DISS. ETH No. 23235

End-to-End Considerations in Unification of High-Performance IO

A thesis submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
Animesh Trivedi
Ing. dipl., ETH Zurich
born on August 02, 1986
citizen of the Republic of India

accepted on the recommendation of
Prof. Dr. Thomas R. Gross, examiner
Dr. Bernard Metzler, co-examiner
Prof. Dr. Torsten Hoefler, co-examiner
Prof. Dr. Edouard Bugnion, co-examiner

2016

To
Asha-Lata Mishra, Dr. Arti Trivedi, Anindita Trivedi,
Late R.K. Tiwari, K.K. Mishra, Akhilesh Trivedi, and Asish Trivedi.

Abstract

The performance of modern distributed storage and computing frameworks considerably depends on the IO performance of the many storage and network devices involved. Fortunately, these IO devices have undergone a rapid transformation in the past decade and are now capable of delivering multi-Gigabits/sec bandwidths and ultra-low IO latencies. However, in contrast to IO devices, the performance improvements of single CPU have stalled in the same time period. Hence, the traditional notion of a single fast CPU connected to multiple slow devices no longer holds. Yet, IO stacks are still designed to optimize the CPU time by executing multiple services and routines on a *fast* CPU while a *slow* IO operation is in progress. This situation has led to a *CPU-IO* performance gap, where the CPU's inability to keep up with the execution of thick software stacks and OS routines during a fast IO operation on high-performance network and storage devices limits the performance delivered to data-crunching applications. Multiple research efforts from industry as well as academia have been launched to improve this situation by reducing the hardware and software overheads by providing better IO interfaces, efficiently managing IO resources, and leveraging manycore CPUs for IO processing. However, these efforts exclusively either target the network or the storage stack but not the combination of both.

In this thesis, we address this performance gap and advocate to take a holistic approach towards managing resources, data flows, and devices (network or storage) to form end-to-end data flows in a distributed setting. We first quantify the software and OS overhead in IO operations and and argue to reduce it by building upon the high-performance networking principle. The general philosophy of the principle is to recognize and separate the slow control setup from the fast data access path, and involve CPU/OS in the former only selectively in managerial roles. In the thesis framework, we extend the separation philosophy from networks to storage devices by identifying common themes in the evolution of their software stacks. After identifying common high-performance IO properties, we make a case to unify the network and storage stacks. We then design and build a proof of concept FlashNet, a unified software IO stack that uses high-performance networking abstractions and semantics to access remote storage. In accordance with the original separation philosophy, FlashNet allows the allocation and translation of both

network *and* storage resources prior to a remote storage access.

We then expand the philosophy and demonstrate how to separate the resource setup from fast data accesses in a distributed environment where resources might be spread across multiple machines. We introduce RStore, a distributed in-memory data store that achieves distributed separation using its unique memory-like API and storage abstractions. We quantify the effectiveness of this approach by developing two distributed data-processing applications on top of RStore, namely a distributed key-value sorter (RSort) and a distributed graph-processing engine (Carafe). They both perform well; RSort, for example, outperforms Hadoop TeraSort by a margin of $8-10\times$ on our 12-machine cluster. At the end of the thesis, we document our experience and give recommendations for system builders on how to leverage the separation principle to cut through the thick IO abstractions and layers to design and implement high-performance distributed data processing applications.

Zusammenfassung

Die Leistungsfähigkeit moderner verteilter hängt in hohem Masse von der IO-Leistung der eingesetzten Speichergeräte und des Netzwerkequipments ab. Diese IO-Geräte haben im letzten Jahrzehnt eine schnelle Weiterentwicklung erfahren; sie bieten heute eine Zugriffsbandbreite von mehreren Gigabits pro Sekunde bei sehr geringer Latenz. Im Gegensatz zu den IO-Geräten hat sich die Leistungsfähigkeit einer CPU im genannten Zeitraum kaum erhöht. Damit ist die bisherige Annahme einer Rechnerarchitektur mit einer einzelnen schnellen CPU, die mit mehreren langsamen IO-Geräten interagiert, nicht mehr gültig. Allerdings sind IO-Stacks immer noch unter der Annahme gestaltet, die CPU-Zeit optimal auszunutzen, indem eine schnelle CPU mehrere Dienste und Routinen abarbeitet, während eine langsame IO-Operation zeitgleich abläuft. Diese Situation hat zu einer Disparität zwischen CPU- und IO-Performance geführt, in der die CPU nicht in der Lage ist, komplexe Softwarehierarchien und Betriebssystem-Routinen während einer schnellen Netzwerk- oder Speicher-IO-Operation auszuführen. Dies limitiert die für Applikationen zur Verfügung stehende Leistungsfähigkeit. Es wurden eine Reihe von Forschungsprojekten in industriellem und universitärem Umfeld gestartet, die diese Situation verbessern wollen, indem die Hardware- und Software-Overheads durch bessere IO-Interfaces, effizientes Management der IO-Ressourcen und die Nutzung von Vielkern-Prozessoren reduziert werden sollen. Diese Aktivitäten beschränken sich jedoch auf entweder den Netzwerk- oder den Speicherzugriff und nicht auf eine Kombination beider Dienste.

Die vorliegende Arbeit adressiert diese Disparität und plädiert für einen holistischen Ansatz, in welchem Ressourcen, Datenströme und Geräte (sowohl Netzwerk- als auch Speichergeräte) so verwaltet werden, dass sie in einem verteilten System zu einem Ende-zu-Ende-Datenfluss formiert werden können. Einleitend wird der Software- und Betriebssystem-Overhead für IO-Operationen quantifiziert und daraus der Vorschlag abgeleitet, diesen mit Designprinzipien von Hochgeschwindigkeitsnetzen zu reduzieren. Das Grundprinzip dieses Ansatzes ist es, zeitunkritische Kontrolloperationen vom zeitkritischen Datenzugriff zu separieren und dabei CPU und Betriebssystem nur bei Kontrolloperationen zu involvieren. Im Rahmen der vorliegenden Arbeit wird dieses ursprünglich für die Einbindung von Netzwerkgeräten eingeführte Separationsprinzip auf Speichergeräte erweitert, indem gemeinsame Problemfelder bei der Evolution der zugehörigen Software-Stacks identifiziert werden. Aus der Identifikation gemeinsamer IO-Eigenschaften beider Stacks wird der Vorschlag abgeleitet, diese in einer Implementierung zu vereinen. Dieses Konzept wird anhand der FlashNet-Implementierung, die Abstraktionen und Semantiken von Hochgeschwindigkeitsnetzen für den Zugriff auf entfernten Massenspeicher nutzt, exemplarisch umgesetzt. FlashNet setzt die Philosophie der Trennung von Kontroll- und Datentransferoperationen um und erlaubt damit eine Reservierung und Bereitstellung von sowohl Netzwerk- als auch entfernten Massenspeicherressourcen vor dem effizienten Zugriff.

Im Weiteren wird die Designphilosophie der Trennung von Daten- und Kontrolloperationen verallgemeinert und auf ein verteiltes System angewandt, in dem die Ressourcen über mehrere Endpunkte verteilt sind. Mit RStore wird ein verteilter, hauptspeicher-residenter Datenspeicher eingeführt, der dieses Prinzip durch den Einsatz eines neuen Applikationsprogramm-Interfaces und von Speicherabstraktionen realisiert. Die Effektivität dieses Ansatzes wird anhand von zwei neu entwickelten RStore-Applikationen quantifiziert: RSort, ein verteilter Key-Value Sortieralgorithmus und Carafe, eine verteilte Anwendung zum Prozessieren von Graphen. Beide Anwendungen erreichen ausgezeichnete Performance - zum Beispiel übertrifft RSort das bekannte Hadoop TeraSort um den Faktor acht bis zehn auf dem verwendeten 12-Knoten-Cluster. Die vorliegende Arbeit schliesst mit einem Resümee der gefundenen Erkenntnisse und mit Empfehlungen für Systemarchitekten, wie mittels des Separationsprinzips aufwendige IO-Abstraktionen und Schichten vermieden werden können, um hoch leistungsfähige verteilte Applikationen zur Datenverarbeitung umsetzen zu können.

Acknowledgments

As I sit here in K124 at IBM Research Lab on a relatively warm and sunny November afternoon, I reflect on my experiences as a PhD student over the last 4-5 years. During this time, many remarkable individuals have provided invaluable support to me during my PhD studies and beyond, and if possible, I would like to thank them all. The list is long, and due to the nature of the work, it cannot be comprehensive. I apologize in advance for any omissions on my side while stating that any such omission cannot take any credit away from the provider.

First of all, I am very grateful to thank my advisor, Thomas R. Gross, for giving me the opportunity to carry out this thesis work with him. He provided the necessary freedom and guidance without which the work presented here would not have been possible. Apart from being a tremendous supervisor, he has also been a wonderful mentor for me on many on and off topics from the core thesis work. I thank him for all his valuable support throughout these years.

I would like to thank my co-examiners Bernard Metzler, Torsten Hoefler, and Edouard Bugnion for serving on my thesis committee. They read through numerous drafts of this thesis and gave much appreciated constructive feedback. Their comments were critical in increasing the scope and accuracy of the work presented in this thesis. Thank you all very much.

At the IBM Research Lab, many people have influenced me and left a lasting impression. Bernard Metzler and Patrick Stuedi supervised and guided me routinely over past years. From the day one, Bernard's attitude towards research, work, and life has been very inspirational. During my forming years, he supported my work, invited me to discuss ideas with him, read and critiqued on countless numbers of PhD proposals, and above all, included me in the group as a colleague rather than a student. He is the original RDMA wizard and no part of this thesis work would have been possible without his encyclopedic knowledge about RDMA and his work on SoftiWARP. Patrick's comprehensive approach towards research and his ability to connect dots and gather supporting arguments are truly exceptional. As a researcher he has demonstrated multiple times how to identify relevant problems, device solutions, and most importantly, write and present ideas clearly. Many papers and reports (which constitute this thesis)

that I have written during my PhD work would not have been possible without his demands for "linearization" of arguments and clarity. He often finds graphical and mathematical ways (one of his favorite templates for my mistakes is: $a \times (b + c) = a \times b + a \times c$) to explain why a chain of arguments does not work in a paper draft. I thank you both very much for getting me started, then continuously supporting, and educating me throughout these years.

I have been fortunate enough to work closely with Nikolas Ioannou on our RDMA and flash integration effort. Nikolas is a truly exceptional person with a large instruction cache and a very low context switch time. He can juggle an emacs, a screen, a debugger, and a chat window faster than anyone can follow. I learned many good kernel software development intricacies from him. Of late, Jonas Pfefferle has become an indispensable part of our group. With his impressive low-level systems skills with devices, IO buses, memories, and CPUs, it has been a joy to speculate, debate, and debug code with him. Thank you both very much.

