transfer in these circumstances (as can be seen by the fraction of operations requiring 32 kB or 48 kB transfers). As a result, NessieHY obtains lower throughput in Figure 4.6, Figure 4.7, and Figure 4.8.

Table 4.2 contains data showing the get operation, put operation, and overall average latencies incurred for different data sizes. It additionally shows the total number of bytes transferred in each case, including both goodput and overhead bytes (for example, from cuckoo hashing). Minimum values for each grouping are shown in bold. We see in the table that, as data values grow in size, so too do the average latencies for get and put operations. We infer that this is caused by the long transfer times and network contention associated with operating on large data. This table also shows, however, that Nessie ("CD") sends substantially fewer total bytes over the network compared to NessieSD ("SD") and NessieHY ("HY") for all data sizes. By eliminating unnecessary data transfers with filter bits, and by preventing get operations from needing to compete with put operations for network resources through the use of local writes, Nessie continues to operate at lower levels of latency than NessieHY and NessieSD at large data sizes. Interestingly, average put operation latencies are minimized for data sizes larger than 16 kB when using NessieHY. We believe this can be attributed to the fact that NessieHY's server workers, which only need to process put operations, are more lightly loaded than those of NessieSD, which must process both get and put operations.

Data access patterns are also important to consider when values become large. Many applications exhibit skew in their access patterns for keys. In order to highlight some of the features of Nessie that have been explicitly designed to optimize for such cases, we next examine the performance of Nessie and NessieSD under these circumstances. NessieSD, due to its data coupling, is not able to benefit from these optimizations. We do not compare against NessieHY as it lacks the optimizations necessary for it to achieve acceptable performance for this workload. We consider a workload that uses a Zipf distribution with an α value of 0.99, following the guidelines of the YCSB benchmark [21]. The results of these

Data Size (kB)	2	4	8	16	32	64	128
CD Get Latency (µs)	**19**	**27**	**46**	**86**	**172**	**391**	**811**
SD Get Latency (µs)	25	43	82	184	412	875	1732
HY Get Latency (µs)	24	47	109	231	524	1193	2455
CD Put Latency (µs)	61	83	133	233	447	858	1533
SD Put Latency (µs)	**25**	**41**	77	171	381	816	1602
HY Put Latency (µs)	42	42	**69**	**129**	**271**	**709**	**1388**
CD All Latency (µs)	40	55	89	**159**	**310**	**624**	**1172**
SD All Latency (µs)	**25**	**42**	**80**	178	397	846	1667
HY All Latency (µs)	33	45	89	180	398	951	1922
CD Transferred (GB)	**514**	**729**	**877**	**975**	**999**	**989**	**1053**
SD Transferred (GB)	1245	1503	1590	1439	1291	1212	1230
HY Transferred (GB)	957	1287	1461	1445	1314	1219	1208

Table 4.2: Latency averages and total bytes transferred (overhead and goodput) per operation for Nessie (CD), NessieSD (SD), and NessieHY (HY) for uniform random key access with 15 nodes, 50 % get operations and 50 % put operations.

experiments can be seen in Figure 4.13, which shows the percentage difference between the goodput of Nessie and the goodput of NessieSD (the "relative goodput" of Nessie). As is the case for workloads that use a uniform random distribution of keys, Nessie provides its peak performance for large value workloads with 50 % get operations and 50 % put operations (an improvement of over 50 % compared to NessieSD). However, unlike the uniform random workload, Nessie also performs well for large values with a large percentage of get operations, outperforming NessieSD by over 40 % for workloads with 99 % get operations.

In addition to showing results for large data values, Figure 4.13 also shows the relative performance between Nessie and NessieSD for small data values of 256 bytes. Because Nessie's operations require more round trips to complete, smaller data values are negatively impacted by Nessie's design. However, get operation-heavy workloads with data sizes greater than 2 kB, or balanced workloads with data sizes greater than 16 kB, are able to benefit from Nessie's design, as it is specifically targeted at those workloads.

Nessie's performance results with a Zipf workload are obtained through a collection of optimizations. Figure 4.14 shows the breakdown of each optimization's contribution for 50 % get operations and 99 % get operations with 16 kB data values. Nessie's most basic form (labelled "Basic") performs much worse than Nessie with all of its optimizations enabled. This is attributable to the Nessie protocol's use of optimistic concurrency control. When many clients attempt to manipulate the same data at the same time (as is the case

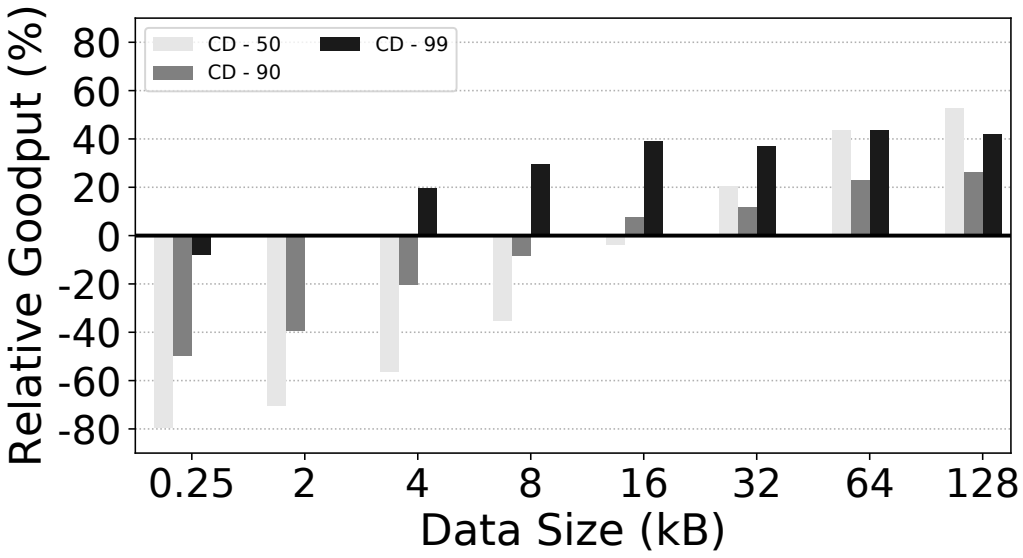

Figure 4.13: Relative goodput of Nessie compared to the goodput of NessieSD. Both systems use 15 nodes to access keys according to a Zipf distribution with 50 % get operations, 90 % get operations, and 99 % get operations.

for the Zipf workload), the high number of resulting conflicts cause operations to abort. This basic design is not well-suited for high levels of contention.

The first optimization we introduce for Nessie is filter bits (labelled "+ Filter"). Filter bits allow get and put operations to skip non-matching DTE lookups during cuckoo hashing by examining hash bits stored in ITEs. Normally filter bits have a moderate impact on throughput by reducing unnecessary bytes sent over the network, especially as data sizes grow or index table load factor increases. The uniform workload results seen in Figure 4.9 and Figure 4.12, for example, show that for that particular pattern of key access, Nessie with filter bits eliminates a value-sized RDMA read verb in 20 % of get operations and another value-sized RDMA read verb in 5 % of get operations, as well as two round trips in 35 % of put operations. NessieHY, which does not use filter bits but otherwise uses almost the same protocol, does not receive these benefits. The results in Figure 4.14, by comparison, are for a workload with a Zipf distribution. For the particular pattern of key access in that experiment, it appears that the most popular keys (which are accessed by Nessie exponentially more often than less popular keys) do not require multiple lookups, as filter bits have little performance impact.

The second optimization we introduce is caching (labelled "+ Caching"), which allows a

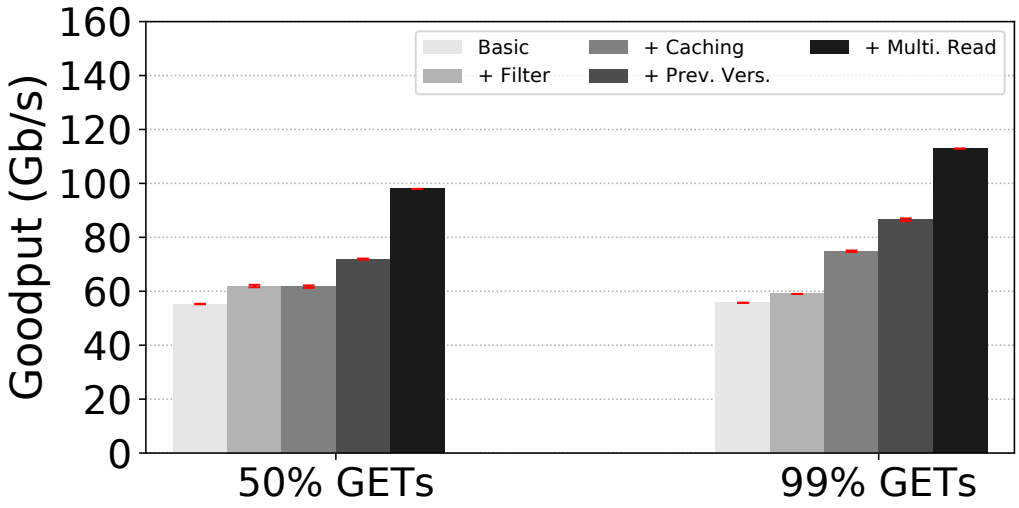

Figure 4.14: Impact of optimizations on Nessie goodput for Zipf workloads across 15 nodes with 50 % get operations and 99 % get operations on 16 kB data values.

client to fetch an otherwise remote DTE from the local cache if a client retrieves an ITE that has not changed since the last time it was retrieved (and the cached DTE has not exceeded its expiration time). As one might expect, caching is not useful for put operation-heavy workloads when keys are accessed according to a Zipf distribution, as the most popular keys change values frequently. Therefore, even when a DTE is cached, a future lookup on the same key will typically find a different ITE than the one that was used to cache the DTE, resulting in a cache miss. In contrast, get operation-heavy workloads make good use of caching by avoiding large data accesses, thereby freeing up network bandwidth for other operations.

While caching does increase Nessie's throughput, it does not solve the fundamental problem of high contention on the most popular keys. This problem occurs when a put operation is in progress for a key and other clients are trying to acquire the value for the same key. Once the put operation changes the ITE to reference its new, but still invalid, data table entry, all of the get operations are forced to back-off and retry until the put operation finishes. This is particularly problematic because each time a get operation reads the invalid DTE, it is consuming a large portion of the network resources and thereby potentially blocking the put operation from completing. To alleviate this contention, DTEs contain a copy of the old ITE so that the get operation can read the previous value referenced by the ITE (labelled "+ Prev. Vers."). Importantly, this allows clients to read

a valid DTE even after the put operation has replaced the candidate ITE. However, it is wasteful to read the entire DTE when the large value portion is only used if the DTE is valid. Therefore, we add our final optimization which uses multiple RDMA read verbs to access a DTE to avoid unnecessary data transfers (labelled "+ Multi. Read"). The first RDMA read on a DTE is for the key and metadata only and then, if the DTE is valid, a second RDMA read verb is issued for the value field. Taken together, with 16 kB data values, the protocol optimizations produce a 60 % improvement for workloads with 50 % get operations, and a 90 % improvement for workloads with 99 % get operations compared to Nessie without any optimizations.

4.6.3 Energy Consumption

Our last set of experiments examines the effects of server-driven versus client-driven RKVSes on energy consumption, as measured using IPMI [42]. Figure 4.15 shows average energy consumption across all nodes given varying levels of total system load (measured as a percentage of maximum possible throughput) for experiments using 15 nodes, uniform key access distribution, a 16 kB data value size, and a 90 to 10 get operation to put operation ratio. The data point for 100 % maximum throughput represents Nessie, NessieSD, and NessieHY when they are generating requests as fast as possible. We use this value to determine a mean for a Poisson distribution that produces interarrival rates for workloads operating at 5 %, 20 %, 40 %, 60 %, and 80 % of maximum throughput for each system.

NessieSD and NessieHY expend energy on server worker threads which poll and process requests, and client worker threads which make requests and poll for responses. Nessie's energy consumption derives entirely from client worker threads making requests and polling for responses. At 100 % of maximum throughput, each system is saturated with requests. Because every thread in each system is busy either servicing a request or polling, the energy consumption of the systems is the same, at around 139 W. When the hardware is completely idle (no worker threads of any variety are running), its energy consumption is 81 W.

As the percentage of maximum throughput decreases, some client worker threads in each system sit idle when no work is available. This translates into energy savings for all three systems. Nessie, however, decreases its energy consumption at a more rapid rate than NessieHY and NessieSD, thanks to its strictly client-driven design. NessieHY and NessieSD both contain server worker threads which always run at 100 % capacity. These systems therefore consume more energy than Nessie at lower percentages of maximum throughput. NessieSD, which requires more server workers than NessieHY due to its strictly server-driven design, consumes more energy than NessieHY.

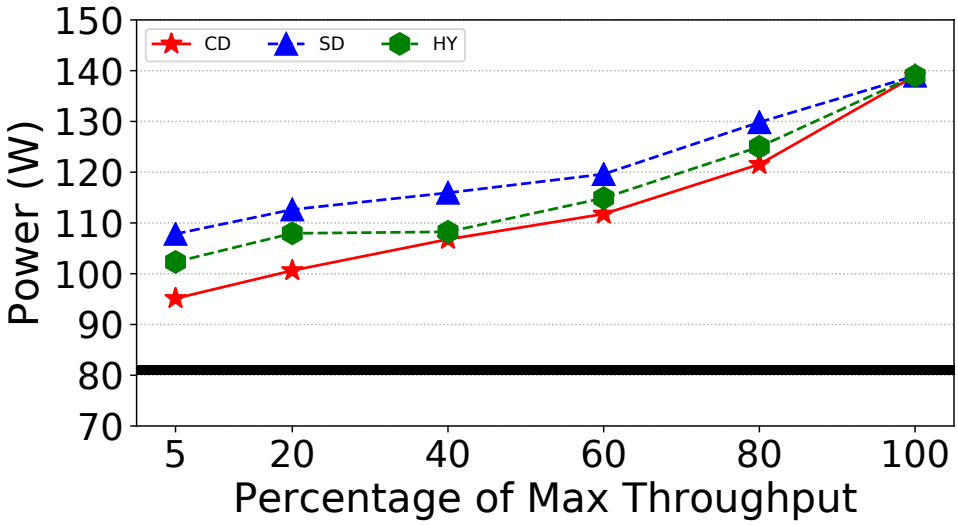

Figure 4.15: Energy consumption for Nessie (CD), NessieSD (SD), and NessieHY (HY) for varying percentages of maximum throughput. Each system uses 15 nodes, and performs uniform random accesses (90 % get operations and 10 % put operations) to 16 kB data values. The dark line at 81 W is the energy consumption of an idle machine.