Beyond these core collaborators, I would like to acknowledge a bigger group of people at IBM Research and beyond that have helped me in various capacities. My manager, Thomas Weigold, and department head Evangelos S. Eleftheriou supported me during the last phase of the thesis. Martin Schmatz approved a critical and heft infrastructure upgrade without a blink of an eye, and provided unconditional support for developing my nascent ideas. Clemens Lutz developed Carafe, one of the applications of RStore, that demonstrated the viability of the approach taken in this thesis. Joerg-Eric Sagmeister helped us with the assembly of our server rack while simultaneously educating us with practical tips and techniques about "How to Assemble your own Data Center Rack in 10 Easy Steps". Anne-Marie Cromack and Charlotte Bolliger proofread many paper drafts and parts of this thesis. Since becoming my office mate, Moshe Rappoport supplied the right perspective by inquiring many times about the relevance of my (and in general computing) research and how we were contributing to society and mankind. Sören Bleikertz has been a very good friend and colleague with whom I routinely discussed many topics on and off the thesis work. We shared many PhD experiences together. I especially want to thank him to get me started with sports and showing me the ropes with skiing and hiking (though he wishes climbing was here too). Zoltan A. Nagy has been running our local hardware cartel and has unconditionally provided many equipments for our infrastructure. In the spirit of providing additional motivation, he often asks "Does anyone really want your thesis work?". I am also thankful to IBM Research, Zurich for hosting me and providing an excellent work environment. Felix Marti and Steve Wise from Chelsio Communications provided unlimited support with our networking hardware. They patiently answered all my questions about RDMA hardware and helped me debug some severe bugs on a moment's notice. Thank you all.

Outside work, a wider social circle of friends helped me to maintain a healthy mental state. Over the past eight years, Peter Hladky has been an indispensable friend, and a wise mentor

for me. He has introduced me to a vast range of social, historical, cultural, philosophical, and existential topics, and demonstrated how to understand and analyze them while building a better understanding for oneself. I cherish his brotherly advices and our discussions on many personal and professional issues that have helped me tremendously to achieve clarity and focus in my life. For all the interesting Friday sessions, wonderful experiences, joyful travels, and great company, thank you Peter very much! Sami Galal has been the source of boundless energy who is up for anything, anytime. He has conspired countless number of spontaneous car trips (to Liechtenstein!), bike rides, fun hangouts, and help-out sessions while simultaneously being curious about the pronunciation of the soft-Danish "D" everywhere. Antonio Barresi's grit and determination to get things done (and enjoy the obligatory celebration afterwards), and his confidence in my abilities have been very motivating. Dominik Rüegger asked many difficult questions over the years, and has returned the favor by organizing many eventful parties and gatherings. Dilber De Vitre and Leo Fellmann routinely inquired about my well-being, and involved me in many of their life events. Suruchi and Divay Bansal have been my family away from family. They hosted me on countless occasions for delicious food tastings, and provided the necessary homely chit chat opportunities that I miss here. In a very short period of time, Ajinkya Kulkarni has become an invaluable friend who very patiently listened and gave sound advice for many of my concerns about life and work. On my second day in Zurich, I met Henry Both at a falafel shop near ETH. After which he took it upon him to show me around in the city of Zurich, and since then he has organized many Indian dinners while introducing me to interesting people, many of whom are now my close friends. Imene Batata provided a very enthusiastic and refreshing company over our countless meetings while exploring restaurants of Zurich. All of them and many more individuals have been responsible for my well-being and healthy mental state, which was crucial for making forward progress in the PhD studies. Thank you all very much.

At last, I am very grateful to my family and friends back in India. My parents, Arti Trivedi and Akhilesh Trivedi, provided a wonderful childhood, and nurtured a creative up-bringing to help me focus on things that I found interesting. From an early age, they provided me ample freedom and independence, and encouraged me to take my own decisions. My little sister, Anindita Trivedi, provided a necessary and very welcoming distraction from the work routine and consistently kept me up-to-date about what she thought I must be interested in knowing. Although, despite my sincerest efforts, she still seems rather skeptic about my interest in her stories. My grandparents, Asha-Lata Mishra and K. K. Mishra, patiently listened to my study stories from school and asked for more. For a countless number of times my grandmother rescued me by teaching me Sanskrit for school tests. My Uncle, Asish Trivedi, took it upon him to arrange for my studies in Lucknow when it was no longer possible for me to study in a town where my parents lived. My late grandfather, R. K. Tiwari, built and fostered a learning culture all around us in the first place. Mrs. C. C. Anil, who was my kindergarten teacher, was instrumental in keeping me at school when I had a tendency to run away. Her family, Mr. Anil and their daughter Yamini Prabha, also supported me many times. At last, I would like to thank my girlfriend, Christa Preisig, for providing a cheerful and supportive company that has been instrumental in making forward progress when it seemed almost impossible. She has been a great source of stability and support whenever I needed her. I can only hope that I have been able to provide her a fraction of support that I received. I thank you all from the bottom of my heart.

Animesh Trivedi

November, 2015

Contents

1 Introduction

We live in a data-centric world where large volumes of data are generated, stored, and processed everyday. These phenomenon, which are collectively termed as *Big Data*, are uniquely defined by data *volume* (multiple Gigabytes (GBs), Terabytes (TBs) or more data everyday), data *velocity* (constant, real-time data arrival), and data *variety* (data collected from multiple sources and in a wide variety of formats). Examples of a few of these Big Data environments that generate lots of data include social networks, web-data graphs, business process data, logs in large infrastructures, scientific experiments (e.g., the LHC experiment at CERN), Internet of Things (IoT) sensors, and wearable computing devices, etc.

This large amount of generated data is analyzed to make insightful and timely business decisions [198]. To facilitate this data-driven decision-making process many distributed data processing frameworks, such as MapReduce [1, 100], Dryad [175], Apache Spark [2, 367], and Naiad [252], have been developed. The performance of these frameworks is under a constant pressure to keep up with the growing demands of fast, real-time data storage and processing requirements. For example, Facebook routinely stores 500TB+ of data everyday and processes 100TB+ of data in under 30 minutes [3, 302]. To meet such performance requirements, these frameworks are typically deployed in parallel across thousands of servers inside a large, heavily networked environment such as data centers. Consequently, the end performance of these data-crunching frameworks strongly depends on the input/output (IO) performance of the many storage *and* network devices involved.

At the same time, the performance of modern network and storage devices have also undergone a rapid evolution in recent years. Ethernet, the most popular interconnect technology, supports 10, 40, and 100 Gigabits/sec data rates with single-digit microsecond link latencies. Storage technology has seen emergence of Non-Volatile Memories (NVMs) in enterprise and commodity computing. NVM storage devices, such as NAND flash and Phase Change Memories (PCM), now offer multi-Gigabits/sec bandwidths with access latencies as low as 15 μsec [138].

However, these advancements also put tremendous pressure on the performance of a single

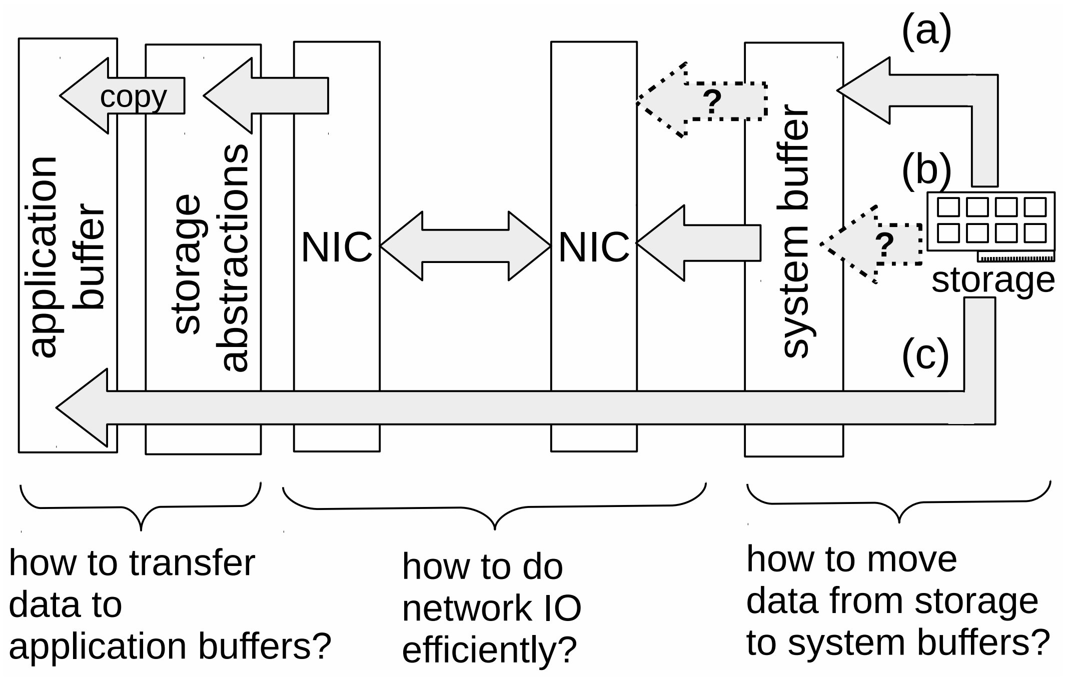

(a)
(b)
(c)
copy
application buffer
storage abstractions
NIC
NIC
?
?
system buffer
storage
how to transfer data to application buffers?
how to do network IO efficiently?
how to move data from storage to system buffers?

Figure 1.1: Comparison of data flow in three different settings. Efforts on the path-(a) focus on storage related challenges and on the path-(b), network related. This thesis takes an end-to-end holistic consideration (marked as the path-(c)) towards the data flow management on both the server as well as the client side in a distributed setting.

CPU. In the past decade, CPU speed improvements have hit a plateau, and the *notion* of a fast CPU and multiple slow IO devices no longer holds. Yet, modern data-processing stacks continue to develop on the same notion where they aim to optimize the CPU time while a slow IO operation is in progress. This way of systems building leads to adding more layers of generic operating system (OS) functionality, abstractions and services, data orchestration logic with copies, expensive protocol processing, multiple (de)multiplexing and scheduling points, device drivers, etc., in the IO processing path. All of these operations and layers must be executed by the CPU while keeping up with the high data rates. Consequently, as IO devices (and data rates) continue to improve rapidly, the CPU's (in)ability to execute the thick IO stacks of data-processing frameworks turns it into a performance bottleneck [53, 68, 208, 224, 266, 289, 330].

Multiple research efforts from industry as well as academia have been launched to reduce the hardware and software overhead by providing better IO interfaces, efficiently managing IO resources, and leveraging manycore CPUs for IO processing. However, these efforts do not take a holistic end-to-end approach towards eliminating overheads in data flows from multiple servers and devices to application buffers. Figure 1.1 shows generic steps taken by such approaches. For example, projects such as Moneta-Direct [71] illustrate how to bring data efficiently from fast flash storage to DRAM buffers, but do not discuss how to transmit it efficiently on a network. Such efforts are marked as path-(a) in the figure. On the other hand, data stores designed around high-performance networks such as FaRM [107] and fast Key-Value (KV) stores [183, 241], etc., demonstrate how to efficiently access data stored in DRAM buffers of remote servers, but do not consider how to stage data from fast storage devices to DRAM buffers. These efforts are shown as path-(b) in the Figure 1.1. On a client side, data is copied though multiple layers of storage abstractions before it arrives in a final application buffer. Additional challenges arise when considering a distributed environment where several servers and devices might be involved in completing an IO request.

To summarize, delivering high end-to-end performance to data access and processing frameworks demands attention on several fronts. A design of such a system needs to consider, among other things:

- What is a suitable interface that can be used to deliver high end-to-end IO performance for transferring data from a network-attached storage device to application buffers?

- How to design a distributed, parallel data store with network-attached storage devices from several servers without incurring additional software/hardware overheads?

- What is the right data abstraction for accessing and processing data stored in multiple servers in such a distributed data store?

In this thesis I answer these question by first applying high-performance networking concepts and IO interfaces to the storage domain to develop an end-to-end data flow path (shown as the path-(c) in the Figure 1.1). Subsequently, I develop a distributed data store, its data abstraction, and associated distributed data processing applications to tackle the challenges associated with a deployment of such a system in a distributed environment.

1.1 Thesis Statement

In this thesis work we consider the problem and challenges associated with delivering high performance data access from high-bandwidth, low-latency network and storage devices. The thesis statement is:

The performance of modern high-performance network-attached storage devices is limited by the software overhead induced by excessive operating system and application involvements in IO flows. This overhead can be reduced (and even eliminated) by leveraging the path separation concept and associated abstractions from the high-performance networking domain and extending them to access network-attached storage. The new extended and unified abstractions facilitate building distributed data stores around a set of common, high-performance IO properties, thus eliminating overhead even when multiple network and storage devices are involved to complete an IO request. This holistic end-to-end approach towards careful data flow management and orchestration in a distributed system can deliver significant performance gains for data-processing frameworks.

1.2 Contributions

This thesis makes the following three contributions.

1. Using experimental analysis, this thesis first identifies excessive OS and application involvement in data flows to be a limiting factor in delivering the full performance while accessing data on a remote high-performance storage device. It then identifies synergies between the concepts from the area of high-performance networking and recent efforts for building high-performance storage stacks for non-volatile storage to eliminate this excessive involvement. I build a case for unifying these efforts, and develop a unified device prototype that allows access to a remote flash storage using Remote Direct Memory Access (RDMA) network operations.

2. Secondly, the thesis extends the concept of the path separation philosophy on which RDMA operations are based to a distributed environment. The work addresses the key challenges of setting and preparing multiple devices separately from IO accesses. RStore, our distributed data store, achieves this separation using a set of novel interfaces and abstractions. These abstractions are designed to eliminate OS and application involvement while minimizing the synchronization and abstraction related costs in the data access path.

3. Lastly, the thesis describes the design and implementation of two distributed data processing applications on top of RStore, namely a distributed key-value sorter (RSort) and a distributed graph processing engine (Carafe). Using the API of RStore, these applications identify, pre-allocate, and pre-fetch expensive IO resources in a distributed environment

before the data-processing phase begins, thus eliminating slow operations from the fast data execution phase. The high performance of these applications validate the design choices we made for developing RStore.

Parts of this thesis work are published in:

- Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Clemens Lutz, Martin Schmatz, Thomas R. Gross. RStore: A Direct-Access DRAM-based Data Store. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, ICDCS'15, pages 674–685, Columbus, OH, USA, July 2015.

- Animesh Trivedi, Nikolas Ioannou, Bernard Metzler, Patrick Stuedi, Jonas Pfefferle, Ioannis Koltsidas and Thomas R. Gross. FlashNet: A Unified High-Performance IO Device, as IBM research report, RZ 3889, April, 2015.

- Animesh Trivedi, Bernard Metzler, Patrick Stuedi, and Thomas R. Gross. On Limitations of Network Acceleration. In *Proceedings of the Ninth ACM Conference on Emerging Networking EXperiments and Technologies* (CoNEXT '13), pages 121-126, Santa Barbara, CA, USA, December 2013.

- Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Roman Pletka, Blake G. Fitch, and Thomas R. Gross. Unified High-Performance I/O: One Stack to Rule Them All. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS13, Santa Ana Pueblo, NM, USA, May 2013.

- Animesh Trivedi, Bernard Metzler, and Patrick Stuedi. A Case for RDMA in Clouds: Turning Supercomputer Networking into Commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys 11, pages 17:1–17:5, Shanghai, China, July 2011.

1.3 Organization

This thesis is organized as follows.

In Chapter 2 we present related work by giving an overview of the evolution of high-performance networking and storage devices. In particular, we discuss multiple network interfaces, protocol stack organizations, operating system support, user-level networking, and implementation and optimization of network protocols. For storage, we examine the recent evolution of non-volatile memories (NVMs), such as NAND flash storage, in enterprise and commodity computing, and the related development of new storage abstractions and interfaces.

We also provide an overview of distributed storage solutions such as NAS and SAN, and how high-performance networks are integrated with them.

In Chapter 3 we provide the necessary contextual and background information on the Remote Direct Memory Access (RDMA) technology, its abstractions, and its implementations. We then describe our experiences and findings related to micro-architectural features of the host system that influence the performance of RDMA based systems.

Chapter 4 we make the case for unifying the interfaces for high-performance network and storage devices after identifying their common evolution patterns.

FlashNet, a software IO stack that is a proof of concept for our unification idea, is presented in Chapter 5. The stack consists of an RDMA controller, a flash controller, and a file system. Together these components enable FlashNet to use high-performance RDMA operations to access data stored on remote flash storage devices.

Chapter 6 describes the design and implementation of our distributed in-memory data store called RStore and its two applications, namely RSort, a distributed key-value sorter, and Carafe, a distributed graph-processing engine. RStore and its applications are developed keeping the path separation principle in mind.

Chapter 7 reports on our experience with porting of RStore to FlashNet, and running RSort on it.

Chapter 8 concludes the thesis by presenting our experience, a few recommendations for future hardware and stack developers, and the closing remarks.

2 Related Work on High-Performance Networking and Storage

In this chapter, we first provide an overview of the evolution, capabilities, and interfaces of modern high-performance networking devices. We subsequently focus our attention on the recent emergence of NVM storage devices in enterprise computing, their host interfaces, storage stacks, and abstractions, etc. We then discuss networked storage architectures and how high-performance network operations such as Remote Direct Memory Access (RDMA) are integrated. At various points in this chapter we provide summaries or commentaries to put the discussion in the context of the work presented in this thesis. Based upon the history of evolution presented in this chapter, in Chapter 4 we identify common high-performance input/output (IO) properties and make a case for unification of networking and storage IO operations.

2.1 Evolution of High-Performance Networking

From the early days of computing, performance problems have existed, which were solved by leveraging multiple server machines (possibly containing multiple processors) connected via a network. The network is used for communication, data transfer, and coordination. Consequently, over the past 20-30 years, a considerable amount of work has gone into the development of a fast and efficient networking stack and associated abstractions. The focus of that work has been on network interface designs, transport protocols, end-host software stacks, application interfaces, and operating system support for high-performance networking. In this section, we first have a brief look at the evolution of network IO from the 1980s to today. We provide an overview and the rationale behind various developments that shaped the modern day networking stacks found in modern operating systems such as GNU/Linux and Microsoft Windows.

2.1.1 Network Controllers and Interfaces

One of the early issues in the design of high-performance communication systems was the work division between the host CPU and the network interface controller (NIC). Questions such as where to perform protocol processing, how and from where network data is copied, how do a host CPU and a NIC communicate to start and stop network processing, what are the application interfaces and network semantics, etc., were the key design questions that influenced the evolution of early network controllers. Peter A. Steenkiste provides an excellent overview of high-speed network cards and related hardware/software optimizations as are discussed in this section [323].

The focus of the initial work in the 1980s was on the development of high-performance transport protocols that were amenable to a fully offloaded, hardware implementation in a network controller [105]. The Universal Receiver Protocol (URP) from Bell Laboratories focused on the receiving side and designed a simplified receive-side state machine with small packet types [127]. The Protocol Engine (PE) from Greg Chesson took a more general approach to the implementation of transport protocols in hardware and proposed a purpose-built network processor that was implemented in VLSI chips [83]. Kanakia and Cheriton adapted a holistic approach and presented a communication system that encompassed the network adaptor design, IO system architecture, and transport protocol design. They presented the VMP Network Adaptor Board (NAB) [184] for the VMP multiprocessor [82] system running the Versatile Message Transaction Protocol (VMTP) [81]. NAB delivered good performance by reducing demands on the host by grouping packet-processing notifications and leveraging a hardware implementation of a newly designed, specialized "request-response" VMTP transport protocol. The Nectar Communication Accelerator Board (CAB) design from CMU went a step further and proposed a network controller containing a microcontroller running a complete multi-threaded operating system [29]. Like NAB, it offloaded much of the protocol processing onto the NIC processor, but also considered desired host operating system support and software overheads in delivering good end-to-end performance. Along with specialized protocols, CAB also supported the implementation of the TCP/IP protocol suit [94].

Instead of full offloading, Bruce Davie of Bellcore suggested a flexible and high-performance end-host interface for Asynchronous Transfer Mode (ATM) networks where a part of the transport processing was done on the host [99]. The flexibility, which was aimed at programmability of segmentation and re-assembly of ATM cells, was achieved by having general-purpose Intel microprocessors processing the IO queues. In their work for the Penn/ATM NIC, Brendan et al. argue against the use of a general-purpose microprocessor in the NIC, which increases the cost and the complexity of a design [319, 337]. The Penn/ATM NIC distilled a common denominator of per-cell functionality in ATM and implemented it in the hardware while executing other, higher-level operations on the host processor. Data was transmitted from copies that were kept in the kernel, and clocked interrupts were used to reduce the processing overhead on the host CPU.

However, researchers also voiced doubts about the effectiveness and apparent benefits of complex, fully offloaded implementations of transport protocols in the NIC hardware. Fore Systems Inc. [93] and Cambridge University/Olivetti Research [146] designed ATM controllers with minimal functionality in the hardware. The rationale behind their design was that the simplicity of NIC hardware design facilitated rapid prototyping and (arguably) offered better performance by taking advantage of aggressive processor technology improvements that closely followed Moore's Law [167]. This reasoning formed the basis of the WITLESS (Workstation Interface That is Low-cost, Efficient, Scalable and Stupid) philosophy by Van Jacobson [177]. Watson and Mamrak remarked that various mechanisms developed in the context of special-purpose protocols, such as collapsed layer processing, use of microcodes, light-weight task management, data copy avoidance, efficient network buffer management, etc., were general purpose and thus also applied to general-purpose protocols such as TCP/IP [354]. Clark et al. [86], Kay and Pasquale [189, 190], and Van Jacobson [177] provided further evidence in support of this hypothesis. Clark et al. did a systematic profiling of TCP performance and found that management-related operations of a TCP/IP implementation, not the protocol processing itself, generated the majority of performance overheads [86]. Van Jacobson's analysis showed that most of the TCP packets can be processed by fewer than 200 instructions [177]. Kay and Pasquale further classified the TCP protocol-processing overheads into two classes. For small messages, which were the majority of packets in real-world LAN packet traces, *non-data-touching* operations such as network buffer manipulations, protocol-specific processing routines, operating system functions, error checks, shared data structures accesses, etc., consumed the majority of the processing time. For large messages, *data-touching* operations, such as data copies and checksum calculations, etc., dominated the processing cost.

As protocol processing gradually moved back onto the host processor, new bottlenecks were exposed in the system. Consequently support from the NIC in these matters dictated the overall communication architecture and performance. One of the critical performance issues was excessive data movement that happened during the sequential processing of the layered protocols. This movement created a bottleneck on the memory bus which did not improve at the same rate as the CPU and network technologies [108]. Furthermore, not only data was crossing the memory bus multiple times, but also CPU cycles were consumed in orchestrating this data movement from application buffers to systems buffers and eventually to the network controller using data copies and programmed IO (PIO). Contemporary RISC processors were not optimized for such heavy data movements. Other micro-architectural factors, such as complex memory-cache organizations, ineffectiveness of large caches for protocol processing, high context-switching costs, expensive system calls, etc., also introduced overheads for protocol processing on the host CPU [247, 263, 268].