We now compare the results of our experiments in terms of "percentage over idle" energy consumption. This value is computed by subtracting the baseline idle energy consumption of 81 W from recorded energy consumption values before comparing them in terms of percentage. This helps give us an idea of the relative differences in consumption for each system, ignoring the hardware's idle power consumption (which cannot be changed) and focusing instead on the consumption levels of the systems themselves (which may vary). At 80 % maximum throughput, the difference between the worst performer, NessieSD at 130 W, and the best performer, Nessie at 121 W, is 18 % over the system's idle consumption of 81 W ($1 - \frac{121\,\mathrm{W} - 81\,\mathrm{W}}{130\,\mathrm{W} - 81\,\mathrm{W}} = 18\,\%$). At 20 % maximum throughput, energy consumption is 113 W for NessieSD and 100 W for Nessie, a difference of 41 % over idle consumption. In both cases, NessieHY falls between the other systems. We therefore conclude that for data centre workloads where inactivity can fluctuate, such as the workloads described in Section 4.3.2, a client-driven approach can provide significant energy savings over other approaches.

4.7 Discussion

One concern with Nessie's design is potential contention on NIC resources from multiple requests, from collocated applications, or from other tenants in a cloud environment. This can result in the NIC becoming the system's performance bottleneck. However, in many environments including the cloud, servers are equipped with more than one NIC. Providing Nessie with a dedicated NIC would ensure that it consumes network resources separate from those used by other applications. Additionally, if NIC saturation becomes an issue, a single Nessie node is able to make use of more than one NIC by creating multiple index and data tables, and partitioning them across the NICs. This can help to ensure that NIC resources do not bottleneck the system.

In this chapter we have primarily focused on workloads where data is generated roughly equally across all nodes, memory is not fully utilized, or minor imbalances can be remedied by migrating data between nodes during off peak periods. These workloads benefit from Nessie's local writes in order to improve performance. Because of this focus, we have not added support for remote writes to Nessie, and as a result it is not possible for us to isolate the performance improvements Nessie receives from writing locally. In future work, Nessie's protocol could be updated to support the placement of DTEs non-locally by replacing direct memory writes with RDMA writes and adding a background mechanism to lazily request lists of empty DTEs on remote nodes. This would allow us to support other workloads that do not share these characteristics and to place data according to other criteria, for example by using application-specific knowledge of data locality. Nessie's performance would be different under these circumstances, as the amount of data written over the network would change for put operations based on workload-dependent factors.

The current Nessie design does not support dynamic group membership changes. Failed nodes can be re-added or replaced, but the number of nodes in the system cannot be modified at runtime. This is something that could be examined in future work, perhaps through the implementation of client-driven consistent hashing for data partitioning.

4.8 Chapter Summary

In this chapter we design, implement, and evaluate the performance and energy consumption of Nessie, a high-performance key-value store that uses RDMA. The benefits of this work derive from Nessie's client-driven operations, in addition to its decoupled indexing and storage data structures. Nessie's client-driven architecture eliminates the heavy loads

placed on CPUs by the polling threads used for low-latency, server-driven designs. This allows Nessie to perform favourably in shared-CPU environments compared to an equivalent server-driven approach, more than doubling system throughput. Additionally, the decoupling of index tables and data tables allows Nessie to perform put operations to local memory, which is particularly beneficial in workloads with large data values. This provides Nessie with a 60 % throughput improvement versus a fully server-driven equivalent when data value sizes are 128 kB or larger. Nessie also outperforms a hybrid system when data values are large. Finally, Nessie's client-driven approach allows it to consume less power than a fully server-driven system during periods of non-peak load, reducing power consumption by 18 % compared with idle power consumption at 80 % system utilization and 41 % compared with idle power consumption at 20 % system utilization. Nessie likewise improves power consumption over a hybrid system, albeit to a lesser extent.

Chapter 5

RocketStreams

5.1 Introduction

Live streaming video services, such as Twitch [119] and YouTube Live [134], account for large amounts of Internet traffic [19]. Twitch's volume is of particular note, with Twitch being the fourth-largest consumer of peak Internet traffic in the United States [30, 118]. In 2016 alone, Twitch served over 292 billion minutes of video [32]. It is expected that by 2022, the volume of live streaming video delivered over the Internet will have increased by a factor of 15 compared with 2017, and it will account for 17 % of all Internet video traffic [19].

In order to meet consumer demand, live streaming video services ingest live video streams from producers into data centres (or equivalently IXPs). Requests from consumers are serviced by servers containing replicas of the live streaming video data. The scale of replication for a popular stream can be non-trivial: previous research has shown that with only 30,000 viewers, a Twitch live stream could be dynamically served from up to 150 different servers [23]. Without this replication (which we refer to as dissemination), services would have difficulty achieving high scalability and low latency. Figure 5.1 shows a high-level overview of a live streaming video service in which producers (streamers) upload video to data centres. The streams are disseminated to servers within the same data centre and to nodes in another data centre. The streams are then delivered upon request from servers in either location to consumers (viewers).

Deploying software to accommodate this workflow is a challenging task. Services must either build their own software, which is difficult and time consuming, or use existing

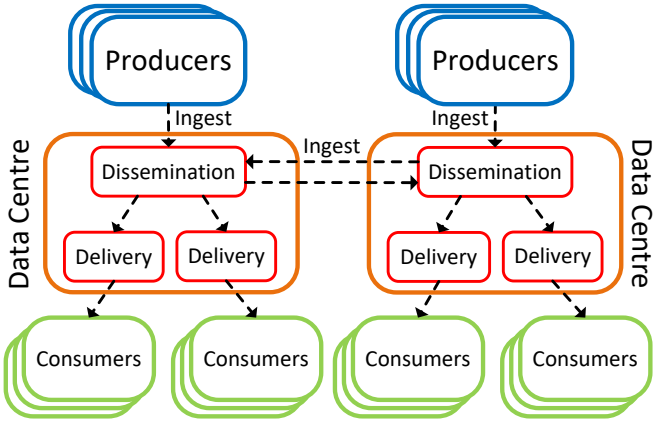

Figure 5.1: An overview of a live streaming video service in which producers (streamers) upload video which is replicated to nodes within and across data centres. The video is then delivered to consumers (viewers).

systems that are not necessarily optimized for live streaming workloads (for example, they often do not provide direct access to managed buffers, resulting in unnecessary copying). Both solutions require the software to be highly efficient, as live streaming video has strict real-time delivery constraints [23], and is disproportionately impacted by losses in quality of experience as users have much higher expectations for live streaming video than video on demand [25]. For example, a 1 % increase in buffering ratio (the fraction of total session time spent buffering) for a 90 minute soccer game translated to viewers watching 3 fewer minutes of the game (thus impacting viewer retention and service revenue) [25].

We posit that services should not be forced to choose between suboptimal performance and a difficult implementation. As a solution, we present RocketStreams, an easy-to-use framework for building efficient and scalable live streaming video dissemination services. The framework, which can be used with new and existing software, provides TCP-based dissemination for replicating video within and across data centres. RocketStreams' API has been designed specifically to facilitate and accommodate live streaming video data, exposing a high-level, event-driven abstraction that eliminates the need for applications to worry about implementing efficient dissemination data management and networking code. The RocketStreams memory buffer abstraction and resulting API calls have been chosen and designed uniquely for their ease of integration with both RDMA and TCP. This abstraction ensures the CPU overhead incurred due to copying is avoided (which is

sometimes required by applications using frameworks built around a sockets abstraction). In addition to providing high TCP performance relative to Redis [87] (which is used as an industry-grade solution for distributed storage by companies such as Pinterest [10], Twitter, GitHub, Snapchat, StackOverflow, Weibo, and Flickr [89]), RocketStreams also provides RDMA-based dissemination which, when appropriate hardware is available, can dramatically reduce CPU utilization and further improve performance and scalability.

5.2 Design

In this section we detail the RocketStreams live stream buffer management API calls, as well as the networking management component, RocketNet, that synchronizes data between buffers on different physical nodes. The intended use case for RocketStreams is for live streaming video services where data moves as follows: data enters the system by being ingested from external producers by logical dissemination nodes (typically inside a data centre or IXP). The data is placed into circular buffers, using RocketStreams, and is disseminated by RocketNet (using framework-managed threads) from the dissemination nodes to logical delivery nodes. Delivery nodes access the data through RocketStreams, and transmit it to consumers on request. We refer to this design pattern as produce-ingest-disseminate-deliver-consume (PIDDC). Figure 5.2 depicts this design pattern, which represents a subset of Figure 5.1. Our current live streaming video implementation supports streamers generating content as producers, and viewers requesting video data as consumers. In the future, the system could be modified to support dissemination nodes producing data to other dissemination nodes, thus facilitating the geo-replication of data.

5.2.1 RocketStreams

RocketStreams is provided as a library implemented in user space. Its API calls are event-driven and can be used to perform dissemination and delivery logic, such as the examples shown in the pseudocode in Listing 5.1 and Listing 5.2. Applications initialize RocketStreams by providing it with a configuration file containing setup details (for example, networking addresses for nodes, the roles and IDs of nodes, and network protocol settings), seen on line 1 of Listing 5.1 and line 1 of Listing 5.2. Once initialized, RocketStreams manages live streaming video buffers on behalf of applications. Ingested data is placed into circular buffers on the dissemination node, one per live stream, in variably sized chunks. This data is disseminated by RocketNet using framework-managed threads created during

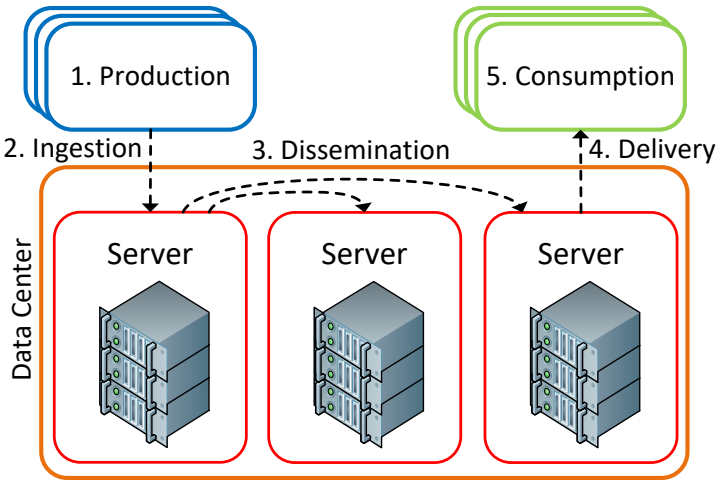

Figure 5.2: An overview of data travelling through a RocketStreams-enabled service according to a produce-ingest-disseminate-deliver-consume (PIDDC) control flow.

initialization. Where dissemination and delivery nodes reside on the same server, network communication is avoided.

We now describe the RocketStreams API. Note that the names provided in this description use our C interface naming convention (`rs_`), while the examples provided in Listing 5.1 and Listing 5.2 use equivalent methods contained within our C++ interface.

Applications call `rs_create_stream` on dissemination nodes, which allocates a dissemination buffer for a live stream and returns a unique ID (across all nodes) for that stream. This is depicted on line 2 of Listing 5.1. The application can choose which delivery nodes (of those specified in the configuration file) receive disseminated chunks from a live stream. This allows live streams to be associated with a flexible number of delivery nodes, per application requirements. This process can be initiated from dissemination nodes or delivery nodes, and in a more complex system would typically be done in consultation with or at the direction of a control plane performing service-wide load balancing. In our simple example, the dissemination node pushes data to a single delivery node. This is set up on line 3 of Listing 5.1 using `rs_set_outbound`.

Applications perform dissemination using a two-phase workflow. In the first phase, memory is requested from a live stream's buffer using `rs_allocate_chunk`, as in line 6 of Listing 5.1. The application provides a requested size and receives a one-shot chunk

```
rs::RsManager rs("config_file.cfg");
rs::RsId id = rs.create_stream("mystream");
rs.set_outbound(delivery_node_endpoint, id);
while (true) {
    size_t size = producer_await_stream(id); // Wait for data.
    rs::RsChunk chunk = rs.allocate_chunk(id, size);
    // Ingest into dissemination buffer.
    producer_receive_data(id, chunk);
    rs.disseminate_chunk(chunk, size, nullptr);
}
```

Listing 5.1: RocketStreams dissemination node example application code.

```
rs::RsManager rs("config_file.cfg");
rs.inbound_callback(on_data_cb);
web_server_run(); // Wait for incoming requests.

void on_data_cb(rs::RsId id, void* data, size_t size) {
    web_server_new_data(id, data, size);
    size_t oldest = web_server_get_oldest(id);
    rs.data_consumed(id, oldest); // Return oldest data.
}
```

Listing 5.2: RocketStreams delivery node example application code.

of memory suitable for writing data to. The chunk is owned by the application, and is returned to the framework during dissemination. Data entering the system through ingestion may be placed there directly by the application in whatever manner it chooses, as is done on line 8 of Listing 5.1. When the application is ready for the data placed in the chunk to be disseminated, it passes the chunk to `rs_disseminate_chunk`, providing the size of the data to disseminate and an optional application-specified callback context. This is done on line 9 of Listing 5.1. At this point, the application no longer owns the chunk (it is being managed by the framework and will be disseminated by RocketNet). `rs_disseminate_chunk` is non-blocking, queuing the chunk for dissemination by RocketNet. Once dissemination is complete, a callback is generated, which is provided with the application's context. The callback method to invoke can be registered by the application using a call to `rs_outbound_callback`. For simplicity, Listing 5.1 omits the callback method and registration.

On delivery nodes, applications register a callback method to be notified of incoming disseminated chunks using `rs_inbound_callback`. When the callback is invoked, it is provided with the ID of the live stream receiving that data, as well as a reference to the data itself. Line 2 of Listing 5.2 shows this registration, with the callback method itself beginning on line 5. The application assumes ownership of data provided to this callback, and can use the associated buffers in whatever manner is needed, such as on line 6 of Listing 5.2, which indicates to a web server that the specified data is ready for delivery. This allows the video data to be sent to clients directly from the framework buffers, avoiding potential copying. When the oldest portion of the stream in the circular buffer is no longer needed, the application returns that memory to the framework by calling `rs_data_consumed` with a size parameter to notify the framework of how much data is being returned. This is seen on line 8 of Listing 5.2.

We believe that the collection of methods in our API provides a design that is highly suitable for applications that wish to disseminate live streaming video. The append-to-buffer model is amenable for handling new live streaming video as it arrives into the system, and access to that data is made easy on delivery nodes as well. Equally important is the fact that the API allows for efficient networking and data access regardless of whether applications use TCP or RDMA. As will be shown later in Section 5.3.1, the ability for applications to use data in-place through RocketStreams reduces memory copying (and therefore CPU utilization) when compared with Redis. This reduction occurs when RocketStreams uses TCP, and is even more pronounced when using RDMA.

5.2.2 RocketNet

RocketNet is responsible for sending and receiving data between buffers on different hosts. It uses framework-managed worker threads and an event-driven model to asynchronously disseminate data between buffers as seen in Figure 5.3. Where required, these threads initiate application-layer callbacks. RocketNet currently supports communication using TCP and RDMA.

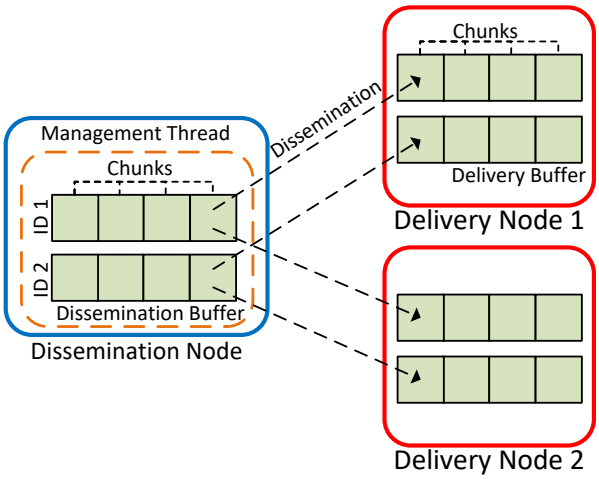

Figure 5.3: Overview of buffer management through RocketStreams. Data placed into managed buffers on dissemination nodes is replicated to delivery nodes by RocketNet using framework-provided threads.

An important consideration for RocketNet is how to make buffer space available for use and re-use for disseminated data. For live streaming video, data is short-lived, and having it available for long periods is not useful (data past a certain age is considered stale and would not be suitable for live video delivery). Thus, RocketNet overwrites the oldest disseminated data in a buffer as new disseminated data arrives. This design avoids overheads due to synchronization, however it makes it possible for data to be overwritten asynchronously on delivery nodes as applications access it (particularly when using RDMA). The framework wraps disseminated chunks with consistency markers, which may be checked by applications to see if data has been modified during the course of serving viewers. This behaviour occurs when the system has been underprovisioned or overloaded, upon which viewers are notified of the error (described in our experiments in Section 5.3.2).

In a real-world scenario, this would be an indicator to perform load balancing by moving either streams or viewers to other nodes.

For TCP, RocketNet's management threads use the `send` and `recv` system calls to direct incoming data from dissemination buffers into delivery node buffers. RocketNet provides RDMA support to achieve a higher level of performance than is achievable with TCP when RDMA-enabled NICs are available. Worker threads perform zero-copy writes by using RDMA write-with-immediate verbs directly from dissemination buffers into delivery node buffers, bypassing both nodes' operating systems and CPUs. The immediate data is used to trigger the callbacks used by RocketStreams, notifying applications of incoming data (as otherwise, RDMA verbs are invisible to the CPU). By avoiding polling mechanisms, CPU resources are conserved for application-related purposes. To reduce RDMA resource contention on the NIC, the buffers used by RocketStreams and RocketNet are allocated from within larger framework-managed RDMA-registered regions of memory, instead of being RDMA-registered individually.

5.2.3 Implementation

Our RocketStreams implementation is coded in C++. We provide a native interface which can be used directly by C and C++ applications (and applications written in other languages with native support). In addition, we provide an interface for socket-based communication with a self-contained version of the framework that exposes delivery buffers through shared memory. This allows language-independent access to the delivery aspect of the framework. Currently RocketStreams supports unicast dissemination, but plans for the future include support for other strategies like tree-based dissemination, and multicast dissemination (via UDP and RDMA).

For our evaluation we have implemented an example dissemination process that performs ingest, and we have also integrated RocketStreams into an open source web server, the userver (to enable delivery to viewers). This integration is depicted in Figure 5.4. As a point of comparison, we also modified the userver to subscribe to and receive data from a Redis [87] broker using the hiredis client library [86]. The userver modifications amount to 248 lines of code for RocketStreams, and 229 lines of code for Redis. Our evaluation in Section 5.3.2 shows that the userver using RocketStreams is able to achieve better performance than it does when using Redis.

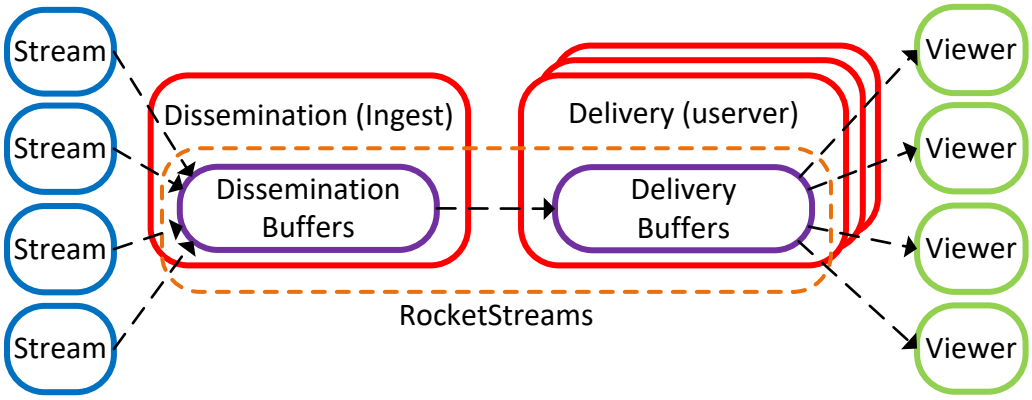

Figure 5.4: An overview of the integration of RocketStreams into the userver depicting how live streaming video is disseminated and then delivered.

5.3 Evaluation

In order to examine the efficiency of our framework, we perform a series of experiments comparing the ingest, dissemination and delivery performance of hosts that use RocketStreams and Redis [87]. Redis is an open source, in-memory data store that has been used by previous research to disseminate live streaming video data [91].

In our experiments, a single host acts as a dissemination node that ingests live streaming video data from multiple simulated producers (running on a separate host). Data is ingested at a bit rate of 2 Mb/s in 500 kB chunks, each containing 2 seconds worth of video data. This is consistent with averages found in literature that examines Twitch-style workloads [84]. Ingested data is pushed to the delivery nodes, which consist of the web server (userver) processes described in Section 5.2.3. Figure 5.4 depicts this setup for RocketStreams, in addition to the viewers used to request data from the web server for our benchmarks in Section 5.3.2.

For experiments with Redis, we take advantage of Redis' publish-subscribe mode to ingest and disseminate data. When using Redis, each dissemination node consists of multiple instances of Redis brokers (one per CPU core), which ingest data from producers. Multiple instances are required to fully utilize available CPU resources, as Redis processes are single-threaded. Delivery nodes consist of web servers integrated with the hiredis client library [86] (which enables communication with the Redis processes on the dissemination

nodes). These delivery nodes subscribe to topics (video streams) on the dissemination node, and therefore receive the data for these topics as it is ingested into Redis on the dissemination node.

Many video streaming services have moved to using encryption to protect the privacy of their viewers (and in the case of live streaming, streamers who may, for example, want to protect paid content from being copied by unauthorized viewers). The desire to help preserve user privacy is great enough that Netflix was willing to perform kernel modifications to enable TLS in their content delivery [101, 102]. Based on this precedent, in our dissemination node experiments, producers use TLS encryption when communicating with the dissemination node. This means that the dissemination node's CPU incurs overhead required for decryption. When conducting delivery node experiments we disable TLS on the producers to allow higher throughput through the dissemination host, since this does not impact the delivery nodes. For our web server experiments, TLS is enabled for all viewers. This places additional strain on the delivery nodes' CPUs. For experiments using Redis, connections between producers and the dissemination node are secured using the stunnel utility (per Redis' recommendation [88]).

For all experiments, our dissemination node runs on a host containing a 2.6 GHz Intel Xeon E5-2660v3 CPU with 10 cores and 512 GB of RAM. All delivery nodes run on separate hosts containing a single 2.0 GHz Intel Xeon D-1540 CPU with 8 cores and 64 GB of RAM. All hosts use 40 Gb/s Mellanox ConnectX-3 NICs, with both the switch and NICs configured to support bandwidths of up to 56 Gb/s [65]. The dissemination host contains four NICs to permit high throughput dissemination to multiple delivery hosts which each contain one NIC. All hosts run Ubuntu 14.04.5 with Linux kernel 4.4.0. We evaluate RocketStreams using both its TCP (Rs-TCP) and RDMA (Rs-RDMA) modes and compare their performance against Redis. Each experiment consists of a 60-second warmup period, followed by a 120-second measurement period.

5.3.1 Microbenchmarks

We first perform a series of microbenchmarks to determine the maximum number of ingested streams (expressed in terms of throughput) that the system can handle as we vary the number of delivery hosts receiving disseminated data. We also observe the impact of handling this incoming data on the CPUs of the delivery hosts. For these microbenchmarks viewers are not issuing requests. This permits us to isolate the impact of disseminated data on the delivery hosts.

Dissemination Host Ingest Throughput: Figure 5.5 shows the maximum sustainable

ingest throughput of the dissemination node while disseminating to different numbers of delivery nodes. To find these throughput values, we increase the number of producers in the system until dissemination fails to keep pace with the rate at which the system ingests data. In these experiments, this corresponds to the point when the dissemination node's CPU is close to 100 % utilization (due to overhead from ingesting encrypted data and disseminating with Redis or RocketStreams).

In all cases, Rs-TCP achieves ingest throughput which is largely comparable to that of Redis (although slightly better). As the number of delivery nodes increases, and therefore the amount of CPU required for dissemination also increases, both Rs-TCP's and Redis' abilities to disseminate the ingested data decline at roughly the same rate. Enabling RDMA for RocketStreams yields a significant boost in maximum ingest throughput versus Redis, which remains fairly consistent even as the number of delivery nodes increases (since RDMA requires little CPU regardless of the amount of disseminated data, those resources can be used instead for ingesting data). With eight delivery nodes, Rs-TCP achieves over 11 Gb/s of ingest throughput, with Redis achieving 10.4 Gb/s. Rs-RDMA achieves 18 Gb/s of ingest throughput, representing an increase of 73 % versus Redis, and simultaneously supports a total of 144 Gb/s of throughput to eight delivery nodes.

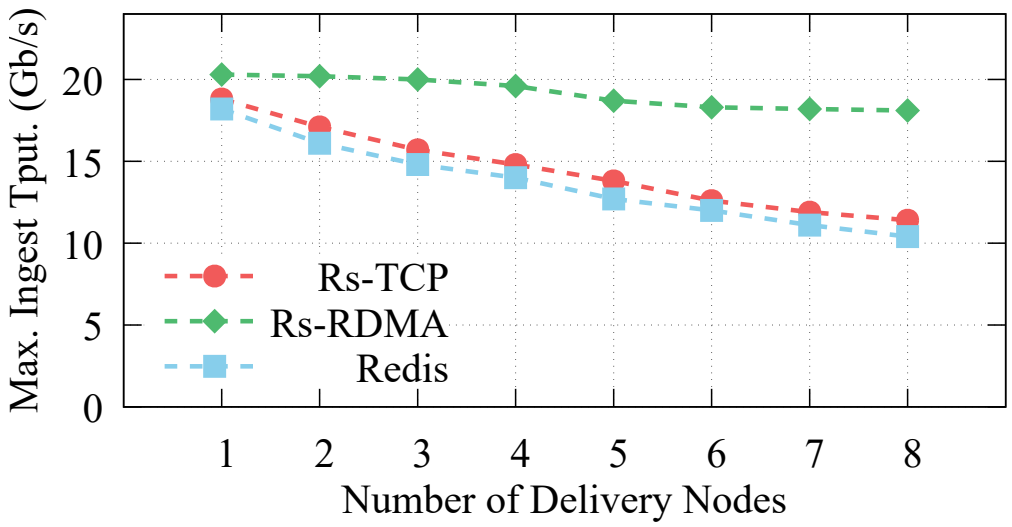

Figure 5.5: Maximum sustainable dissemination host ingest throughput for RocketStreams with TCP (Rs-TCP), RocketStreams with RDMA (Rs-RDMA), and Redis as the number of delivery nodes in the system is increased.

Delivery Host CPU Utilization: Figure 5.6 shows the average percentage of CPU utilization on a delivery host as the amount of disseminated data it receives increases. The results show that RocketStreams requires significantly less CPU on delivery hosts when compared with Redis. For example, at 32 Gb/s, the CPU utilization of Redis is 50 %, whereas for Rs-TCP it is 23 %, a relative reduction of 54 %. By profiling the Redis-based delivery host we found that 35 % of the CPU time is spent in `memcpy`, used both by hiredis internally, and by the server to retrieve data from hiredis. This retrieval is necessary because hiredis frees data immediately after a received data event is handled. This copying is not required when using RocketStreams because, by design, applications have direct access to data inside receive buffers. This data persists until after it is no longer useful (due to the live nature of the video), and it is overwritten by newer video. This provides a sufficiently long period during which applications may directly access the data without needing to copy it first. As discussed in Section 5.2.2, consistency markers prevent data from being used while it is in the process of being overwritten. For Rs-RDMA, CPU utilization is negligible regardless of the amount of data being received by the delivery node. At 32 Gb/s, utilization is capped at 3 % (a 95 % reduction relative to Redis). In Section 5.3.2, we show how these CPU savings allow the system to support higher simultaneous viewer throughput.

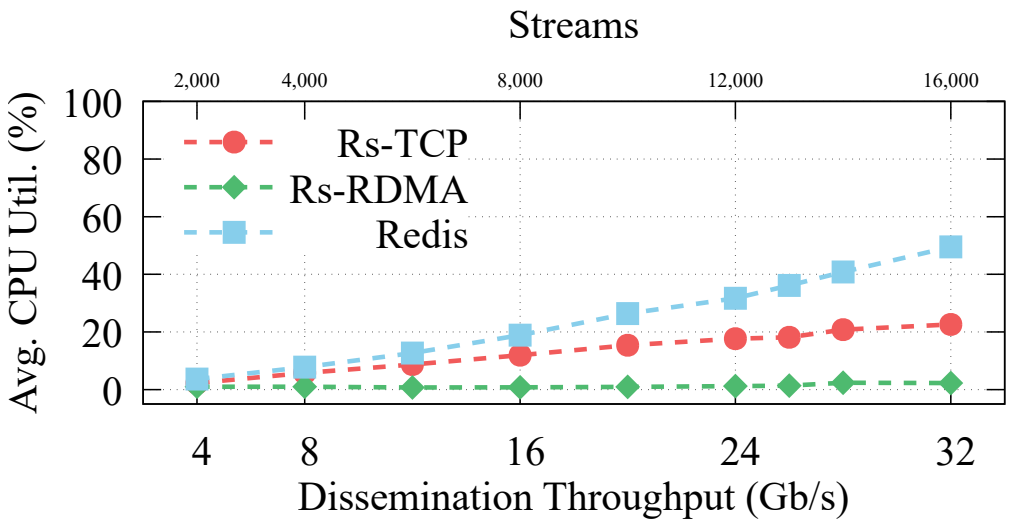

Figure 5.6: Average delivery host CPU utilization percentage (with 8 CPU cores) for RocketStreams with TCP (Rs-TCP), RocketStreams with RDMA (Rs-RDMA), and Redis as incoming dissemination throughput increases. No data is being served to viewers.