To address these issues, simple, stateless operations were gradually moved away from the CPU and added back to the NIC. These operations were simple enough to facilitate rapid development of NIC hardware, but were still effective in reducing the pressure on the host CPU and the memory bus. The use of Direct Memory Access (DMA) mechanism was proposed to move data directly between host memory and NIC using a special hardware module without involving the host CPU. However, the performance implications of the interaction of a DMA operation with the host CPU memory accesses, cache hierarchy, and virtual memory management, etc., were not well understood and required further analysis. Ramakrishnan discussed these issues in detail and presented a model to analyze the performance of an application which was governed by the performance of data copies, design of memory system, NIC interface, and cache characteristics of the host CPU [282].

NIC support for on-board packet and data buffering was proposed to further reduce the number of data crossings over the memory bus. Packet buffering facilitated the offloading of checksum calculation to a NIC. Large on-board data buffering helped in replacing systems buffers (residing in the host memory) with on-board buffers (residing on the NIC). This change in the buffering location resulted in the elimination of two memory bus transfers while copying data from application buffers to system buffers. Furthermore, staging data for reception and transmission (or re-transmission for reliable protocols like TCP) in on-board NIC buffers allowed bursty transmission of data at the full network rate by eliminating data access over slow, internal IO buses. Nectar CAB supported on-board buffering, on-NIC protocol processing, and used PIO for data copy [29]. Follow-up card designs such as Gigabit Nectar [322], Afterburner [98], and Medusa [39] followed the WITLESS architecture (i.e. without offloaded protocol processing), had on-board data buffers, allowed offloaded checksum calculations, and added support for DMA operations for data transfers.

The Medusa [39] and Afterburner [98] cards from HP Labs were built on the idea of a *single-copy* stack. In a conventional network stack with the ubiquitous BSD/socket network API, data is moved as much as five times across the memory bus. The single-copy stack design suggested to replace system buffers with on-board NIC buffers and copy data directly from application buffers to on-board buffers, hence crossing the host memory bus twice or even only once if DMA was used. After the copy, data logically resided in the system buffers, but physically in on-board NIC buffers and was left there as long as possible. The idea of the single-copy stack was generalized enough to be applicable on both the sending and the receiving side. However, the single-copy architecture violated the layering principle in which part of the transport layer processing and responsibilities were shifted to the presentation layer. For example, packetization of user data (to leave header and trailer space) was done during the copy in the socket send/receive calls, and thus much before the transport processing takes place. Also, parts of the transport processing on the receive path, e.g., TCP acknowledge generation, were tied up with the application data copy, where the checksum was calculated and verified.

More recently, the demand for high and predictable performance for data-center workloads has revived the idea of programmable NICs. To reduce tail latencies, the Chronos networking stack moves network processing to the userspace and leverages application-driven partitioning to load balance requests with the help of the NIC [186]. Kaufmann et al. propose the design of FlexNIC and advocate to offload parts of the packet processing to the NIC [187]. In contrast to FlexNIC's approach, Han et al. argue that with recent advancements in packet processing the performance gap between a hardware and a software NIC has shrunk. They propose a software-augmented NIC, called SoftNIC, which presents a programmable NIC interface for development, but can even leverage hardware support, if present [152].

2.1.2 Design and Implementation of High-Speed Networking Stacks

Concepts: Seminal works from Clark, Tennenhouse, and Feldmeier gave general initial recommendations on building a high-speed communication system [87, 118, 331]. Clark and Tennenhouse considered architectural requirements and key engineering issues for the next generation of communication protocols [87]. They suggested to decouple the protocol architecture from its engineering concerns, such that the former should not restrict implementation choices for the latter with inessential constraints. They propose Application Level Framing (ALF) as the key architectural and Integrated Layer Processing (ILP) as the key engineering principles. The ALF principle recommends the use of application-specified data units or ADUs (not network packets) as the fundamental unit of data processing in a protocol stack. ILP advocates to implement all data-manipulation operations from multiple protocol layers in a few integrated processing loops. For example, checksum calculation and data copy can be combined in a single operation. This issue of a protocol design versus its implementation was revisited by Tennenhouse [331] and Feldmeier [118]. Tennenhouse pointed out that in the absence of any design guidelines, *ad-hoc, layered multiplexing* as implemented by developers introduces statistical delays and results in QoS crosstalk between applications, which leads to poor performance. In a follow-up work, Feldmeier systematically presented and analyzed a wide range of considerations for multiplexing issues (physical vs. logical) in a communication system design [118].

Data Copies: After the analysis of Clark et al. [86], many development efforts focused on efficient and optimized implementations of transport stacks by eliminating data copies. Kleinpaste et al. described a single-copy stack (or zero-copy from the point of view of the CPU) where data was transferred once to the card, but was funneled symbolically through the end-host networking stack to maintain the same stack and processing abstractions [193]. This single-copy stack required support from the NIC, which provided on-board buffering and checksum calculation facilities [322]. Other schemes that relied on the virtual memory (VM) system to avoid data copies were also proposed. Copy-on-Write (COW) mechanism from Mach and Accent operating systems was used to avoid copying data by sharing VM pages involved in the application buffers between the application and the networking subsystem in the read-only mode [20, 120]. For correctness, if the application attempted to write the buffers while they were used in transmission, a copy was done. Virtual page remapping, as demonstrated in the V kernel and DASH operating systems, allowed data transfers using mapping and un-mapping of DRAM pages between the application and networking protection domains or address spaces [80, 343]. However, this mechanism required support from the NIC to split data and packet headers, and deposit data on page-aligned buffers. Another approach was to share pre-allocated, fixed-address buffers between the application and networking domains [303]. These fixed-address network buffers restricted applications to always put network data in pre-determined locations. Chu implemented page remapping and copy-on-write techniques to achieve a Zero-copy TCP stack in the Solaris operating system [84]. Fast buffers or fbufs proposed by Druschel and Peterson combined page remapping with shared virtual memory mechanisms to provide an efficient cross-domain data transfer facility [110]. Performance gains from these techniques strongly depended on the performance of the host memory and caching architecture, MMU design, and application behavior. Brustoloni and Steenkiste presented a novel taxonomy of buffer allocation (either from the system or an application) and IO semantics (copy vs. move vs. share) between applications and an operating system [63].

Per-Packet Overheads: Further research on the optimization of the networking performance focused on non-data-touching or per-packet overheads [189]. Jeffery C. Mogul identified and provided evidence of network-locality for better protocol implementations [243]. This network-locality was analyzed at the scope of processes and suggested that incoming network packets usually typically belong to the process that most recently sent a packet. This behavior can be leveraged to build an efficient de-multiplexing system where the control block of the transport protocol can be cached to eliminate inefficient lookups when a new packet arrives. McKenny and Dove built on the idea of network-locality and showed that though it worked (even with a single entry cache) for streamed data transfers with packet trains, certain classes of workloads such as online transactions, did not exhibit such locality behavior [228]. They analyzed the performance of two proposed variants (the move to front algorithm proposed by Jon Crowcroft, and the last-sender/last-receiver algorithm proposed by Craig Partridge and Stephen Pink) for BSD's TCP de-multiplexing implementation.

To reduce the number of packets that traverse the host networking stack on the transmit side, simple segmentation offloading (e.g. TCP segmentation offloading (TSO)) was suggested. A more complex scheme, the Large Receive Offload (LRO) was proposed for TCP on the receive side [148]. The basic idea of these approaches was to let the NIC handle segmentation or assembly of multiple small TCP packets into a big packet to amortize the per-packet network processing overheads. Menon and Zwaenepoel presented a more generalized approach for receive-side packet aggregation in software, called Receive Aggregation [235]. They also proposed to offload acknowledgement packet generation in a similar spirit as TCP segmentation offloading. Chan et al. [74] designed a network fastpath architecture where pure control packets, such as TCP ACK packets, are identified and extracted early in the device driver on the receive path. The control information of the extracted packets (e.g., sequence numbers) is passed directly to the TCP stack without funneling the packet through the networking stack. Various interconnects supported the use of large packet sizes to reduce the number of packets required to carry a given amount of user data. Ethernet allows Jumbo frames which are up to 9000 bytes (in comparison, the standard Ethernet MTU size is 1500 bytes). Interrupt coalescing or disabling interrupts and switching to polling [111, 248] helped to avoid a receive livelock condition under load [282]. Chase et al. surveyed many of these techniques and quantitatively evaluated their effect with empirical experiments [75]. Segmentation offloading, packet aggregation (found as Generic Receive Offload or GRO in GNU/Linux), interrupt coalescing with selected polling (NAPI in GNU/Linux), and Jumbo frames are part of any modern-day commodity Ethernet/TCP/IP protocol stack.

Operating System (OS) Support: Much of the high-speed networking research was also influenced by the mechanisms and abstractions provided by the host operating system. Clark presented *upcalls* as an OS mechanism to facilitate synchronous communication between multiple modules of a layered implementation of network protocol processing [85]. Fast buffers (fbufs) provided a copy-free mechanism to share and transfer network data across multiple protection domains [110]. IO-lite extended the idea of fbufs to a general-purpose system to share a single copy of data between multiple subsystems, including filesystem buffering and caching for networked-storage applications [269]. Soft-timers provided a probabilistic mechanism to efficiently schedule microsecond-granular event-handling routines of network protocols, such as TCP, with least disruption to the rest of the system [30]. Druschel et al. identified improper resource accounting and the lack of request shedding mechanisms as the primary reasons for poor under-load performance of network systems [109]. They proposed Lazy Receive Processing (LRP), which combined the idea of early demultiplexing with user-triggered lazy protocol processing to deliver high performance under load.

In traditional monolithic kernels, such as UNIX [288], network protocols were often implemented as a part of the OS implementation. Although done for performance reasons, this structure made experimentation with the development of protocol difficult. On the other extreme, micro-kernel [20] and Exo-kernel operating systems [117, 140] pushed the implementation of these protocols out of the kernel into the userspace. The x-kernel project from the University of Arizona integrated network protocols as first-class citizens by providing abstractions to capture their interaction and dependencies [162, 163]. These abstractions were then efficiently compiled and mapped onto the host OS's abstractions, minimizing overheads and maximizing performance. In a more generalized approach, Scout OS, designed by Mosberger and Peterson, proposed to make the execution path of an IO request a first-class abstraction from an operating system [249]. The path abstraction provides better buffer management, efficient layered protocol execution, specialization of the common fast path, better accounting and QoS than traditional solutions.

Prevalent micro-kernel operating system designs in contemporary operating systems such as Mach [20], and factors such as the presence of multiple protocols, easier code maintenance and debugging, and the possibility of customization by exploiting application-specific knowledge, etc., led many groups to advocate building networking stacks in userspace. Initial work from Forin et al. designed and implemented userlevel IO servers (networking, storage, graphics, etc.) for Mach [125]. However, for high-speed networking devices, the userlevel IO server approach did not deliver the expected performance [216]. In the subsequent works, Thekkath et al. [335] and Maeda et al. [217] presented TCP protocol stacks which were linked as a user-level library with existing applications instead of putting the protocol stack into a separate process space. Maeda and Bershad further proposed to decouple network IO interface of an application (send/recv) from its operating system interface (for network management) for a better protocol management while keeping the OS in charge of all services other than the core send/recv interfaces [217]. Edwards et al. described an implementation of TCP protocol in userspace by leveraging the hardware features of JetStream/Afterburner NIC in a UNIX-type operating system (HP-UX) [114, 115]. Their primary challenges were the efficient handling of asynchronous events and the processing of the TCP state-machine within the scope of a user-process. Experienced gained with the development of these user-level protocol stacks helped with the development of new user-level abstractions and stacks (see Section 2.1.3).