5.3.2 Live Streaming Video Benchmarks

We next run a set of benchmarks to determine how dissemination with Redis and dissemination with RocketStreams impact the system's overall capacity to deliver live streaming video to consumers (viewers). For these experiments, we use the application httperf [72] to generate load on the delivery nodes' web servers. This load consists of viewers that use TLS connections to request video content at the rate at which it is produced (2 Mb/s). A sufficient number of hosts is used for load generation to ensure that viewers and the network do not act as bottlenecks. Viewers also check received video data for timeliness and validity. These experiments therefore emulate an end-to-end deployment, with video being produced, ingested, disseminated, and finally delivered to (emulated) viewer sessions. This also implies that the CPU utilization characteristics of these experiments are different from those of Section 5.3.1, as there are increased demands on the delivery nodes due to encryption and content delivery.

When the number of viewers requesting video exceeds the capacity of a delivery node (the web server is not able to meet the liveness constraints of viewer requests), viewers exhaust their playout buffers or receive expired video segments. These occurrences are recorded as errors. For varying numbers of produced streams (measured in terms of their total throughput), we conduct a sequence of experiments to determine the maximum throughput a delivery node can support without any viewers reporting errors. In our experiments, we found that this threshold is reached when the delivery host's CPU is saturated servicing incoming dissemination data, and outgoing delivery requests.

Figure 5.7 shows the results of these experiments. As noted in Section 5.3.1, Rs-TCP uses less CPU than Redis for handling incoming disseminated data by avoiding copying, and this discrepancy grows as the amount of disseminated data increases. As a result, for 20 Gb/s of incoming disseminated data, while the userver using Redis is only able to serve 13.5 Gb/s of video to viewers, the userver using Rs-TCP achieves over 17 Gb/s, a relative increase of 27 %. The userver using Rs-RDMA for dissemination achieves delivery throughput of 21 Gb/s, regardless of dissemination throughput (55 % higher than Redis).

5.3.3 Summary

We have seen, through microbenchmarks in Section 5.3.1, that on dissemination nodes RocketStreams with TCP uses an amount of CPU that is comparable to that of Redis. As a result, both Redis and RocketStreams with TCP sustain similar levels of ingest throughput as the number of delivery nodes being serviced increases. On delivery nodes,

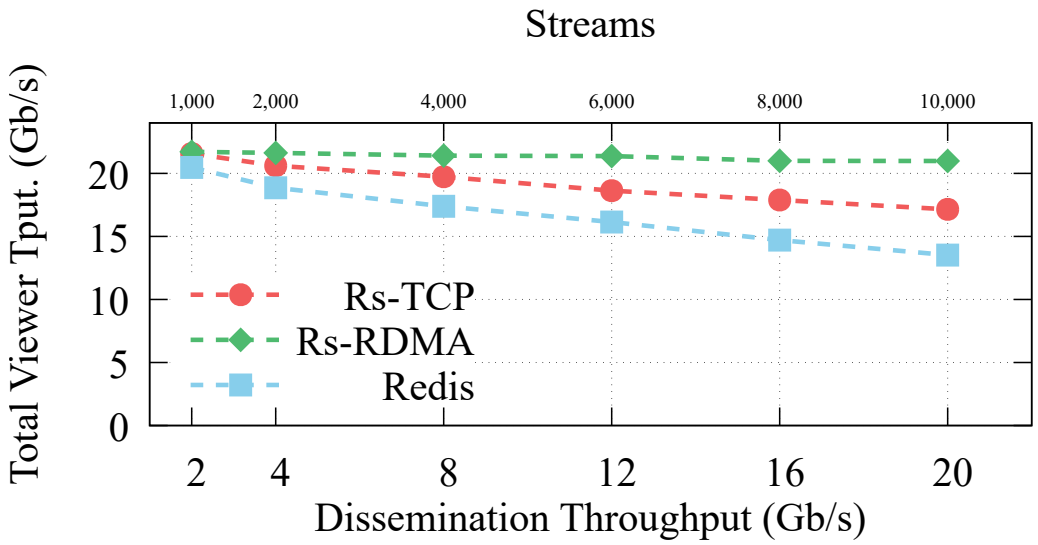

Figure 5.7: Maximum error-free viewer throughput supported by the userver on a delivery host for RocketStreams with TCP (Rs-TCP), RocketStreams with RDMA (Rs-RDMA), and Redis as incoming dissemination throughput increases.

however, RocketStreams with TCP uses less CPU than Redis for the same amount of dissemination throughput. With RDMA enabled, RocketStreams uses substantially less CPU than Redis, on both dissemination and delivery nodes. The net result is that, as shown in Section 5.3.2, delivery nodes using RocketStreams (in both RDMA and TCP modes) are able to achieve and sustain higher levels of viewer throughput than delivery nodes using Redis. Furthermore, as explained in Section 5.2.3, the number of lines of code required to integrate the userver with RocketStreams is similar to the number of lines of code required to integrate Redis. We therefore conclude that RocketStreams meets our stated goals, both in terms of performance and usability.

5.4 Chapter Summary

In this chapter we introduce RocketStreams, a framework for efficiently handling live streaming video dissemination. We highlight RocketStreams' easy-to-use design, and how it eliminates the need for applications to implement their own conceptually and technically difficult data management and networking code. By providing direct access to framework-managed buffers, RocketStreams eliminates the need to copy data and thereby provides

performance similar to, or better than, Redis. We modify a web server, the userver, to access disseminated live streaming video data through RocketStreams, and use load generators to evaluate its ingest and delivery capacities relative to those of Redis. Our benchmarks show that RocketStreams provides similar dissemination performance to Redis, and on delivery nodes it reduces CPU utilization by up to 54 % and thereby increases viewer throughput by up to 27 %. RocketStreams also provides RDMA capabilities, which allows it to support up to 73 % higher ingest traffic on dissemination nodes. Likewise, on delivery nodes, RDMA-enabled RocketStreams reduces CPU utilization by 95 % compared with Redis, and increases simultaneous viewer throughput by 55 %.

Potential future work for RocketStreams, including multicast dissemination support and extensions to support other targeted workloads such as publish-subscribe and message queuing, is described in greater detail in Section 6.2.5.

Chapter 6

Conclusions and Future Work

6.1 Thesis Summary

In this thesis, we examine techniques for improving the hardware utilization, and therefore performance, of component systems of content delivery services. Increasingly, these services must adapt to modern circumstances under which growing content is sent to expanding numbers of users. We find that many systems are ill-equipped to meet modern performance requirements, and attempt to address the underlying issues present in these systems.

We first create a library named Libception to increase the throughput of HTTP streaming video web servers that serve video on demand content from disk. Next, we develop an RDMA-enabled key-value store protocol, Nessie, that uses CPU exclusively on nodes that issue requests, and not on nodes from which requests are serviced. Finally, we introduce a platform called RocketStreams that allows for the easy and resource-efficient dissemination of live streaming video, and use it to improve the scalability of an HTTP web server, the userver. In addition to describing the techniques we use in each project to achieve higher performance, we also provide empirical evidence through evaluation to demonstrate these performance improvements.

6.1.1 Libception

Despite streaming video now being the largest contributor to Internet traffic [19], the web servers used to serve it perform very poorly by default. In past research, we showed that directly modifying a custom web server, the userver, on FreeBSD to support serialization

and aggressive prefetching resulted in a large improvement in system throughput, owing to a more workload-appropriate use of the hard disk [107]. Unfortunately, bringing these modifications to widely used web servers like Apache and nginx is substantially more time consuming, as their code bases are large and complex. Furthermore, the technique used to improve the userver is FreeBSD-specific, and cannot be easily ported across operating systems.

To avoid the limitations of our previous approach, and to make efficient disk techniques available to widely used web servers and other applications, we implement a user space library named Libception. Libception can be dynamically linked with unmodified applications at launch time to provide them with the benefits obtained from aggressive prefetching and serialization. Libception uses techniques that are portable across multiple operating systems, and requires no application code changes in order to be used. We use Libception to double the peak HTTP streaming video delivery throughput of the Apache, nginx, and userver web servers. Furthermore, by examining kernel source code, we are able to understand poor default Linux kernel performance, and find and tune Linux kernel parameters to obtain similar levels of performance to those achieved with Libception. Finally, we also demonstrate Libception's potential utility for other workloads through microbenchmarking with the diff utility.

6.1.2 Nessie

In-memory key-value stores are used by content delivery services to provide fast, distributed access to data that would be too slow to effectively retrieve from disk. In order to further improve performance, many modern key-value stores have been implemented with RDMA-enabled protocols [70, 26, 47, 127]. Despite this, existing systems have implementations that may not perform well for three particular workloads, namely workloads involving large data values, workloads that are run in environments with shared-CPU resources, and workloads that wish to reduce energy consumption.

To address these issues, we design a key-value store protocol called Nessie that is entirely client-driven and decouples its indexing metadata from its stored data. Next, we evaluate our protocol against similar server-driven and hybrid protocols with varied levels of data coupling. Our findings indicate that Nessie's client-driven, decoupled nature improves throughput by 60 % relative to a fully server-driven, tightly coupled equivalent when data values are large. We show that in environments with CPU interference from other processes, Nessie's client-driven protocol is more resilient than similar server-driven and hybrid protocols, more than doubling relative system throughput as CPU contention

increases. Finally, we find that Nessie reduces energy consumption by up to 41 % (over idle power consumption) versus a server-driven design as the system becomes less utilized.

6.1.3 RocketStreams

Live streaming video is an increasingly dominant presence on the modern Internet [19]. Over the past few years, services delivering live streaming video have seen an explosion in growth which is expected to continue [19]. When compared with services that provide video on demand content, live streaming video services face unique challenges. Real-time delivery constraints, a lack of ability to pre-cache live video, and higher user expectations [25, 23] make efficient delivery imperative for live streaming video services. However, implementing code to manage and efficiently disseminate live streaming video can be complicated and time consuming.

To this end, we introduce RocketStreams, a framework that enables efficient live streaming video dissemination. RocketStreams exposes an API to applications that is tailored for managing live streaming video, and allows applications to disseminate and access disseminated data without needing to write complex data management and networking code. Using an implementation of our framework, we augment an existing web server, the userver, to be able to take advantage of these facilities. We find that, when compared with the industry-used Redis data store (which required the addition of and changes to a similar number of lines of code as RocketStreams), RocketStreams using TCP reduces CPU utilization on delivery nodes by 54 % versus Redis, resulting in a 27 % increase in simultaneous viewer throughput. When RDMA is available, RocketStreams reduces CPU utilization on delivery nodes by 95 % versus Redis, resulting in a 55 % increase in simultaneous viewer throughput.

6.2 Future Work

In this section we discuss some potential avenues for future work deriving from this thesis.

6.2.1 Other Libception Workloads

In Chapter 3, we explored Libception primarily through its utility in improving the throughput of HTTP streaming video server workloads. Other workloads could also see improvements from the use of Libception. For example, web servers are used to serve other types

of large, static content, including software (such as video games and operating systems). Whereas video on demand content may occupy hundreds of megabytes, software can occupy gigabytes or even tens of gigabytes, and therefore serialization and prefetching would be good candidates for improving the disk-based performance of web servers delivering software. Outside of large content, however, we also provide some evidence in Section 3.4.9 that Libception is also able to improve the runtime of a microbenchmark consisting of two copies of the diff utility comparing versions of the Linux kernel (which is mostly made up of files with sizes in the kilobyte range). It would therefore also be useful to characterize how large or concurrent a disk-bound workload must be in order to benefit from aggressive prefetching and serialization.

Additionally, storage mediums have continued to evolve since Libception was developed. Although we provide some minor insights into Libception's performance impact on two disks in Section 3.4.8, it would be useful to further characterize performance as the number of disks continues to grow, as this has a notable impact on how much prefetched data is able to retained in the file cache relative to how much data is being retrieved from disk. Additionally, it would be useful to characterize how Libception interacts with solid state drives and to explore new user space techniques for improving the throughput of applications using solid state drives. Likewise, it would be interesting to characterize workloads that use a mixture of hard drives and solid state drives.

6.2.2 Intelligent Prefetching

The size of aggressive prefetching performed by Libception in Chapter 3 is configured at launch time, and therefore does not actively changed based on workload feedback and system properties. However, in previous research we showed that dynamically choosing a prefetch size at runtime can have substantial benefits for disk throughput [107]. The algorithm advanced in that work relies on file cache miss ratios and read times from disk in order to calculate good prefetch sizes to use. These metrics are system-wide, and Libception already makes use of file cache residency to determine whether or not perform prefetching. We believe that, with appropriate adjustments, Libception should be able to take advantage of automated prefetch sizing based on system data. Furthermore, we believe that this change should be possible in a manner that is invisible to the application embedding Libception, preserving Libception's ability to provide throughput gains without requiring application code modification.

6.2.3 Nessie Protocol

Although in Chapter 4 we show that the Nessie protocol performs well for certain types of workloads, in the time since it was developed, several possibilities for improvements have arisen which we believe would be beneficial to explore in future work. For example, Nessie must use RDMA read verbs to access index tables regardless of locality, as the RDMA compare-and-swap verbs used to update index table entries are only atomic relative to other RDMA verbs. As hardware transactional memory (HTM) increases in availability, however, Nessie could be modified to take advantage of HTM in a manner similar to DrTM [127]. It would also be useful to explore options for adding and removing nodes from a Nessie system, a feature which the current protocol does not address.

Another candidate for protocol modification in future work is reducing the number of RDMA verbs required by Nessie for its various operations. With N-way cuckoo hashing, most put operations (for example) will have to access all N index table entries, and some multiple times. FaRM [26] has demonstrated that hopscotch hashing can be used effectively with RDMA in a hybrid protocol, and helps to keep the number of required round trips for an operation low. Exploring the intricacies of integrating hopscotch hashing (or another equally efficient form of hashing) into a fully client-driven protocol, like Nessie's, is a good opportunity for future research.

6.2.4 Nessie Reliability

The Nessie protocol presented in Chapter 4 does not address reliability in its design. We believe that this would be an important avenue for future work. Although it would be possible to modify the protocol such that data table and index table elements are duplicated to disk, if not done properly this could have a serious performance impact on the system. Another potentially better alternative could involve the use of non-volatile memory, as has been explored by systems like Octopus [63], Orion [133], or the follow-up research to FaRM [27].

6.2.5 Extending RocketStreams

In Chapter 5, RocketStreams is used in conjunction with its networking component, RocketNet, to provide applications with the ability to easily disseminate live streaming video. Due to the general applicability of the produce-ingest-disseminate-deliver-consume (PIDDC) design pattern discussed in Section 5.2, however, we envision that RocketStreams and

RocketNet could be extended to accommodate a host of other use cases. By modifying the system to support other features, more PIDDC-style workloads could be handled by RocketStreams and RocketNet, for example publish-subscribe systems and message queues. Additionally, extending our exploration of live streaming video should also prove useful. Our evaluation uses a static video bit rate, and disseminates video equally across delivery nodes, from which it is delivered to viewers uniformly requesting the data. A future benchmark could provide a more detailed evaluation with more representative workloads.

Finally, another avenue for future work would be to extend the networking capabilities of the framework. For example, by using UDP or RDMA multicast, RocketNet could reduce outgoing bandwidth overhead due to dissemination, thus increasing the number of delivery nodes that can be supported. However, this could come with potential drawbacks as well (for example, RDMA multicast verbs must be performed over unreliable connections and are limited to sizes much smaller than those of RDMA verbs on reliable connections). Exploring the trade-offs would help to determine under which conditions (if any) RocketStreams benefits from multicast.

6.3 Concluding Remarks

Content delivery services form the backbone of today's Internet, but the systems they use face increasingly stringent performance requirements as user bases expand and content sizes grow. Without careful consideration for modern workloads, the components of these systems perform poorly and run the risk of unnecessarily driving up hardware costs for the services employing them, especially as they scale out to meet demand. Making efficient use of hardware resources is therefore crucial for services that want to remain competitive in a growing and shifting market.

This thesis describes research on how to improve the performance of three major components of content delivery services. Using a portable library named Libception, we show that it is possible to introduce aggressive disk prefetching and request serialization into unmodified industry grade web servers, improving their disk utilization and more than doubling system throughput on both Linux and FreeBSD. With our distributed, in-memory key-value store Nessie, we describe an RDMA-based, client-driven protocol that reduces idle energy consumption through the elimination of polling server threads. Nessie is also resilient to performance degradation caused by CPU interference, and performs well compared to alternative system designs when dealing with large data values. Finally, with RocketStreams, we introduce a framework that exposes an easy-to-use API through which applications can implement efficient live streaming video dissemination. We also show that,

when integrated into a web server, the framework provides better performance than Redis, and improves performance even further when RDMA-enabled hardware is available.

Across all three projects, we have demonstrated that, through careful and workload-appropriate use of hardware, better performance is achievable with already-existing resources. We believe that, overall, the contributions of this thesis can help to pave the way for content delivery services whose component systems make better, more efficient use of underlying hardware. We envision that these services will benefit greatly from these improvements, as their user bases, content, and objectives continue to evolve.

References

[1] Vijay Kumar Adhikari, Yang Guo, Fang Hao, Matteo Varvello, Volker Hilt, Moritz Steiner, and Zhi-Li Zhang. Unreeling Netflix: Understanding and improving multi-CDN movie delivery. In *Proc. International Conference on Computer Communications (INFOCOM)*, pages 1620–1628. IEEE, 2012.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. Remote regions: A simple abstraction for remote memory. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 775–787. USENIX, 2018.

[3] Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing messages while sharing memory. In *Proc. Symposium on Principles of Distributed Computing (PODC)*, pages 51–60. ACM, 2018.

[4] Apache Software Foundation. Apache HTTP server project. https://httpd.apache.org. Accessed April 12, 2019.

[5] Apache Software Foundation. RDD programming guide. https://spark.apache.org/docs/latest/rdd-programming-guide.html. Accessed August 26, 2019.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proc. Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS / Performance)*, pages 53–64. ACM SIGMETRICS / IFIP Performance, 2012.

[7] Jens Axboe. Linux block IO — present and future. In *Proc. Ottawa Linux Symposium (OLS)*, pages 51–61, 2004.

[8] Pavan Balaji, Hemal V. Shah, and Dhabaleswar K. Panda. Sockets vs RDMA interface over 10-gigabit networks: An in-depth analysis of the memory traffic bottleneck.

In *Proc. RDMA Applications, Implementations, and Technologies Workshop (RAIT)*, pages 1–21. IEEE, 2004.

[9] Ali Begen, Tankut Akgul, and Mark Baugher. Watching video over the web: Part 1: Streaming protocols. *IEEE Internet Computing (IC)*, 15(2):54–63, 2011.

[10] Adam Bloom. Using Redis at Pinterest for billions of relationships. https://blog.pivotal.io/pivotal/case-studies/using-redis-at-pinterest-for-billions-of-relationships, 2013. Accessed April 12, 2019.

[11] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proc. International Conference on Measurement and Modeling of Computer Systems*, pages 157–168. ACM SIGMETRICS, 2005.

[12] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems (TOCS)*, 14(4):311–343, 1996.

[13] Benjamin Cassell, Huy Hoang, and Tim Brecht. RocketStreams: A framework for the efficient dissemination of live streaming video. In *Proc. Asia-Pacific Workshop on Systems (APSys)*, pages 84–90. ACM SIGOPS, 2019.

[14] Benjamin Cassell, Tyler Szepesi, Tim Brecht, Derek Eager, Jim Summers, and Bernard Wong. Disk prefetching mechanisms for increasing HTTP streaming video server throughput. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):7:1–7:30, 2018.

[15] Benjamin Cassell, Tyler Szepesi, Bernard Wong, Tim Brecht, Jonathan Ma, and Xiaoyi Liu. Nessie: A decoupled, client-driven, key-value store using RDMA. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(12):3537–3552, 2017.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[17] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proc. European Conference on Computer Systems (EuroSys)*, pages 26:1–26:17. ACM, 2016.

[18] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proc. Conference on SIGCOMM*, pages 231–242. ACM SIGCOMM, 2013.

[19] Cisco. Cisco visual networking index: Forecast and trends, 2017–2022 white paper, 2018.

[20] Mark Claypool, Daniel Farrington, and Nicholas Muesch. Measurement-based analysis of the video characteristics of Twitch.tv. In *Proc. Games Entertainment Media Conference (GEM)*, pages 1–4. IEEE, 2015.

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. Symposium on Cloud Computing (SoCC)*, pages 143–154. ACM, 2010.

[22] Bhavesh Davda and Josh Simons. RDMA on vSphere: Update and future directions. In *Proc. OpenFabrics Alliance (OFA) Workshop*. OpenFabrics Alliance, 2012.

[23] Jie Deng, Gareth Tyson, Felix Cuadrado, and Steve Uhlig. Internet scale user-generated live video streaming: The Twitch case. In *Proc. Passive and Active Measurement Conference (PAM)*, pages 60–71. Springer, 2017.

[24] Sudhir N. Dhage, Smita K. Patil, and B. B. Meshram. Survey on: Interactive video-on-demand (VoD) systems. In *Proc. International Conference on Circuits, Systems Communication and Information Technology Applications (CSCITA)*, pages 435–440. IEEE, 2014.

[25] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. In *Proc. Conference on SIGCOMM*, pages 362–373. ACM SIGCOMM, 2011.

[26] Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. FaRM: Fast remote memory. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414. USENIX, 2014.

[27] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 54–70. ACM, 2015.

[28] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pages 371–384. USENIX, 2013.

[29] Alessandro Finamore, Marco Mellia, Maurizio M Munafò, Ruben Torres, and Sanjay G Rao. YouTube everywhere: Impact of device and infrastructure synergies on user experience. In *Proc. Internet Measurement Conference (IMC)*, pages 345–360. ACM SIGCOMM, 2011.

[30] Drew FitzGerald and Daisuke Wakabayashi. Apple quietly builds new networks. https://www.wsj.com/articles/apple-quietly-builds-new-networks-1391474149, 2014. Wall Street Journal. Accessed April 12, 2019.

[31] Brad Fitzpatrick. Distributed caching with memcached. *Belltown Media Linux Journal*, 2004(124):1–5, 2004.

[32] Evan Freitas. Presenting the Twitch 2016 year in review. https://blog.twitch.tv/presenting-the-twitch-2016-year-in-review-b2e0cdc72f18, 2017. Accessed April 12, 2019.

[33] Debasish Ghose and Hyoung Joong Kim. Scheduling video streams in video-on-demand systems: A survey. *Springer Multimedia Tools and Applications*, 11(2):167–195, 2000.

[34] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. YouTube traffic characterization: A view from the edge. In *Proc. Internet Measurement Conference (IMC)*, pages 15–28. ACM SIGCOMM, 2007.

[35] Paul Grun, Sean Hefty, Sayantan Sur, David Goodell, Robert D. Russell, Howard Pritchard, and Jeffrey M. Squyres. A brief introduction to the OpenFabrics interfaces — a new network API for maximizing high performance application efficiency. In *Proc. Symposium on High-Performance Interconnects (HOTI)*, pages 34–39. IEEE, 2015.

[36] Qiyun He, Jiangchuan Liu, Chonggang Wang, and Bo Li. Coping with heterogeneous video contributors and viewers in crowdsourced live streaming: A cloud-based approach. *IEEE Transactions on Multimedia*, 18(5):916–928, 2016.

[37] Qiyun He, Cong Zhang, Xiaoqiang Ma, and Jiangchuan Liu. Fog-based transcoding for crowdsourced video livecast. *IEEE Communications Magazine*, 55(4):28–33, 2017.

[38] Brandon Heller, Srinivasan Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving energy in data center networks. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pages 249–264. USENIX, 2010.

[39] Masoud Hemmatpour, Bartolomeo Montrucchio, and Maurizio Rebaudengo. Communicating efficiently on cluster-based remote direct memory access (RDMA) over InfiniBand protocol. *MDPI Applied Sciences*, 8(11):2034:1–2034:17, 2018.

[40] Tayler H. Hetherington, Mike O'Connor, and Tor M. Aamodt. MemcachedGPU: Scaling-up scale-out key-value stores. In *Proc. Symposium on Cloud Computing (SoCC)*, pages 43–57. ACM, 2015.

[41] Jian Huang, Xiangyong Ouyang, Jithin Jose, Md. Wasi-ur-Rahman, Hao Wang, Miao Luo, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. High-performance design of HBase with RDMA over InfiniBand. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, pages 774–785. IEEE, 2012.

[42] Intel. Intelligent platform management interface (IPMI). https://www.intel.com/content/www/us/en/servers/ipmi/ipmi-home.html. Accessed April 12, 2019.

[43] Nusrat S. Islam, Md. Wasi-ur-Rahman, Jithin Jose, Raghunath Rajachandrasekar, Hao Wang, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. High performance RDMA-based design of HDFS over InfiniBand. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 35:1–35:35. IEEE, 2012.

[44] Song Jiang, Xiaoning Ding, Yuehai Xu, and Kei Davis. A prefetching scheme exploiting both data layout and access history on disk. *ACM Transactions on Storage (TOS)*, 9(3):10:1–10:23, 2013.

[45] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md. Wasi-ur-Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K Panda. Scalable memcached design for InfiniBand clusters using hybrid transports. In *Proc. International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 236–243. IEEE, 2012.

[46] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur-Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, et al. memcached design on high performance RDMA capable interconnects. In *Proc. International Conference on Parallel Processing (ICPP)*, pages 743–752. IEEE, 2011.

[47] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *Proc. Conference on SIGCOMM*, pages 295–306. ACM SIGCOMM, 2014.

[48] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 437–450. USENIX, 2016.

[49] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pages 185–201. USENIX, 2016.

[50] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs can be general and fast. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–16. USENIX, 2019.

[51] Mangesh Kasbekar. On efficient delivery of web content. https://www.sigmetrics.org/sigmetrics2010/greenmetrics/MangeshKasbekar.pdf, 2010. GreenMetrics Keynote Talk. Accessed August 15, 2019.

[52] Alan "Dormando" Kasindorf. memcached — a distributed memory object caching system. https://memcached.org. Accessed April 12, 2019.

[53] Mehdi Kaytoue, Arlei Silva, Loïc Cerf, Wagner Meira, Jr., and Chedy Raïssi. Watch me playing, I am a professional: A first study on video game live streaming. In *Proc. International Conference on World Wide Web (WWW)*, pages 1181–1188. ACM, 2012.

[54] Alexey Khrabrov and Eyal de Lara. Accelerating complex data transfer for cluster computing. In *Proc. Workshop on Hot Topics in Cloud Computing (HotCloud)*, pages 1–6. USENIX, 2016.

[55] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 390–405. ACM, 2017.

[56] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review (OSR)*, 44(2):35–40, 2010.

[57] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-performance in-memory key-value store with programmable NIC. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 137–152. ACM, 2017.

[58] Chuanpeng Li, Kai Shen, and Athanasios E. Papathanasiou. Competitive prefetching for concurrent sequential I/O. In *Proc. European Conference on Computer Systems (EuroSys)*, pages 189–202. ACM, 2007.

[59] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proc. Symposium on Cloud Computing (SoCC)*, pages 1–15. ACM, 2014.

[60] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444. USENIX, 2014.

[61] Xiaoyi Lu, Nusrat S. Islam, Md. Wasi-ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K. Panda. High-performance design of Hadoop RPC with RDMA over InfiniBand. In *Proc. International Conference on Parallel Processing (ICPP)*, pages 641–650. IEEE Computer Society, 2013.

[62] Xiaoyi Lu, Md. Wasi-ur-Rahman, Nusrat Islam, Dipti Shankar, and Dhabaleswar K. Panda. Accelerating Spark with RDMA for big data processing: Early experiences. In *Proc. Symposium on High-Performance Interconnects (HOTI)*, pages 9–16. IEEE, 2014.

[63] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: An RDMA-enabled distributed persistent memory file system. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 773–785. USENIX, 2017.

[64] Nick Mathewson, Azat Khuzhin, and Niels Provos. libevent — an event notification library. https://libevent.org. Accessed April 12, 2019.

[65] Mellanox. How to configure 56GbE link on Mellanox adapters and switches. https://community.mellanox.com/s/article/howto-configure-56gbe-link-on-mellanox-adapters-and-switches. Accessed April 12, 2019.

[66] Mellanox. RDMA aware networks programming user manual rev 1.7. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf. Accessed April 12, 2019.