In general to improve the modularity, customization, and extension of network protocols and more specifically in the context of userspace networking, researchers have also advocated leveraging programming languages support. Tschudin used Pascal to design flexible protocol stacks [342]. Abbott and Peterson proposed Morpheus, which is a specialized object-oriented language that was designed to capture protocol specification graphs and verify them. The actual protocol code can then be generated from a verified graph [19]. Prolac is a statically-typed, expression-oriented language whose aim is to improve readability, modularity, and the extension of protocol implementations [196] Biagioni implemented TCP using an extension of the typesafe Standard ML (SML) language [49]. Fiuczynski and Bershad used a typesafe language to ensure a safe execution of application-specific customized protocols in the operating system kernel [122]. Naturally, running code in the kernel requires support from the operating system for extensibility [45, 117]. In contrast to extensibility, Arpaci-Dusseau et al. proposed Infokernel which exposes many key information pieces from the kernel to the userspace for better management of OS state and algorithms [31]. Using such a mechanism, Gunawi et al. have implemented safe userspace-level network services [150]. These services leveraged customized TCP extensions that managed in-kernel TCP state in the userspace in an Infokernel.

Over the years, the system call interface has been the de-facto standard to request an OS service. To reduce the overhead of a system call, Soares et al. designed and implemented exception-less system calls called FlexSC [320]. FlexSC eliminated architectural and scheduling overheads associated with the traditional system call implementations that used exceptions to trap into the kernel. They demonstrated its efficiency with networked applications, such as web and database servers [321].

More recently, VectorOS [344] proposed by Vasudevan et al. aims to reduce overheads in the network software stack by organizing IO requests into collections of similar, but independent units of work, thereby providing opportunities for amortization and elimination [345]. Arrakis [274] and IX [44] are two state-of-the-art operating systems designed to deliver high network IO performance to data-center applications. Their designs eliminate the kernel from the fast data path by separating the management and scheduling responsibilities of the kernel (control plane) from network data processing (data plane). The data plane leverages device multiplexing capabilities present in modern-day network cards (due to virtualization capabilities, e.g., SR-IOV) to directly assign instances of virtualized devices to applications for zero-copy, low-overhead, high-performance network operations. PIKA [42] is a networking stack whose design was influenced by multi-kernel message-passing operating systems such as Barrelfish [41]. PIKA splits the network stack into several components that do not share data and communicate using a low-overhead message-passing layer. In their work, Shinde et al. focus on an operating system's ability to leverage abundant hardware resources of modern NICs to deliver good networking performance to applications [316]. They propose Dragonet, a networking stack design that models protocol processing as dataflow graphs and maps them to the available

NIC resources [315].

Performance of TCP: Despite all this work, questions about the viability and applicability of the TCP protocol and the BSD/socket abstraction to provide high performance remained. Rodrigues et al. identified shortcomings of TCP in a high-speed LAN environment, and argued that many applications depend upon the socket interface rather than on the TCP protocol for data transfers [290]. They decoupled the transport protocol and API abstraction, and developed *fast socket*, a high-performance socket implementation in a user-level library using Active Messages [353] as the transport service. However, Gallatin et al. presented a counter position and demonstrated that, if implemented correctly with state-of-the-art optimizations and support from the NIC and interconnect, TCP/IP can deliver very high performance to applications [139]. By enhancing the FreeBSD network stack with large MTUs with scatter/gather IO, page-aligned payload buffers, adaptive message pipelining, interrupt suppression, and checksum offloading, they reported the highest TCP bandwidths on public record at that time.

Apart from software and abstractions related overheads, many researchers also investigated the effect of micro-architectural features on the performance of TCP protocol implementations. In his analysis, Blackwell concluded that high memory performance demands of TCP for small messages came from a poorly organized protocol implementation code than the data copies [54]. He proposed *Locality-driven Layer Processing* to schedule layered protocol processing to improve performance by reducing accesses to memory. Nahum et al. also analyzed the cache behavior of network protocols to conclude that the performance of instruction/code cache has a significant effect on the network performance [253].

Foong et al. analyzed the effect of CPU performance scaling and memory bus loads on the TCP performance [124]. Their analysis suggested higher memory bandwidth demands for a reference TCP implementation than previously expected. The CPU frequency scaling did not help to bridge the performance gap due to the large number of memory stalls from compulsory cache misses on newly arrived data. They revisited the conventional wisdom of 1GHz/Gbps and illustrated that it may not hold true for small packets. They also pointed out the socket interface, which is closely tied to the TCP implementation, to be a significant part of overall overheads. Balaji et al. looked into the memory overheads generated due to socket-based network operations and confirmed multiple, excessive memory bus crossings of network data, which lead to high memory demands [36]. Mankineni et al. also studied the processing requirements of TCP/IP protocol on the Pentium M microprocessor to illustrate the high memory demands of the receive-side TCP processing [221]. All this research works conclusively showed that just faster processors would not be sufficient to handle upcoming 10, 40, and 100 Gbits/sec network speeds, a sentiment echoed previously on similar grounds [225, 263].

In follow-up works, Huggahalli et al. from Intel recognized that high memory latencies of servers prevent the host CPU to process packets at the same rate as delivered by a high bandwidth network such as 10 Gbits/sec Ethernet [161]. They proposed Direct Cache Access (DCA) mechanism to inject network data and headers directly into CPU caches to avoid compulsory cold data misses for newly arrived data. DCA also helped to reduce excessive write-back traffic from dirty CPU caches that unnecessary increased the memory traffic. To further reduce the memory overheads of NIC-CPU communication and data transfers, Binkert et al. made a case of a closer NIC-CPU integration [51]. They proposed a simple, integrated NIC design which lets the OS manage on-chip buffers directly (in a similar spirit to Afterburner) [52]. The key benefit of their approach came from having a very simple NIC design that eliminates DMA descriptor maintenance overheads (which can incur significant overheads [359]). The OS managed network TX and RX data FIFO queues and copied data in and out using PIO from on-chips buffers to the OS-internal data structures (in their case, Linux's `SKB`). The Intel IXP network processor offered a similar, tighter coupling between the core processing and networking resources such as FIFO TX, RX queues [239]. A more recent NIC interface design from Liao et al. also focuses on efficient DMA descriptor management, but unlike the previous approach, it moves the management from the NIC to the CPU [206]. Thus, it retains most of the modern NIC features such as DMA transfers, direct cache data injection, and multi-queues for multi-core scalability while simplifying the NIC-CPU interaction for DMA management.

The work on the performance of TCP/socket continues to this day. When CPU frequency scaling stopped circa 2002-2003, TCP offload engines (TOEs) were suggested and evaluated to cope with TCP processing requirements [128]. Others proposed TCP *on-loading*, where a dedicated CPU core can be used to perform TCP processing [285]. However, with necessary in-kernel TCP stack performance improvements and modest processor performance progress, the effectiveness, scope, stability, and generality of these methods were questioned and their deployment was niche [244, 317, 318].

More recent efforts have focused on the scalability and parallelization of small packet processing on manycore systems with 10, 40, and 100 Gbit/sec Ethernet. Willmann et al. analyzed the two most prevalent network parallelization strategies, namely message-based and connection-based [360]. They concluded that neither provides a perfect speed-up and that the overhead depends on the locking contention, cache inefficiencies, and scheduling overheads. Shalev et al. looked into data sharing and lock contention overheads in multicore systems [310]. They proposed an isolated stack design called IsoStack where network stack processing is delegated to a dedicated processor core, thus eliminating any sharing, dependencies and locking among processors. Pesterev et al. from MIT pointed out that executing multiple tasks (e.g., interrupt processing, system calls, top and bottom halves, application processing, etc.) related to a single network request on different cores leads to significant overheads due to inefficient cache and CPU utilization [272]. They propose Affinity-Accept, a framework that localizes all processing of a TCP connection on a single core, thus improving its locality performance. Han et al. analyzed and identified various abstraction-related overheads from sockets, file descriptors, the VFS layer, and system calls while processing small messages [153]. They proposed Megapipe, a new channel-based, asynchronous, scalable, message-oriented network abstraction. The key abstraction in Megapipe is the concept of the IO channel, which is a per-core, bi-directional "pipe" between the kernel and user space to exchange IO requests and completion notifications. Jeong et al. proposed mTCP stack that eliminated all kernel-level overheads (from shared abstractions, locking, rigid structuring, consistency, etc.) by moving the TCP stack into the user space [178]. They leveraged modern high-performance, user-level packet IO libraries to deliver raw packets in userspace for processing. mTCP provides the best performance and scalability on manycore machines, and remains the state-of-art implementation of a TCP stack on Linux.

Application-specific Networking: Multiple groups investigated developing networking stacks and abstractions in the context of a specific application. This specialization helped with achieving a highly optimized implementation of the networking stack. Back in the 1990s, with the emergence of high-performance networks many projects focused at low-latency networking in the context of Remote Procedure Calls (RPCs) [50, 179, 275, 303, 309, 327, 333], and identified overheads stemming from data copies, context switches, inefficient host interfaces, marshaling and un-marshaling parameters, packet processing, and poor protocol composition, etc.

Today, Key-Value (KV) stores are one such very useful and widely used application. Multiple implementations of KV stores exist that leverage specialized NICs using Remote Direct Memory Access (RDMA) [183, 241, 326], utilize raw user-space packet libraries [205, 207], or are deployed in a specialized system-on-chip hardware [208].

Raw packet delivery and processing is another important workload that has attracted quite a bit of attention because of its high performance requirements. Back in 1987, Mogul et al. proposed the Packet Filter mechanism (and associated interpreted language) to efficiently demultiplex packets in the kernel and deliver them to userspace protocol implementations [242]. More recently, RouteBricks is a fast and programmable packet router design built using commodity servers [104]. It elegantly parallelizes the router processing across multiple servers, CPU cores, and NICs to deliver high packet-forwarding rates. Netmap is an end-host software packet delivery framework that proposes to simplify the design and implementation of a networking stack [289]. It delivers great performance by eliminating per-packet dynamic memory allocations by pre-allocating resources, reducing system call overheads by batching and reducing copies by sharing packet buffers between the userspace and the kernel. Intel's Data Plane Development Kit exposes device TX and RX queues to applications for raw packet processing [169]. Using these frameworks, Marinos et al. presented highly-specialized networking stacks for web and DNS servers that leverages application-specific execution and requirements in a clean-slate networking stack implementation [224].

2.1.3 Networking in Distributed Multiprocessor Systems

Also in the 1990s, researchers had started to build distributed multiprocessor systems to run high-performance parallel computing applications. These applications required lower latencies and higher bandwidths than what was possible with the traditional networking stacks, and led to the development of new, more user-amenable abstractions and end-system designs. These designs explored networking support in the context of various applications and workloads, included messaging support in the memory controller (e.g., Cray T3E [26], Stanford FLASH [197], etc.) or in the cache controller (e.g., the MIT Alewife [21], the Stanford DASH multiprocessor [204], etc.), close integration of messaging with the processor architecture (e.g., the MIT J-Machine multicomputer [258], the CMU/Intel iWarp machine [59], the Manchester dataflow computer [151], etc.), and the architecture of dedicated messaging co-processor for low-latency messaging (e.g., Intel Paragon [276], Myrinet [57], etc.). Martin et al. provided an overview of the effect this work had on the cluster architectures [227].

Active Messages proposed by Von Eicken et al. focused on reducing the communication overhead by allowing communication to overlap computation [353]. They key idea in Active Messages was that the header of a message contained an address of a user-level instruction sequence (or handler) that would be executed when the message arrived. The handler extracted the data payload and integrated it in the current computation. Compared with the traditional messaging models — blocking send/receive (cannot hide network latencies) and non-blocking send/receive (requires complex buffer management, scheduling), active messages offered an asynchronous messaging mechanism with good performance and without the complexity of buffer management and scheduling. Brewer et al. proposed Remote Queues (RQs) as a communication model, in which they decouple the message arrival from its handler invocation in Active Messages to allow further optimizations with better programming semantics [60].

Efficient NIC-CPU Interaction: A set of efforts focused on integrating network communication into the bigger problem of application/processor coordination in a distributed environment. Hence, these efforts required support from the system architecture for efficient NIC and CPU interfaces for communication and coordination.

Henry and Joerg provided a general taxonomy of NIC-CPU integration mechanisms, namely OS-managed DMA-based interfaces, user-level memory-mapped interfaces, user-level registered-mapped interfaces, and hardwired interfaces [155]. They proposed and analyzed a tightly-integrated NIC-CPU architecture to significantly reduce the communication latencies. For example, using their NIC with a register-mapped interface remote reads or writes can be accomplished in just two RISC instructions. Mukherjee et al. pointed out the high cost of writing *uncachable* NIC resources (registers, and queues) for fine-grained communication when initiating IO by posting IO descriptors for conventional DMA-based interfaces [251]. They presented a coherent network interface (CNI) that integrated into a CPU's memory hierarchy and participated in the coherency events by snooping. This design allowed fast and efficient cache-assisted communication between a NIC and a CPU.

The SHRIMP (Scalable High-Performance Really Inexpensive Multi-Processor) project from Princeton University proposed a new application communication abstraction based upon remote virtual memory mappings [55, 56, 112]. In remote virtual memory mapping, a sender exported a segment of its address space as an IO buffer, which was imported and *mapped* by the intended receiver as its local IO buffer. This initial step before the data transfer decoupled the fast, low-overhead data movement from the expensive memory segment setup. When the sender application read or wrote data on these special exported IO segments using processor load and store instructions, the SHRMP NIC snooped the data, and network IO (called *automatic update*) was triggered. Another variant called *deliberate update* required the sender to explicitly issue a send command. In a similar spirit, Thekkath et al. argue to build distributed systems using the tighter coupling provided by separating the segment setup and notifications from data transfers [334].

The Stanford FLASH (FLexible Architecture for SHared memory) project efficiently fused user-level low-overhead message passing with cache-coherent shared memory abstraction to provide a single machine abstraction on top of nodes connected using low-latency, high-bandwidth interconnects. It used a custom-designed node controller that integrates the memory controller, IO controller, network interface, and a programmable protocol processor [197]. Reinhardt et al. proposed the Tempest interface specification and its implementation Typhoon [286] that was not restricted to one network IO abstraction, but was flexible enough to support multiple abstractions. To develop hybrid, application-specific communication abstractions (shared memory or message passing), the authors identified four basic mechanisms, namely, low-overhead messages (similar to Active Messages), bulk node-to-node transfer, virtual memory management, and fine-grained access control. Typhoon was an implementation of the Tempest interface specification together with the proposed Stache protocol to provide a functional, flexible, fine-grained user-level shared memory abstraction for parallel programs.

More recently, Mario Flajslik and Mendel Rosenblum present a low-latency Ethernet NIC design that is aimed for small request-response protocols [123]. It delivers good performance by encapsulating Ethernet packets into a single PCI transaction. Scale-out NUMA (soNUMA) proposed a low-latency distributed in-memory processing model to bind DRAM from multiple machines into one single memory domain [259]. It simplifies the network protocol state-machine by specializing it for maintaining coherency and then integrated it into a custom remote memory controller. In a follow-up work on soNUMA, Daglis et al. present a network interface design for manycore system-on-chip (SoC) systems [97]. To keep the NIC-CPU communication overhead low (for low latency) and to reduce data traffic on the on-chip communication fabric (to deliver high bandwidth), they present a split network interface (NI) design. A *frontend* part in the split design is put with every core and interact with the application to initiate network operations. A *backend* part which is put on the SoC edges, is responsible for data movement without overburdening the SoC fabric.

User-accessible Network Controllers: User-accessible network controllers were first introduced in series of supercomputers such as Thinking Machines CM-5 [203] and Meiko's CS-2 [43]. The design was then also supported by other concurrent projects such as Typhoon [286], memory-mapped NIC for SHRIMP multicomputer [56], and Stanford FLASH system [197]. Follow-up works in the field focused on commoditizing the technique commodity network equipments such as ATM.

Application Device Channels or ADCs proposed by Druschel, Peterson, and Davie provided a mechanism to bypass the host operating system for network IO by giving applications direct, but restricted and safe access to the OSIRIS ATM NIC's memory [111]. This memory, which represented RX and TX queues, was divided into processor-addressable pages and mapped into an application address space. Application data was, however, maintained in the host memory, and only IO descriptors were posted into the userspace-mapped queues. Virtual circuit identifier (VCI) of an incoming packet was used to de-multiplex incoming packets. Praat and Fraser proposed Arsenic, a user-accessible NIC for Gigabit Ethernet [280]. Arsenic differed from its predecessors in that it enabled flow-level multiplexing in the NIC on connectionless datagram networks such as Ethernet. It achieved this by associating each flow with a virtual interface (with its own TX and RX queues), and giving applications direct access to the interface using programmable packet filters. Support from these user-accessible network controllers was crucial for the development of various userspace networking stacks [98, 99, 217, 319, 335].

Building upon the concepts of user-level networking, Von Eicken et al. proposed U-Net, a communication architecture that provides direct, but secure access to virtual network interfaces to applications [352]. In U-Net, every application had its own set of network resources consisting of message queues (send, receive, and free queues) and associated communication segments that hold the network data. The resource setup was mediated by the operating system and supported by the NIC hardware. All buffer management and network processing were moved to userspace, thus allowing optimized implementations of network protocols. Data was sent and received directly from userspace communication segments (or even application buffers). The NIC multiplexed the packets to the right message queue using network tags (e.g., VCIs on the ATM networks). The flexible network abstraction of U-Net offered high performance for both traditional protocols, such as TCP and UDP, and novel communication abstractions, such as Active Messages.

The Hamlyn network interface initially proposed by John Wikes [358] and later implemented by Buzzard et al. [66] from HP Labs is a sender-managed interface for high-performance, low-overhead inter-processor communication. In Hamlyn, the sender not the receivers chose the destination buffer address where the data was received at the receiver. This design reduced the complex buffer management, multiplexing, and control transfers at the receiving side and avoids packet losses due to receive buffer overruns. Like U-Net, both the sender and the receiver were given direct access to the networking hardware by pre-allocating and preparing communication buffers and resources, which were validated by the operating system before being installed on the network card. Data movement and control notifications were separated, and data could be moved directly between the application memory and the network without interrupting the host.

2.1.4 High-Performance Networking APIs

In this section we provide a basic overview of key modern high-performance networking APIs. We use the term networking API in its broadest sense as many of the following APIs are more than just a networking API. However, all of them can be used as a basic building block that can be utilized in developing high-performance applications in a distributed environment. An application may choose one API over another based upon the level of control it wants to exercise over the network IO. Although the list of APIs which are discussed below is not comprehensive, we provide a general theme of these APIs and put them in context with the work presented in this thesis in the Comments Subsection(2.1.4.6).

2.1.4.1 Virtual Interface Architecture (VIA)

Virtual Interface Architecture or VIA [91] was an industry effort to standardize many high-performance networking efforts, in particular user-level networking under a common framework. Before VIA, several proprietary System Area Networks (SANs) were developed that delivered high performance, but sacrificed the portability of applications written for them. Commodity networks, such as Ethernet, were standardized, and supported a simple development model, but failed to deliver performance that was needed for high performance computing (HPC) applications. Hence, the key goal of VIA was to strike a balance between high performance and standardization of interconnects. At the same time, several academic projects (U-Net, Hamlyn, ADC, SHRIMP etc.) explored and established a common set of high-performance properties which were eventually included in the VIA specification. The VIA specification provides the abstractions, semantics and principles to integrate many of the research ideas, such as operating system bypassing, user-level networking, splitting of the data and control setup [352], virtual-memory mapped communication [112], and explicit memory management [358], etc., into a viable end-host networking stack [65].

The VIA specification virtualizes the network interface and provides each client or VI consumer with its own private, directly accessible virtual network interface (VI). Each VI can be connected to a remote VI and has a set of queues for data transfers. VI consumers can asynchronously post IO descriptors on these queues to initiate data transfers. IO completion notifications are delivered on a separate queue. IO is performed from previously pre-registered memory buffers. The VI architecture supports two types of data transfer models for network IO. In a Send-Receive model, which is similar to the traditional message passing, a sender explicitly sends a message and the receiver specifies where to receive the data. The second model is the Remote Direct Memory Access (RDMA) model where the sender (after acquiring the necessary credentials and capabilities) directly reads or writes buffers in remote memory locations.

VIA was also supported by the emerging and now popular, InfiniBand interconnect. The InfiniBand specification was later standardized by InfiniBand Trade Association (IBTA). The standard contains details of an abstract end-host interface called Verbs, which is a superset of the VIA specification. Today, Open Fabric Alliance (OFA) [16] provides the OFA Enterprise Distribution (OFED) stack, which is the reference implementation of the Verbs specification on multiple platforms. We briefly discuss VIA, InfiniBand, and OFED in more detail in Chapter 3. A more comprehensive introduction to the VIA/RDMA can be found in Philip Frey's PhD thesis work [129].

2.1.4.2 Portals

Portals is a low-level network API for high-performance networking IO [287]. It is developed by Sandia National Laboratories and the University of New Mexico. The key concept in Portals is the definition of elementary building blocks called Match Entry. Broadly speaking a match entry contains a packet matching criteria and associated actions that can be combined to support arbitrary upper-level IO semantics in the network. An incoming message can be tagged and matched against a match entry based upon source or destination id, job id, user id, 64-bits match or ignore entries, etc. When a match entry matches, the corresponding memory descriptor buffer is used to receive the rest of the data. Multiple match entries (in a preference order) can be linked to make a list, called match list. A match list is then installed in a slot in a Portal table that is attached to a NIC. A slot in a portal table corresponds to a specific high-level protocol such as MPI. This basic mechanism allows building *receiver-driven* network IO where upon receiving incoming data, a Portals implementation (either in hardware or software) demultiplexes incoming messages based upon their port table entries, and then walks the match list to match the tag present in the received message to available entries to determine where to deliver data.