[67] Babar Naveed Memon, Xiayue Charles Lin, Arshia Mufti, Arthur Scott Wesley, Tim Brecht, Kenneth Salem, Bernard Wong, and Benjamin Cassell. RaMP: A lightweight RDMA abstraction for loosely coupled applications. In *Proc. Workshop on Hot Topics in Cloud Computing (HotCloud)*, pages 1–6. USENIX, 2018.

[68] Microsoft. Detours. https://github.com/microsoft/detours. Accessed April 12, 2019.

[69] David L. Mills. Internet time synchronization: The network time protocol. *IEEE Transactions on Communications (TCOM)*, 39(10):1482–1493, 1991.

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

[71] Michael Mitzenmacher. Some open questions related to cuckoo hashing. In *Proc. European Symposium on Algorithms (ESA)*, pages 1–10. Springer, 2009.

[72] David Mosberger and Tai Jin. httperf — a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review (PER)*, 26(3):31–37, 1998.

[73] Netflix. Appliance hardware. https://openconnect.netflix.com/en_gb/hardware. Accessed April 12, 2019.

[74] Netflix. Appliance software. https://openconnect.netflix.com/en_gb/software. Accessed April 12, 2019.

[75] Netflix. Requirements for deploying embedded appliances. https://openconnect.netflix.com/en_gb/requirements-for-deploying. Accessed April 12, 2019.

[76] nginx. nginx. https://www.nginx.com. Accessed April 12, 2019.

[77] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, et al. Storm: A fast transactional dataplane for remote data structures. In *Proc. International Conference on Systems and Storage (SYSTOR)*, pages 97–108. ACM, 2019.

[78] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review (OSR)*, 43(4):92–105, 2010.

[79] R. Pagh and F. Rodler. Cuckoo hashing. *Elsevier Journal of Algorithms*, 51(3):122–144, 2004.

[80] George Panagiotakis, Michail D. Flouris, and Angelos Bilas. Reducing disk I/O performance sensitivity for large numbers of sequential streams. In *Proc. International Conference on Distributed Computing Systems (ICDCS)*, pages 22–31. IEEE, 2009.

[81] Athanasios E. Papathanasiou and Michael L. Scott. Aggressive prefetching: An idea whose time has come. In *Proc. Workshop on Hot Topics in Operating Systems (HotOS)*, pages 1–5. USENIX, 2005.

[82] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R. Cheriton. Comparing the performance of web server architectures. In *Proc. European Conference on Computer Systems (EuroSys)*, pages 231–243. ACM, 2007.

[83] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 79–95. ACM, 1995.

[84] Karine Pires and Gwendal Simon. DASH in Twitch: Adaptive bitrate streaming in live game streaming platforms. In *Proc. Workshop on Design, Quality and Deployment of Adaptive Video Streaming (VideoNext)*, pages 13–18. ACM, 2014.

[85] Karine Pires and Gwendal Simon. YouTube Live and Twitch: A tour of user-generated live streaming systems. In *Proc. Multimedia Systems Conference (MMSys)*, pages 225–230. ACM, 2015.

[86] Redis. hiredis. https://github.com/redis/hiredis. Accessed April 12, 2019.

[87] Redis. Redis. https://redis.io. Accessed April 12, 2019.

[88] Redis. Securing connections with SSL/TLS. https://docs.redislabs.com/latest/rc/securing-redis-cloud-connections. Accessed July 8, 2019.

[89] Redis. Who's using Redis? https://redis.io/topics/whos-using-redis. Accessed August 26, 2019.

[90] Luigi Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review (CCR)*, 27(1):31–41, 1997.

[91] Luis Rodríguez-Gil, Javier García-Zubia, Pablo Orduña, and Diego López-de Ipiña. An open and scalable web-based interactive live-streaming architecture: The WILSP platform. *IEEE Access*, 5:9842–9856, 2017.

[92] Yaoping Ruan and Vivek S. Pai. Making the "box" transparent: System call performance as a first-class result. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 1–14. USENIX, 2004.

[93] Yaoping Ruan and Vivek S. Pai. Understanding and addressing blocking-induced network server latency. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 143–156. USENIX, 2006.

[94] Chris Ruemmler and John Wilkes. UNIX disk access patterns. In *Proc. USENIX Winter Conference*, pages 405–420. USENIX, 1993.

[95] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[96] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's time for low latency. In *Proc. Workshop on Hot Topics in Operating Systems (HotOS)*, pages 1–5. USENIX, 2011.

[97] Sandvine. Global Internet phenomena report, 2016.

[98] Sandvine. Global Internet phenomena report, 2018.

[99] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pages 69–87. USENIX, 2018.

[100] Ralf Steinmetz. Multimedia file systems survey: Approaches for continuous media disk scheduling. *Elsevier Computer Communications*, 18(3):133–144, 1995.

[101] Randall Stewart, John-Mark Gurney, and Scott Long. Optimizing TLS for high-bandwidth applications in FreeBSD. Technical report, Netflix, 2015.

[102] Randall Stewart and Scott Long. Improving high-bandwidth TLS in the FreeBSD kernel. Technical report, Netflix, 2016.

[103] Denny Stohr, Tao Li, Stefan Wilk, Silvia Santini, and Wolfgang Effelsberg. An analysis of the YouNow live streaming platform. In *Proc. Local Computer Networks Conference Workshops (LCN Workshops)*, pages 673–679. IEEE, 2015.

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

[105] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is faster than server-bypass with RDMA. In *Proc. European Conference on Computer Systems (EuroSys)*, pages 1–15. ACM, 2017.

[106] Jim Summers, Tim Brecht, Derek Eager, and Alex Gutarin. Characterizing the workload of a Netflix streaming video server. In *Proc. International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2016.

[107] Jim Summers, Tim Brecht, Derek Eager, Tyler Szepesi, Benjamin Cassell, and Bernard Wong. Automated control of aggressive prefetching for HTTP streaming video servers. In *Proc. International Conference on Systems and Storage (SYSTOR)*, pages 5:1–5:11. ACM, 2014.

[108] Jim Summers, Tim Brecht, Derek Eager, and Bernard Wong. Methodologies for generating HTTP streaming video workloads to evaluate web server performance. In *Proc. International Conference on Systems and Storage (SYSTOR)*, pages 2:1–2:12. ACM, 2012.

[109] Jim Summers, Tim Brecht, Derek Eager, and Bernard Wong. To chunk or not to chunk: Implications for HTTP streaming video server performance. In *Proc. International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 15–20. ACM, 2012.

[110] Tyler Szepesi, Benjamin Cassell, Tim Brecht, Derek Eager, Jim Summers, and Bernard Wong. Using Libception to understand and improve HTTP streaming video server throughput. In *Proc. International Conference on Performance Engineering (ICPE)*, pages 51–62. ACM / SPEC, 2017.

[111] Tyler Szepesi, Bernard Wong, Benjamin Cassell, and Tim Brecht. Designing a low-latency cuckoo hash table for write-intensive workloads using RDMA. In *Proc. International Workshop on Rack-scale Computing (WRSC)*, pages 1–6, 2014.

[112] Josh Tabak. Boom. More transcode servers. https://blog.twitch.tv/boom-more-transcode-servers-427bcbfb519, 2015. Accessed April 12, 2019.

[113] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. Tailwind: Fast and atomic RDMA-based replication. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 851–863. USENIX, 2018.

[114] Arash Tavakkol, Aasheesh Kolli, Stanko Novakovic, Kaveh Razavi, Juan Gomez-Luna, Hasan Hassan, Claude Barthels, Yaohua Wang, Mohammad Sadrosadati, Saugata Ghose, et al. Enabling efficient RDMA-based synchronous mirroring of persistent memory transactions. *arXiv*, pages 1–11, 2018.

[115] Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Clemens Lutz, Martin Schmatz, and Thomas R. Gross. RStore: A direct-access DRAM-based data store. In *Proc. International Conference on Distributed Computing Systems (ICDCS)*, pages 674–685. IEEE, 2015.

[116] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 306–324. ACM, 2017.

[117] Shin-Yeh Tsai and Yiying Zhang. Building atomic, crash-consistent data stores with disaggregated persistent memory. *arXiv*, pages 1–14, 2019.

[118] Hayley Tsukayama. More than 21 million people watched gaming's biggest annual show on Twitch. https://www.washingtonpost.com/news/the-switch/wp/2015/06/29/more-than-21-million-people-watched-gamings-biggest-annual-show-on-twitch/, 2015. Washington Post. Accessed April 12, 2019.

[119] Twitch. Twitch. https://www.twitch.tv/. Accessed April 12, 2019.

[120] Steve VanDeBogart, Christopher Frost, and Eddie Kohler. Reducing seek overhead with application-directed prefetching. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 1–14. USENIX, 2009.

[121] Elizabeth Varki, Allen Hubbe, and Arif Merchant. Improve prefetch performance by splitting the cache replacement queue. In *Proc. International Conference on Advanced Infocomm Technology (ICAIT)*, pages 98–108. IEEE, 2013.

[122] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proc. European Conference on Computer Systems (EuroSys)*, pages 18:1–18:17. ACM, 2015.

[123] Matthew Wachs, Lianghong Xu, Arkady Kanevsky, and Gregory R. Ganger. Exertion-based billing for cloud storage access. In *Proc. Workshop on Hot Topics in Cloud Computing (HotCloud)*, pages 1–5. USENIX, 2011.

[124] Md. Wasi-ur-Rahman, Nusrat Sharmin Islam, Xiaoyi Lu, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K. Panda. High-performance RDMA-based design of Hadoop MapReduce over InfiniBand. In *Proc. International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW)*, pages 1908–1917. IEEE, 2013.

[125] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pages 233–251. USENIX, 2018.

[126] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. Replication-driven live reconfiguration for fast distributed transaction processing. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 335–347. USENIX, 2017.

[127] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 87–104. ACM, 2015.

[128] Wikipedia. ABA problem. https://en.wikipedia.org/wiki/ABA_problem. Accessed April 12, 2019.

[129] Wikipedia. iWARP. https://en.wikipedia.org/wiki/IWARP. Accessed April 12, 2019.

[130] Wikipedia. RDMA over converged ethernet. https://en.wikipedia.org/wiki/RDMA_over_Converged_Ethernet. Accessed April 12, 2019.

[131] Wikipedia. Sequential consistency. https://en.wikipedia.org/wiki/Sequential_consistency. Accessed August 20, 2019.

[132] Xinan Yan, Bernard Wong, and Sharon Choy. R3S: RDMA-based RDD remote storage for Spark. In *Proc. International Workshop on Adaptive and Reflective Middleware (ARM)*, pages 4:1–4:6. ACM / IFIP / USENIX, 2016.

[133] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *Proc. Conference on File and Storage Technologies (FAST)*, pages 221–234. USENIX, 2019.

[134] YouTube. YouTube Live. https://www.youtube.com/live. Accessed April 12, 2019.

[135] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14. USENIX, 2012.

[136] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Very Large Data Bases (VLDB) Endowment*, 10(6):685–696, 2017.

[137] Cong Zhang and Jiangchuan Liu. On crowdsourced interactive live streaming: A Twitch.tv-based measurement study. In *Proc. International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 55–60. ACM, 2015.

Appendices

Appendix A

Nessie

A.1 Full Protocol

A.1.1 Overview

The Nessie protocol is relatively complex, involving a variety of operations on index table entries (ITEs) and data table entries (DTEs). In this appendix, we give a step-by-step breakdown of each Nessie operation. We also discuss the conflicts that can be caused by each type of operation, and how the protocol resolves these conflicts and prevents them from affecting consistency.

A.1.2 Get

Get operations in Nessie involve RDMA read verbs on ITEs and remote DTEs, and DRAM reads on local DTEs. These actions do not modify values, and therefore do not create consistency concerns on their own. Get operations, as with any operations that look up ITEs and their associated DTEs, insert ITE-DTE pairs into a thread-local LRU cache. Caches are not shared, and consistency for any given thread's local cache is maintained by associating entries with expiration timestamps. These caching properties apply to all operations, not just get operations. In summary, get operations on their own cannot create consistency conflicts under the Nessie protocol. A get operation will only encounter conflicts if a concurrent put, migrate, or delete operation interferes with the get operation. At a high level, a get operation performs a forward pass (for cuckoo hashing purposes) over

index table entries looking for a value, and then a reverse pass (to prevent concurrency errors). The detailed steps for a get operation G on a key, K^G, are as follows:

Step 0: A timer is started for G. If at any point G's timer exceeds the system's configured expiration period length, G is aborted, returns a timeout error, and must be retried, as it may no longer be working with consistent values.

Step 1: G computes the ITEs, I^G, to which K^G hashes. In n-way cuckoo hashing this will be the ITEs I_1^G through I_n^G.

Step 2: G iterates over each ITE I_i^G in I^G, reading them in order using RDMA read verbs. Each I_i^G, if it is not empty, refers to a DTE, where I_1^G refers to D_1^G, I_2^G refers to D_2^G and so on. G must check each DTE D_i^G, making up the set D^G, looking for one that contains K^G. This is accomplished through several substeps after each I_i^G is read, explained below.

Step 2.1: If I_i^G is empty, G repeats Step 2 for I_{i+1}^G.

Step 2.2: If I_i^G is not empty, G checks its filter bits, $filter(I_i^G)$. If $filter(I_i^G) \neq filter(K^G)$, then it is guaranteed that D_i^G does not contain K^G, and G repeats Step 2 for I_{i+1}^G.

Step 2.3: If $filter(I_i^G) = filter(K^G)$, then it is possible that D_i^G contains K^G. If D_i^G is local, it is read directly from memory. If D_i^G is remote and a non-expired entry for I_i^G exists in the cache, the cache entry's expiration time is updated to the time I_i^G was retrieved plus an expiration period, and D_i^G is retrieved from the cache. Otherwise, D_i^G is retrieved remotely using an RDMA read verb, and a copy is inserted into the cache, with I_i^G as the key and an expiration time of I_i^G's retrieval time plus an expiration period. If D_i^G does not contain K^G, then G repeats Step 2 for I_{i+1}^G.

Step 2.4: If D_i^G has its valid bit set, it can be returned and the operation is complete.

Step 2.5: If D_i^G is invalid, then a concurrent operation, O, that is modifying D_i^G is in progress. Because O does not mark the DTE it replaces, D_{cnc}^O, for recycling until O completes (and sets D_i^G to valid), D_{cnc}^O is still valid and can be returned by G if it is retrieved within a single expiration period from the time that I_i^G was read. From a consistency perspective, using this value means that G completes before O. O has recorded the index table entry referring to D_{cnc}^O, I_{cnc}^O, inside D_i^G as a previous version index. G extracts I_{cnc}^O from D_i^G and, if it is not empty, uses it to retrieve D_{cnc}^O from local memory, the cache, or remote memory as in Step 2.3 (albeit, the expiration time for any inserted or updated cache entry for D_{cnc}^O would be based on the time that I_i^G was retrieved). If D_{cnc}^O does not contain K^G or is invalid, G aborts and returns a concurrent operation error, and the user

may re-attempt G (if desired). Otherwise, as D^O_{cnc} is valid and contains K^G, its value is returned by G and the operation is complete.