The Portals API solved three significant issues associated with VIA in supporting large-scale HPC infrastructure [61]. First, Portals API provided more expressive, higher-level events and network operations such as triggers, counters, and event types etc., than VIA. These mechanisms help in developing light-weight, efficient, and fast higher-level implementations of operations such as MPI collective IO. Second, Portals API simplified network receive processing (via building blocks matching) while simultaneously providing very expressive and powerful network operations to upper-level protocols. In contrast, VIA imposed strict FIFO ordering for processing receive queues. The matching-based receive provided network independence and isolation while guaranteeing progress for applications. And lastly, in order to scale to thousands of systems, Portals nodes were made connectionless and required no connection-specific establishment and resource management.

Apart from these differences, Portals also shares many of the key design choices with VIA such as OS-bypass, zero-copy, user-space application-private networking. Like VIA, Portals also supports two-sided and one-sided network operations with additional matching-based network receives operations.

Portals is used as the lowest-level native network programming API for their custom designed SeaStar network interface on the XT3 system.

2.1.4.3 Open Fabrics Interface (OFI)

Open Fabric Interface or OFI is a next generation of communication interfaces from OpenFabric Alliance [149]. OFI focuses on meeting the performance and scalability requirements of high-performance computing applications, and enterprise applications running in a closely coupled setting on a high-performance interconnect. It also extends semantics of network operations to integrate Non-Volatile Memory (NVM) storage accesses as a first class citizen. Multiple prototypes (such as our proposal for FlashNet, Chapter 5) and products (IBM Active Storage [119], Mangstor NX-Series products [223]), already provide RDMA based access to remote flash storage.

For a better development and analysis of key requirements, OFI breaks the full stack into application interfaces, provider libraries, kernel services, daemons, and test applications. Also, OFI borrows and expands on many of the key ideas from other popular interfaces such as Portals, and MPI. For example, it supports tag-matching based receives for MPI as done in the Portals interface. The implementation of OFI focuses heavily on improving performance and scalability. For performance, it streamlines code by removing branches, eliminates unnecessary and residual data structures from the old OFA verbs API, provides cache-aligned new structures, etc. For scalability, it introduces an address vector interface which greatly reduces the memory requirements per connection.

2.1.4.4 Unified Communication-X (UCX)

Unified Communication-X (UCX) is a recent proposal from multiple industry and academia groups to unify high-performance programming models [311]. UCX designers argue that although there have been multiple efforts in the field of high-performance computing to unify networking interface, they do not provide portability, performance, openness, and generality of UCX. Beside providing a unified framework, UCX also considers the next generation of more heterogeneous computing platforms with massive threaded cores, hierarchical memories, and specifically computing accelerators such as GPGPUs.

To achieve its goals, UCX breaks down networking and compute concerns into three main pieces. These components communicate using a clearly defined public API. The first component is a common service framework called UC-Service or UC-S. UC-S provides general common functionality, resources (e.g., memory buffers), and common helpful routines used by other components. UC-Transport or UCT is the unified low-level transport service that implements communication protocols specifically to the underlying transport system such as InfiniBand Verbs, IBM BG/Q Torus, Crays uGNI, or OFI libfabrics, etc. It defines interfaces for immediate (short), buffered copy-and-send (bcopy), and zero-copy (zcopy) communication operations for small, medium, and large message transfers. And lastly, UC-Protocol or UCP implements higher-level protocols that can be used to build application-facing front-end APIs such as MPI or PGAS. Currently, UC-P provides interfaces for initialization, Remote Memory Access (RMA) communication, Atomic Memory Operations (AMO), Active Message, Tag-Matching, and Collectives.

2.1.4.5 Programming Abstractions

Parallel programming abstractions provide a higher-level programming frameworks and constructs than low-level networking APIs. These frameworks dictate how general high-performance problems such as weather forecasts or particle simulations can be broken down in to small parallel tasks, how these tasks coordinate for processing, how data is distributed and shared across these tasks, etc.

Two fundamental models in which parallel processes can interact are message passing (or distributed memory) and shared memory.

Shared Memory: Shared memory is a mechanism to share data between tasks. The data could be interpreted for locking, synchronization, ordering etc. or actual program data. The data is stored in a globally shared address space which parallel jobs can read and write asynchronously.

The GASNet (Global-Address Space Networking) project from Berkeley designed this idea of global parallel, shared address space for multiple parallel tasks in a standardized form [5]. It decoupled language-independent abstractions from their low-level implementations over a variety of networks. Using this model, various run-time libraries and language-extensions with compiler supports were developed. Follow-up work in the field identified that not all memory addresses are equal and local memory accesses are naturally faster than remote memory accesses. This observation led to development of the Partitioned Global Address Space (PGAS) models which merged the performance of local accesses with the simplicity of global data referencing for parallel programs. The PGAS model is the basis of Unified Parallel C (UPC), Co-Array Fortran, Fortress, Chapel, X10, Global Arrays and SHMEM programming APIs. The PGAS model (or the shared memory model) is arguably simpler to program than explicit message passing [23].

Message Passing: As the performance and scaling issues with early shared-memory multiprocessor systems became evident, the focus of systems design shifted on efficient message passing interfaces. In the message passing abstraction, parallel tasks exchange data, communicate, and coordinate via passing messages to one another. These messages can be sent in an asynchronous or synchronous ways.

A message passing framework can leverage various degrees of support from an underlying network to deliver good performance to applications. Hence, multiple commercial and academic projects implemented their own proprietary versions of message passing frameworks optimized for their systems. Message Passing Interface (MPI) API was born out of a necessity to unify these fragmented efforts to provide a standard messaging interface between processes running on a distributed multi-processor system [250]. In 1994, the first version of MPI specification was released. It laid out details about message semantics and a run-time environment support for MPI communication. MPI has been tremendously successful in the high-performance computing arena [147]. Since then, two more revisions, version 2 (MPI-2) in 1997 and version 3 (MPI-3) in 2012, have been made. MPI-2 added support for dynamic resource management and remote memory operations. MPI-3, the latest version of MPI, introduced non-blocking collective operations and a more efficient use of one-sided operations [157]. In essence, MPI provides various communication primitives (point-to-point, collective, one-sided, and IO operations), declarative concepts (datatypes, groups, processes, and topology) and associated toolchain (linking and runtime).

Summary: In comparison to other low-level networking APIs discussed so far, programming models such as MPI or PGAS, are not native networking APIs, but are broader programming frameworks that define communication as well as computation abstractions for parallel programs. We mention them here because these frameworks can also be used in the context of raw networking and storage IO (e.g., MPI-IO for storage). Most of their ideas are well established, properly documented, and evaluated. Multiple implementations of them are available on low-level networking APIs discussed in the previous section [58, 154, 210, 214]. These implementations hide the complexity and heterogeneity of underlying transports and NIC designs. Hence, it is a responsibility of implementations to deliver performance close to the raw networking by leveraging the right set of operations.

2.1.4.6 Comments

The aforementioned high-performance APIs provide a variable degree of control on communication and performance delivered to applications. HPC networking interfaces such as Portals which are primarily developed keeping hardware support in mind (e.g., Quadrics/Elan4, CNIC on ASCI Red supercomputer, Myrinet, etc.), offer rich network semantics and functions. These functions include collective network communication, conditional notifications, selective receive, arbitrary IO ordering, and programmable triggers, etc. In most cases, NICs of these interfaces have advanced programming functions to provide QoS, manage memory buffers, setup connections, and execute network logic etc. However, none of these APIs inherently depend upon the availability of such features in the interconnects and can be implemented on top of simpler NICs. In contrast, commodity interfaces such as VIA (and follow-up networks e.g., InfiniBand, iWARPs, RoCEE, etc.), were born out of commodity computing (from U-Net and successors) and focused on providing an efficient end-host networking stack design for point-to-point communication. This design included memory-mapped user-space IO, zero-copy data transfers, one-sided operations, and OS-bypassing, all of which were also present in the HPC interfaces. These commodity interfaces required support from applications as well as the OS to manage communication resources such as connection endpoints, and memory buffers, etc. So far, in contrast to HPC interfaces, which can scale up to 10,000s and 100,000s of nodes, commodity interfaces have been proven successful for small to moderate size clusters (100s to 1000s of nodes). In this thesis, we choose to use a low-level VIA-inspired interconnect (iWARP) because (a) VIA being a basic point-to-point networking IO gave us maximum flexibility and control over communication resources when developing a distributed service like RStore; and (b) the general availability of hardware/software stacks enabled the open development without being tied-up to a particular solution.

2.1.5 Summary

Thanks to Moore's Law, the CPU speed doubled every 18 months in the 1990s and provided the much required performance boost for protocol processing on the host CPU. Consequently, outside HPC, interest in the active development of offloaded, fully-programmable network controller designs for general-purpose commodity computing diminished because the host CPU delivered competitive performance. Further research in this field (see Section 2.1.3) was driven by the interest and performance demands of emerging parallel and distributed applications. Commodity efforts focused on improving the performance of TCP/socket-based computing. However, much of the research done in the 1990s is finding its way back in to data-center computing, as the stagnant CPU speed improvements (in comparison to network and storage devices) have again made it a potential performance bottleneck.

2.2 Evolution of Non-Volatile Memories

For many years, persistent storage as provided by spinning hard disks underwent little performance improvements. In comparison, the processor, memory, and interconnect technologies continued to improve on a regular basis. However, this situation changed rapidly with the introduction of Non-Volatile Memories (NVM) in desktop and enterprise computing almost a decade ago. Since then, many new system architectures, storage stacks, abstractions, and performance enhancements have been proposed to deliver this phenomenal performance improvement in the storage hardware to data-crunching applications.

In this section, we first give a general overview of the physical attributes and performance of current and emerging Non-Volatile Memory technologies, focusing on the NAND flash and related technologies in the context of performance. We then discuss various host interfaces through which they are attached to the host and finally look at the large body of work done in improving the performance of software systems by developing new abstractions and optimizations on top of NVM storage.

2.2.1 Current and Emerging NVM Technologies

Flash Memory: Flash memory was invented in the 1980s at Toshiba. Early applications of flash memory were in embedded devices, cameras, mobile phones, and electronic equipment, etc. Flash memory stores data in memory cells made out of floating-gate transistors. These transistors trap electrons and alter their voltage characteristics. This action is called *programming* the cell. This change in voltage is sensed by measuring the current flow, whose absence or presence determines weather 0 or 1 is stored in that cell bit. To increase the storage density, multi-level cell (MLC), including triple-level cell (TLC), devices were invented. These devices can store multiple bits in a cell, and the amount of current, rather than just its presence or absence, determines the stored bits. Before storing data again, a cell must be *erased* or *reset*.