Step 3: By the time G iterates to I^G_n, if it has found no I^G_i containing K^G, it is possible that a concurrent operation, O, moved an entry containing K^G to an ITE in I^G_1 through I^G_{n-1}. G must therefore iterate over I^G_{n-1} through I^G_1, retrieving them with RDMA read verbs. Because all operations, including migrate operations and delete operations, issue new DTEs which are guaranteed to be unique for a full expiration period, any series of operations that changed ITEs in I^G_1 through I^G_{n-1} will be visible simply by re-reading them with an RDMA read verb. If any such changes are found, then G aborts, a concurrent operation error is returned, and the user may re-attempt G (if desired). If no such changes are found, then G returns that K^G is not in the system.

A.1.3 Put

Put operations in the Nessie protocol use RDMA read verbs to access ITEs and remote DTEs, and DRAM reads to access local DTEs. As with get operations, these RDMA read verbs and DRAM reads do not modify any data and therefore do not cause consistency conflicts. Put operations also interact with the local cache, however the local cache is thread-specific and not shared, and therefore this cannot cause consistency conflicts.

Put operations use RDMA compare-and-swap verbs to update ITEs. They furthermore perform RDMA write verbs on remote memory or DRAM writes on local memory when creating new DTEs. Because these actions modify globally accessible state, they can cause consistency conflicts for other operations. We will examine the steps taken by a put operation in order, and discuss how those can conflict with concurrent operations of varying types, and how the Nessie protocol resolves these conflicts. At a high level, a put operation makes a forward pass (for cuckoo hashing) looking for an empty or matching index to write to. After finding an appropriate index table entry, due to cuckoo hashing, the put operation must continue iterating over entries to make sure there are no duplicate entries for the same key. After inspecting all entries, the put operation performs a reverse pass to look for conflicts caused by other operations that would otherwise result in concurrency errors. In detail, a put operation P on a key, K^P, and a value, V^P, proceeds as follows:

Step 0: A timer is started for P. If at any point P's timer exceeds the system's configured expiration period length, P is aborted, returns a timeout error, and must be retried, as it may no longer be working with consistent values.

Step 1: P computes the ITEs, I^P, to which K^P hashes. In n-way cuckoo hashing this will be the ITEs I^P_1 through I^P_n.

Step 2: P iterates over each ITE I_i^P in I^P, reading them in order using RDMA read verbs. Each I_i^P, if it is not empty, refers to a DTE, where I_1^P refers to D_1^P, I_2^P refers to D_2^P and so on. P checks each I_i^P and DTE D_i^P, making up the set D^P, looking for a candidate ITE, I_{cnd}^P, which is empty or refers to a DTE that contains K^P. This is accomplished through several substeps after each I_i^P is read, explained below.

Step 2.1: If I_i^P is empty, it is immediately be used as I_{cnd}^P.

Step 2.2: If I_i^P is not empty, P checks its filter bits, $filter(I_i^P)$. If $filter(I_i^P) \neq filter(K^P)$, then it is guaranteed that D_i^P does not contain K^P, and P repeats Step 2 for I_{i+1}^P.

Step 2.3: If $filter(I_i^P) = filter(K^P)$, then it is possible that D_i^P contains K^P. If D_i^P is local, its key is retrieved using a DRAM read. Otherwise, the cache is checked for I_i^P, and if a non-expired entry exists, the key from D_i^P is extracted from the cache. If no cache entry is found, D_i^P without its value (which is not needed for P) is retrieved using an RDMA read verb. If D_i^P contains K^P but is invalid, then P aborts and a concurrent operation error is returned (the user may re-attempt P if desired). If D_i^P is valid, then I_i^P is used as I_{cnd}^P. I_i^P is also used as I_{cnd}^P if D_i^P is invalid but was previously inserted by the client performing P (this can occur if a previous attempt of P, or a migrate operation incurred by a previous attempt of P, timed out or was otherwise aborted). If D_i^P does not contain K^P, then P repeats Step 2 for I_{i+1}^P.

Step 2.4: If I_{cnd}^P is not found, it means that all ITEs in I^P are occupied, and all DTEs in D^P contain keys which are not K^P. A migrate operation, M, must be performed, and P must be re-attempted. Migrate operations are discussed in Appendix A.1.4.

Step 3: P allocates a DTE, D_{new}^P, containing K^P, V^P, and I_{cnd}^P (as a previous version index). D_{new}^P is written to local memory using a DRAM write. This could equivalently be done on a remote node's memory using an RDMA write verb with minor changes to the system to support this behaviour, and no changes to consistency. When D_{new}^P is created, its valid bit is set to false. I_{cnd}^P is not updated to refer to D_{new}^P until D_{new}^P is fully written. This means that the allocation of D_{new}^P does not cause consistency conflicts, as it is invisible to other operations in the system. Operations which complete while D_{new}^P exists but is not referred to by I_{cnd}^P can be considered to have occurred before P.

Step 3.1: Up until this point, P has performed no actions which cause consistency conflicts for concurrent operations. Now, however, it must update I_{cnd}^P to refer to D_{new}^P. This is done atomically with an RDMA compare-and-swap verb, C_{cnd}^P, preventing consistency issues that could arise from simultaneous RDMA verbs operating on I_{cnd}^P. If C_{cnd}^P fails, D_{new}^P

can immediately be marked for recycling as it has never been visible to the system, and P aborts returning a concurrent operation error (the user may re-attempt P if desired). If C^P_{cnd} succeeds, I^P_{cnd} is overwritten and is now a non-empty ITE, I^P_{new}, which contains $filter(K^P)$ and the location of D^P_{new}. Consistency concerns raised by C^P_{cnd} are as follows:

Step 3.1 Get Operation Conflict: Consider the effects of C^P_{cnd} on a concurrent get operation, G. If C^P_{cnd} occurs before G reads I^P_{new} during its Step 2, G will see that D^P_{new} is invalid, abort, and return a concurrent operation error. If C^P_{cnd} (or any number of later operations) occurs on I^P_{cnd} after G has read I^P_{cnd}, then one of two cases occurs. If I^P_{cnd} was the last ITE for G ($cnd = n^G$) then G completes, and is considered to have occurred before P. Otherwise, G sees that I^P_{cnd} has changed to I^P_{new} during its Step 3, aborts, and returns a concurrent operation error.

Step 3.1 Put Operation Conflicts: C^P_{cnd} can conflict with another put operation, P_2. First consider the case where both P and P_2 choose the same candidate, in other words P_2 chooses a candidate $I^{P_2}_{cnd} = I^P_{cnd}$ and attempts an RDMA compare-and-swap verb, $C^{P_2}_{cnd}$, on it. The atomicity of RDMA compare-and-swap verbs prevents C^P_{cnd} and $C^{P_2}_{cnd}$ from both completing successfully. If either RDMA compare-and-swap verb completes before the other put operation reads I^P_{cnd} in Step 2, then the slower of P and P_2 will see an invalid data table entry, abort, and return a concurrent operation error. If either RDMA compare-and-swap verb completes after the other put operation reads I^P_{cnd} in Step 2, then the slower of P and P_2 will fail its RDMA compare-and-swap verb, abort, and return a concurrent operation error. Next, consider the case where both P and P_2 choose, and successfully modify different candidates ($I^P_{cnd} \neq I^{P_2}_{cnd}$) using RDMA compare-and-swap operations. Despite having successfully modified their candidates, only one of P and P_2 will complete, while the other will eventually abort and return an error. This occurs because the put operation that chooses candidate I_i such that $i = min(cnd^P, cnd^{P_2})$ will clear the other put operation's candidate in Step 4. Conversely, the put operation that chooses candidate I_j such that $j = max(cnd^P, cnd^{P_2})$ will, during Step 5, detect that its candidate has been overwritten, abort, and return a concurrent operation error. This generalizes across any number of concurrent put operations: whichever put operation chooses the candidate with the smallest index will clear the candidates for the other put operations, causing them to abort and return concurrent operation errors.

Step 3.1 Migrate Operation Conflicts: C^P_{cnd} can conflict with a concurrent migrate operation, M, initiated by another put operation, P_2. First, consider when M uses I^P_{cnd} as its destination. If C^P_{cnd} occurs before M checks if I^P_{cnd} is empty, then M will see an invalid entry and return a concurrent operation error having made no changes to the system. If C^P_{cnd} occurs immediately after M checks I^P_{cnd} then there are two possibilities: the first is

that I^P_{cnd} was initially empty, M attempts to RDMA compare-and-swap it, fails, aborts, and returns a concurrent operation error having made no changes to the system. If I^P_{cnd} was not empty, M will attempt a recursive migrate operation using I^P_{cnd} as its source. This brings us to considering cases in which M uses I^P_{cnd} as its source. If C^P_{cnd} occurs before M reads I^P_{cnd}, M sees an invalid DTE, aborts, and returns a concurrent operation error. Otherwise, if P reads I^P_{cnd} and then M clears I^P_{cnd} before C^P_{cnd} occurs, then C^P_{cnd} fails and P aborts and returns a concurrent operation error, allowing M to proceed normally. Finally, there is the case where C^P_{cnd} changes I^P_{cnd} after it has been read by M, but before M clears it, causing M to fail its RDMA compare-and-swap. Before M returns a concurrent operation error and P_2 is retried, M attempts to clear its leftover destination copy, I^M_{dst}, using an RDMA compare-and-swap verb, C^M_{dst}. If C^M_{dst} succeeds, M adds I^M_{dst} to its list of items to recycle, I^M_{rec}. If C^M_{dst} fails, it implies that I^M_{dst} has already been cleared by another operation. In this case, there is no additional work for M, which returns.

Step 3.2: If I^P_{cnd} was not empty, the DTE originally referred to by I^P_{cnd}, D^P_{cnd}, must be recycled. This does not occur immediately, as D^P_{cnd} is used for the previous version optimization. Therefore, I^P_{cnd} is added to a list of ITEs, I^P_{rec}, whose DTEs must be marked for recycling once P has successfully completed. If at any point P must be re-attempted, I^P_{rec} is maintained. Thus, no matter how many attempts P requires to complete, all of I^P_{rec} will eventually be recycled, and P will cause no memory leakage.

Step 4: P must now clear any unexamined ITEs, I^P_{clr}, which contain K^P (if they exist). To do so, P iterates over each ITE I^P_i in I^P_{cnd+1} to I^P_n, reading them in order using RDMA read verbs.

Step 4.1: If I^P_i is empty, P repeats Step 4 for I^P_{i+1}.

Step 4.2: If I^P_i is not empty, P checks its filter bits, $filter(I^P_i)$. If $filter(I^P_i) \neq filter(K^P)$, then it is guaranteed that D^P_i does not contain K^P, and P repeats Step 4 for I^P_{i+1}.

Step 4.3: If $filter(I^P_i) = filter(K^P)$, then it is possible that D^P_i contains K^P. If D^P_i is local, then its key is read directly from DRAM. If D^P_i is remote, the cache is checked for I^P_i, and if a non-expired entry exists, the key from D^P_i is extracted from the cache. Otherwise, D^P_i without its value (which is not needed for P) is retrieved remotely using an RDMA read verb. If D^P_i does not contain K^P, then P repeats Step 4 for I^P_{i+1}.

Step 4.4: If D^P_i contains K^P, then I^P_i is recorded as I^P_{clr}, and I^P_i is set to be empty using an RDMA compare-and-swap verb, C^P_{clr}. If C^P_{clr} fails, it is indicative that another conflicting operation is underway, so P aborts and returns a concurrent operation error

(after which the user re-attempts P). If C^P_{clr} is successful, I^P_{clr} is added to I^P_{rec}. Consistency issues for C^P_{clr} are as follows:

Step 4.4 Get Operation Conflicts: Consider the impact of C^P_{clr} on a concurrent get operation, G. If I^P_{clr} contains a key that G is not interested in, there is no impact on G's consistency regardless of the order in which operations occur. Otherwise, if G is attempting to find K^P, it means that G could not have encountered I^P_{new}, as that would have caused G to return. G therefore must have encountered a different value (for example if I^P_{cnd} was empty before C^P_{cnd}). G will detect that I^P_{cnd} changed during its Step 3, abort, and return a concurrent operation error.

Step 4.4 Put Operation Conflicts: C^P_{clr} can affect a concurrent put operation, P_2. If P_2 is for any key that is not K^P, conflicts between P and P_2 have low impact: it is possible that P will clear an ITE that has already been inspected by P_2, causing P_2 to abort and return a concurrent operation error during its Step 5. However, C^P_{clr} will never modify an ITE containing a value directly relevant to P_2, as C^P_{clr} only occurs on ITEs that reference DTEs containing K^P. In the case that P_2 is also for key K^P, C^P_{clr} can possibly affect ITEs in I^P_{cnd+1} through I^P_n. If C^P_{clr} clears an ITE immediately before P_2 attempts to clear it with an RDMA compare-and-swap verb, P_2 will abort, return a concurrent operation error, and be re-attempted. If C^P_{clr} clears P_2's candidate ITE, P_2 will detect this during its Step 5, abort, return a concurrent operation error, and be re-attempted. P_2 may also see, during Step 5, that P has inserted I^P_{new}, which will cause P_2 to abort, return a concurrent operation error, and be re-attempted. In all scenarios, consistency is maintained.

Step 4.4 Migrate Operation Conflicts: C^P_{clr} can conflict with a concurrent migrate operation, M, initiated by another put operation, P_2. First, consider when M uses I^P_{clr} as its destination. If C^P_{clr} occurs before M checks if I^P_{clr} is empty, M will proceed normally. If P reads I^P_{clr}, and then M immediately modifies I^P_{clr}, C^P_{clr} will fail, allowing P to abort, return a concurrent operation error, and be re-attempted. If C^P_{clr} occurs after M has read I^P_{clr} but before M attempts an RDMA compare-and-swap verb on I^P_{clr}, then M's RDMA compare-and-swap verb will fail allowing it to abort, return a concurrent operation error, and retry P_2. Next, consider when M uses I^P_{clr} as its source. If C^P_{clr} occurs before M reads I^P_{clr}, then M sees an empty entry, allowing it to immediately return and retry P_2 (as M is no longer necessary). If P reads I^P_{clr} and then M clears I^P_{clr} before C^P_{clr} occurs, then C^P_{clr} will fail, P aborts, returns a concurrent operation error, and is retried. The last case that can occur when using I^P_{clr} as M's source is when C^P_{clr} modifies I^P_{clr} after it has been read by M, but before M marks I^P_{clr} as empty. C^P_{clr} causes M to fail its RDMA compare-and-swap verb. Before M aborts and returns a concurrent operation error, M attempts to clear its leftover destination copy, I^M_{dst}, using an RDMA compare-and-swap verb, C^M_{dst}. If C^M_{dst}

succeeds, M adds I^M_{dst} to its list of items to recycle, I^M_{rec}. If C^M_{dst} fails, it implies that I^M_{dst} has already been cleared by another operation. In this case, there is no additional work for M, which returns. Regardless of the outcome, P_2 is retried.

Step 5: As with get operations, by the time P inspects I^P_n, it is possible that previous entries were changed by concurrent operations. P iterates over I^P_{n-1} through I^P_1, retrieving them with RDMA reads. If any differences are detected from Step 2, Step 3, or Step 4, then P aborts, a concurrent operation error is returned, and P must be re-attempted.

Step 6: If no changes have been detected, then P is successful. P must now change D^P_{new}'s valid bit to true, using a DRAM write if it is local, or an RDMA write verb if it is remote. Despite this write occurring to a DTE which could be concurrently read by other operations, this does not raise any consistency issues. The valid bit, being a single bit, can only be seen by other threads as set or unset, and not an in-between value. At worst, a concurrent operation, O, will see the valid bit while it is still false, O will abort and return a concurrent operation error, and then O will be retried.

Step 7: Now that P is successful, all DTEs referred to by ITEs in I^P_{rec} must be marked for recycling. This may be done before returning from P, or alternatively, to reduce the latency of P, may be done later in the background. Regardless of when it is done, Nessie sets each DTE referred to by I^P_{rec} for recycling, using DRAM writes for local DTEs and RDMA write verbs for remote DTEs. In doing so, the expiration times are set to the current time plus an expiration period, and the recycle bit is flipped to true in each DTE. The recycle bit in a DTE is placed in memory immediately after the timestamp. Because the recycle information is updated in byte order, the recycle bit is not flipped until after the timestamp has been fully written. Threads that attempt to inspect the timestamp only do so if the recycle bit is set to true. This prevents threads in the system from reading semi-written timestamps. Expiration times prevent this step from having any impact on system consistency.

A.1.4 Migrate

In Nessie, a migrate operation, M, on a key, K^M, is run when a put operation, P, on K^M is unable to find an appropriate location to perform an insertion or update. At a high level, the migrate operation is attempting to empty a source index table entry by moving its contents to one of its alternate cuckoo hash locations, the destination index table entry. This is repeated recursively if necessary, up to a system-configurable depth. The details of a migrate operation are as follows:

Step 0: A timer is started for M. If at any point M's timer exceeds the system's configured expiration period length, M is aborted, returns a timeout error, and must be retried, as it may no longer be working with consistent values.

Step 1: M computes the ITEs, I^M, to which K^M hashes. In n-way cuckoo hashing this will be the ITEs I_1^M through I_n^M.

Step 2: M selects a source ITE, I_{src}^M, from I^M. In our current implementation, Nessie chooses $I_{src}^M = I_1^M$, although other selections are equally valid. I_{src}^M is read using an RDMA read verb.

Step 2.1: If I_{src}^M is empty, then it has been freed by a concurrent operation between P and M, and M returns successfully.

Step 2.2: If I_{src}^M is not empty, M checks its filter bits, $filter(I_{src}^M)$. If $filter(I_{src}^M) \neq filter(K^M)$, then it is guaranteed that the DTE, D_{src}^M, to which I_{src}^M refers does not contain K^M, and M continues by skipping to Step 3.

Step 2.3: If $filter(I_{src}^M) = filter(K^M)$, then it is possible that D_{src}^M contains K^M. If D_{src}^M is local, it is retrieved directly with a read from DRAM. If D_{src}^M is remote, the local cache is checked for I_{src}^M, and if a non-expired entry exists, D_{src}^M is extracted from the cache. Otherwise, D_{src}^M is retrieved remotely using an RDMA read verb. If D_{src}^M contains K^M due to a concurrent operation between P and M, then M returns successfully.

Step 2.4: If D_{src}^M is invalid, M aborts (as it would conflict with an in-progress operation) and returns a concurrent operation error.

Step 3: D_{src}^M contains a key, K^{src} which is not K^M. M computes the ITEs, I^{src}, to which K^{src} hashes. In n-way cuckoo hashing this will be the ITEs I_1^{src} through I_n^{src}.

Step 4: M selects a destination ITE, I_{dst}^M, from I^{src}. In our current implementation, Nessie chooses the first $I_{dst}^M = I_i^{src}$ such that $I_i^{src} \neq I_{src}^M$, and I_i^{src} is not in I^M. I_{dst}^M is read using an RDMA read verb.

Step 4.1: If I_{dst}^M is empty, then it is used immediately.

Step 4.2: If I_{dst}^M is not empty, then a recursive migrate operation, R, must be performed, using $I_{src}^M = I_{dst}^M$. If R fails, it returns an error, and M must be re-attempted. Otherwise, R's success means that I_{dst}^M is now empty.

Step 5: The source is now ready to be swapped to the destination. M allocates a copy of D_{src}^M, D_{dst}^M, with a false valid bit. In the description of the protocol provided in this thesis,

D^M_{dst} is written locally using a DRAM write, but it could equally be placed on a remote node using an RDMA write verb with minor changes to support this behaviour and no changes to consistency.

Step 6: I^M_{dst}, which is currently recorded as being empty, must be updated to point to D^M_{dst} using an RDMA compare-and-swap verb, C^M_{dst}. If C^M_{dst} fails then, as M still has not made any visible changes to the system, D^M_{dst} is immediately marked for recycling, a concurrent operation error is returned, and M is retried. This portion of a migrate operation is, in terms of consistency, largely identical to Step 3.1 of a put operation that chooses an empty entry for its candidate. In addition to those consistency concerns, listed in Appendix A.1.3, consistency concerns for this step are:

Step 6 Get Operation Conflicts: The success of C^M_{dst} creates an invalid, duplicate entry in the system for K^{src}. For a concurrent get operation, G, if $src^M < dst^M$, then G will read I^M_{src} without reading I^M_{dst} and return normally. From a consistency perspective G is considered to have completed before M. It will never be the case that $src^M = dst^M$, due to the restrictions placed on choosing I^M_{dst} in Step 4. If $dst^M < src^M$, then G's outcome depends on when C^M_{dst} occurs. If C^M_{dst} occurs before G has read I^M_{dst} during its Step 2, then G sees an invalid entry, aborts, and returns a concurrent operation error. If C^M_{dst} occurs after G has read I^M_{dst} during its Step 2, then if C^M_{src} has yet to occur, G reads I^M_{src} and returns successfully. If C^M_{src}, discussed in Step 7, occurs before G reads I^M_{src}, then G detects a change to I^M_{dst} during its Step 3, aborts, and returns a concurrent operation error.

Step 6 Put Operation Conflicts: It is possible for another concurrent put operation, P_2, to use I^M_{dst} as a candidate, $I^{P_2}_{cnd}$. If it does, whichever operation finishes its RDMA compare-and-swap verb first will cause the other operation to abort and return a concurrent operation error, as the slower operation will either read an invalid DTE or fail its RDMA compare-and-swap verb. Alternatively, if P_2 is using a different ITE as a candidate, but I^M_{dst} still belongs to the list of ITEs it inspects (in other words, $dst^M \neq cnd^{P_2}$), there are two possible outcomes. If $dst^M < cnd^{P_2}$, P_2 will have already read the value of I^M_{dst}, and will detect a change, abort, and return a concurrent operation error during its Step 5. If $cnd^{P_2} < dst^M$, then P_2 will appropriately clear $I^{P_2}_{cnd}$ during its Step 4 (if D^M_{dst} contains the same key that P_2 is operating on, otherwise P_2 will simply ignore it).

Step 6 Migrate Operation Conflicts: C^M_{dst} can cause conflicts with a concurrent migrate operation, M_2. If M_2 chooses I^M_{dst} as its destination, then whichever migrate operation completes its RDMA compare-and-swap verb on I^M_{dst} first will cause the other migrate operation to either see an invalid entry or fail its RDMA compare-and-swap verb. In either case, the slower operation aborts and returns a concurrent operation error. C^M_{dst} cannot interfere directly with M_2 if it has chosen I^M_{dst} as its source. The reason for this is

that C^M_{dst} is only attempted by M on an entry it believes to be empty. M_2 will only choose a source it believes to be non-empty. Therefore, if C^M_{dst} conflicts with M_2 when M_2 uses I^M_{dst} as a source, then it implies that another concurrent operation has occurred which is the actual source of a conflict. Regardless, if M_2's view of I^M_{dst} is more recent, when C^M_{dst} occurs it will fail and M will abort and return a concurrent operation error. If M's view of I^M_{dst} is more recent, then M_2 will fail its RDMA compare-and-swap verb during its Step 7, abort, and return a concurrent operation error.

Step 7: I^M_{src} is now marked as empty using an RDMA compare-and-swap verb, C^M_{src}. If C^M_{src} succeeds, then similar to a put operation, I^M_{src} is added to a list of ITEs, I^M_{rec}, whose DTEs must be marked for recycling when the operation is finished. Consistency concerns for C^M_{src} are similar to those in Step 4.4 of a put operation, discussed in Appendix A.1.3. Consistency concerns unique to C^M_{src} are:

Step 7 Get Operation Conflicts: C^M_{src} can have an impact on a concurrent get operation G. If G reads I^M_{src} before C^M_{src} occurs, then G returns normally. However, if C^M_{src} completes first, then G will not find a value in I^M_{src}. If $dst^M > src^M$, then G will subsequently read I^M_{dst} and return an appropriate value. If $src^M > dst^M$, then G must have recorded a value for I^M_{dst} before M performed C^M_{dst} (as otherwise G would have already returned). During G's Step 3, it will detect a change to I^M_{dst} and return a concurrent operation error. The use of timeouts on operations and expiration times on recycled DTEs guarantees that this change will be detected regardless of whether or not additional operations occur on I^M_{dst}. This prevents the case where, for example, another subsequent migrate operation erroneously causes G to miss an existing entry for the value it is looking for.

Step 7 Put Operation Conflicts: I^M_{src} must be non-empty for C^M_{src} to occur, but it is still possible for another concurrent put operation, P_2, to use I^M_{src} as a candidate, or even to clear it during P_2's Step 4. First, consider when P_2 chooses I^M_{src} as a candidate. If C^M_{src} occurs before P_2 reads I^M_{src}, then P_2 will continue normally. If C^M_{src} occurs immediately after P reads I^M_{src}, then P_2 will fail its RDMA compare-and-swap verb, abort, and return a concurrent operation error. If P_2 performs an RDMA compare-and-swap verb on I^M_{src} before C^M_{src} occurs, then C^M_{src} will fail. In this case, M immediately attempts to clear I^M_{dst}, with an RDMA compare-and-swap verb, C^M_{clr}. If C^M_{clr} succeeds, M adds I^M_{dst} to I^M_{rec}, aborts, and returns a concurrent operation error. Otherwise, it implies that another operation has cleared I^M_{dst}, and M aborts and returns a concurrent operation error without needing to recycle anything. Regardless of the outcome of C^M_{clr}, P is reattempted. Next, consider when P_2 attempts to clear I^M_{src} during its Step 4. If C^M_{src} happens before P_2 reads I^M_{src}, then P_2 continues normally. If C^M_{src} happens immediately after P_2 reads I^M_{src} but before its RDMA compare-and-swap verb, then P_2's RDMA compare-and-swap verb fails and it

aborts and returns a concurrent operation error. Finally, if P_2 clears I^M_{src}, then C^M_{src} will fail. M therefore attempts to clear I^M_{dst} with an RDMA compare-and-swap verb. If this final compare-and-swap fails, then another operation has already replaced I^M_{dst}, and M may abort and return a concurrent operation error without needing to recycle I^M_{dst}. Otherwise, I^M_{dst} is added to I^M_{rec}, then M aborts and returns a concurrent operation error.

Step 7 Migrate Operation Conflicts: M can interfere with another migrate operation, M_2, that has been initiated by a put operation, P_2. First, consider the case where M_2 has chosen I^M_{src} as its destination. Under these circumstances, M_2 must believe I^M_{src} is empty to operate on it. Conversely, M believes I^M_{src} to be non-empty. This can result from another concurrent operation interfering with both M and M_2. If M's view of I^M_{src} is more recent, regardless of whether or not it completes C^M_{src}, M_2 will fail its RDMA compare-and-swap verb, abort, and return a concurrent operation error. Likewise, if M_2's view of I^M_{src} is more recent, then C^M_{src} will fail, and M aborts and returns a concurrent operation error. If M fails, it attempts to clear and recycle I^M_{dst} before returning. Next, consider the case where M_2 has chosen I^M_{src} as its destination. In this case, the migrate operation that finishes its RDMA compare-and-swap verb first will cause the other migrate operation to return a concurrent operation error, either when it inspects I^M_{src} and sees an invalid entry, or when its RDMA compare-and-swap verb fails. The failing migrate operation attempts to clear and recycle its destination entry before aborting and returning a concurrent operation error. The failing migrate operation would then be re-attempted.

Step 8: The migrate operation has now successfully completed. To denote that it is finished, M changes D^M_{dst}'s valid bit to true using a local DRAM write or an RDMA write verb, depending on its location.

Step 9: As with a put operation, the final step of a migrate operation is to mark any entries remaining in I^M_{rec} for recycling. This can be done at the end of the migrate operation, or can be delayed until an advantageous time, such as when the system is idle, to minimize the impact on latency.

A.1.5 Delete

A delete operation in Nessie is nearly identical to a put operation, except it inserts a temporary blank value and then clears it from the system. A delete operation, D, on a key, K^D, performs Step 0 through Step 7 of a put operation P with the following minor differences:

Step 3: Instead of allocating a DTE, D^P_{new}, containing a specified value, the delete operation inserts D^P_{new} with an arbitrary (blank) value. D^P_{new} will never be marked as valid and therefore its value will never be returned by any operations. However, inserting D^P_{new} is still necessary to preserve consistency. By pointing an ITE to D^P_{new}, conflicting concurrent operations will be able to detect that operation D is in progress, in the same way that they would detect that operation P is in progress.

Step 6: Instead of marking D^P_{new} as valid, the delete operation performs a final RDMA compare-and-swap verb to mark I^P_{new} as empty. This operation has consistency concerns identical to Step 4.4 of a put operation. If the RDMA compare-and-swap verb succeeds, then I^P_{new} is added to I^P_{rec}, and the DTE it references will be reclaimed in Step 7. If the RDMA compare-and-swap verb fails, then D aborts, returns a concurrent operation error, and D is retried.