There are two main types of flash memory, namely NAND and NOR flash, resembling their namesake gate behavior. From a system point of view, NOR flash provides random byte-addressability and is typically used to store BIOS and firmware routines that do not change often. Commercial flash storage uses NAND flash, which has lower program/erase latencies and a higher density than NOR flash. In a typical packaging, NAND flash cells are organized into 2-8kB *pages*. 64 to 256 of these pages are grouped together to form a *block*. A page is a basic unit of read or program, whereas a block is a basic unit of erasing. A 4kB NAND flash page can typically be read in 60-100μsec and a block can be erased in a few milliseconds. However, NAND cell blocks can only be programmed/erased a finite number of times, usually 100K times for SLC and 5K-3K times for MLC. The decay of NAND flash cells that gradually become unable to store data or incur a very high error rate is called *wear-leveling* of flash. To provide a device-wide effective and uniform wear leveling, a redirection layer between the system's view of logical addresses and the physical page addresses on flash, called Flash Translation layer (FTL), was introduced. During a NAND flash block erase, the FTL enables a flash controller to transparently move data from the old to the new physical flash pages without affecting the system's view of where data is stored. However, the different sizes of the flash pages (read granularity) and blocks (erase granularity) present some interesting design and performance trade-offs when building a flash storage system. Agrawal et al. provide an excellent overview of these trade-offs [22]. In their work for Gordon supercomputer, Caulfield et al. [70] describe in detail various system designs and trade-offs when integrating flash into a data-centric system. They also proposed an FTL design that was tailored towards the needs of data-intensive application. Unsurprisingly, the design and implementation of an effective flash

	Read	**Write**	**Endurance**
DRAM	25-50ns	35-100ns	10^{18}
STT-MRAM	29ns	95ns	10^{15}
PCM	48ns	150ns	10^{8}
NAND SLC flash	25μs	200μs	10^{5}

Table 2.1: IO latencies and endurance summary of various storage technologies [90]. The next generations of PCM and STT-MRAM deliver a performance very close to DRAM while providing persistency and durability of the stored data.

controller and a FTL are very active research fields.

Phase Change Memory (PCM): Phase Change Memory (PCM) is the most viable memory technology to succeed flash as the non-volatile storage of choice. PCM stores data bits by changing the crystalline state of a Chalcogenide material by applying heat produced by the passage of an electric current. The crystalline state, amorphous or crystalline, has distinct resistive properties, which can be used to determine the bit stored. Unlike NAND flash, PCM bit cells can be manipulated individually, hence making it bit-addressable. PCM cells do not require an explicit erase operation. Although PCM cells are 10-100× more durable than the NAND flash cells, they still require some wear leveling. However, in the absence of different program vs. erase size granularities, the wear leveling mechanisms of PCM cells are much simpler.

The performance and organizational characteristics of PCM memories are projected to be close to those of DRAM. Hence, many researchers are considering it to be a viable alternative to replace or augment DRAM as a main memory technology [200, 281]. From a system-building perspective, researchers are treating PCM as being available on the memory bus and accessed using the CPU load and store instructions.

Beyond PCM: Many materials and technologies, such as Spin-transfer torque magnetic random-access memory (STT-RAM or STT-MRAM), Ferroelectric RAM (FeRAM, F-RAM or FRAM), Magnetoresistive random-access memory (MRAM), and Memristors, etc., are currently undergoing a rapid development. These technologies offer similar structural organizations as PCM, but offer lower energy cost, higher endurance, and improved performance.

Table 2.1 summaries the performance and endurance characteristics of various non-volatile memories. For a comprehensive treatment of the physical properties of non-volatile memories, please refer to the Non-Volatile Memory Technology Database (NVMDB) at UCSD [328].

2.2.2 Host Interfaces

As SSDs: There are several ways in which non-volatile memory technologies can be attached as a device to a host system. In its simplest form, NVM technology such as NAND flash can be packed in a disk form factor and attached to a Serial ATA (SATA) or Serially Attached SCSI (SAS) bus. This packaging is called Solid State Disks (SSDs). These SSDs are available from a number of manufacturers such as Intel [172], Samsung [293], Segate [305], etc. The key advantage of packing NVM storage as SSDs is that the system software (drivers, the kernel and the block layer) can use them without modifications. Although helpful in the initial integration, the limited capacity and high overhead of modern storage protocols become a major performance bottleneck when accessing low-latency (in nanoseconds) and high-bandwidth NVM storage [68]. Additional operating system involvement due to system calls, file system caching, context switches, etc., can cause up to 50%-63% performance overhead.

On the PCIe bus: To remedy this situation and deliver better performance, flash memory can be attached directly to a PCIe bus interface, thus eliminating slow SATA/SAS interfaces [69, 137]. Caulfield et al. proposed Moneta, which is a PCIe-attached flash device with a programmable controller [69]. The controller implements separate queues for reads and writes, performs bandwidth balancing via round-robin stripe buffer allocation to requests, and hides the access latencies by exploiting the parallelism of multiple flash dies. Numerous software optimizations have also been proposed, including bypassing the block IO scheduler, atomic request issuing, allowing multiple threads to handle interrupts using atomic reaping, and using polling instead of interrupts under heavy load. Moneta Direct (Moneta-D) is a follow-up work on Moneta in which further overheads due to entering the kernel for system calls and performing file system permission checks on every IO request have been eliminated [71]. File system checks are offloaded to a capable hardware, and virtualized, application-private IO channels, where DMA requested are posted directly, are used by a userspace library for IO operations. FusionIO also developed its own implementation of PCIe-attached flash and commercialized it for enterprise computing [137]. Their interface specification is close sourced. Unlike Moneta, FusionIO runs a part of its controller and the FTL implementation on the host processor [181]. Onyx is a prototype high-performance PCIe-attached Phase Change Memory with an FPGA-based controller [25].

Over the past couple of years, many vendors have developed their proprietary interfaces to PCIe for accessing the attached flash controller. The Non-Volatile Memory Host Controller Interface Specification (NVMHCI) (also known as NVM Express or NVMe) is an attempt to standardize the access interface of these devices [10]. The controller is designed from scratch to deliver high performance by having a low software overhead, and exploiting internal parallelism of flash chips. The first NVMHCI 1.0 specification was released in 2008, and compliant devices are available from several vendors [171, 294].

On the Memory Bus: Next-generation NVM technologies, such as PCM, will necessitate integration on the memory bus because of their close-to-DRAM performance characteristics.

eNVy, proposed by Wu and Zwaenepoel, was one of the early systems to experiment with such a design [362]. Although using block-addressable flash chips, eNVy presents a byte-addressable DRAM-like access interface to the host CPU on the front-side memory bus (FSB) using a custom memory controller. To hide large memory write latencies, eNVy uses a battery-backed SRAM as a write buffer for copy-on-write pages. Mogul et al. have argued for a hybrid approach where pages can be moved between DRAM and NVM depending on their update frequencies [245]. Qureshi et al. [281] and Lee at al. [200] explore the possibility of replacing DRAM with PCM for better density and energy efficiency. Bailey et al. discuss the OS implications of having a fully-persistent main memory [34]. Integrating persistent memory into the processor's memory hierarchy raises many questions about ordering, consistency, durability, and failure semantics [47], etc. These questions are part of the active research done in the field that is exploring new memory controller designs [212, 370], hardware primitives [92, 113, 254], and programming abstractions [89, 347, 351].

2.2.3 Storage Systems, Abstractions, and Stacks

In 1994, Douglis et al. examined the possibility of using flash storage in mobile computers [106]. Their study concluded that although very helpful with improving the read performance by an order of magnitude and decreasing the energy consumption by 90%, in absence of the considerable software precautions, the garbage collection procedures of the flash decrease the write performance by a factor of 10. In this section, we cover the relevant work that led to the development of such considerate system software and application interfaces for integrating NVM technologies.

File Systems: A file system presents a known and familiar interface to applications. Consequently, many flash and NVM-aware file systems have been proposed to deliver high storage performance to applications. Kawaguchi et al. were the first to consider the asymmetric performance of flash read and writes, the absence of in-place updates, and its limited endurance in a file-system design [188]. They leveraged the Log-structured file system designed proposed by Rosenblum and Ousterhout [291] to design a flash-based file system. Many other file systems for embedded systems that accessed NAND flash chips directly, such as JFFS [361], YAFFS [18], Microsoft's FFS [226], etc., were also developed. Because of the limited support from embedded hardware, these file systems also dealt with issues of FTL management, garbage collection, and wear-leveling, etc. The Direct File System (DFS) by Josephson et al. proposed to virtualize flash chips at the FTL level by providing a very large storage address space [181]. The virtualization layer, which also included the responsibilities of the FTL, block allocation, garbage collection, etc., helped in simplifying the design of a file system. However, the DFS design requires support from the flash device. F2FS is a more general, device-independent, mature state-of-the-art GNU/Linux file system that was designed from scratch to handle flash idiosyncrasies in the SSD format [201]. Other general file-system-level performance and design optimization works include Anvil [356] and Nameless writes [368].

Other researchers have focused on non-flash-based file systems, which are better suited for emerging PCM storage. Condit et al. propose Byte-addressable Persistent File System (BPFS) to leverage the byte-addressability of PCM storage [92]. BPFS uses short-circuit shadow paging (SCSP) to provide atomic, fine-grained updates (in-place write, in-place append, and partial copy-on-write) in BPRAM. SCSP relies on two new proposed hardware primitives: atomic 8-bytes writes and epoch barriers. The atomic 8-byte write (even in the case of a power failure) ensures that the pointer updates are consistent and atomic. Epoch barriers are necessary to get guarantees on the memory ordering done by modern memory controllers. These new primitives help maintaining atomic pointer updates in internal file and directory structures of BPFS with strict ordering guarantees. Such hardware support is also assumed by others [89]. Persistent Memory file system (PMFS) is a light-weight, POSIX-complaint kernel-level file system that provides direct access to persistent memory via memory-mapped IO to applications [113]. The work also proposes a new simple hardware primitive called *pm_barrier* to ensure durability of writes that are flushed from CPU caches. Unlike BPFS, PMFS uses larger in-place writes, maps storage directly into the application address space using mmap, and assumes simpler hardware support. Storage-class memory file system (SCMFS) leverages the large and sparse virtual memory address space to lay out files from byte-addressable persistent storage into large contiguous virtual address chunks [363]. Aerie is a flexible file system architecture [350] that implements most of the functionality in the user space [349]. The kernel's role is reduced to just multiplexing memory. To demonstrate the flexibility of using Aerie's interface, authors have implemented a POSIX-compliant PXFS and a simplified key-value file system called FlatFS. Quill [329] is another file system that provides direct user-level access to file data through memory mappings.

Memory Hierarchy and Caching: Several projects investigated the integration of NVMs in the memory hierarchy in various capacities. FlashVM uses flash storage as a fast paging device while optimizing the performance and reliability of the device by using the knowledge and usage pattern from the virtual memory management system [299, 300]. A key advantage of this approach is that it does not require any modifications to applications. Badam and Pai proposed SSDAlloc, a hybrid and integrated SSD/DRAM memory management system [33]. SSDAlloc transparently manages application objects between DRAM and SSD while preserving an application's view of the virtual addresses of these objects. Hence, applications can transparently extend their memory footprint to the combined capacity of SSDs and DRAM.

Other ways of using flash SSDs include using them as logging device [78], as transparent fast cache [48, 191, 301], or letting applications such as databases manage their own storage on flash [202].

Persistent Data Structures: The availability of persistent byte-addressable memories facilitates the development of persistent data structures as the storage interface. Such an architecture does not restrict NVM into a second-tier storage role and brings it closer to the applications. The key challenges in building such data structures are how to ensure correctness and durability in the presence of failures. Three concurrent projects, namely Mnemosyne [351], NV-heaps [89], and Consistent and Durable Data Structures (CDDS) [347], provide ways to develop generic and safe data structures in NVMs to user-applications. NV-Heaps allows applications to allocate, reference, and de-allocate space in a NVM storage [89]. It uses software transactional memory (STM) and redo logging to provide transactions. HV-heaps assumes hardware support in terms of 8-byte atomic pointers and epoch-based data flushes [92]. Like NV-heaps, Mnemosyne also provides low-level transactional support to allocate and manage data structures in NVMs [351]. However, Mnemosyne is language independent (hence cannot provide type-safe pointers) and assumes no modifications to processor hardware. It uses a combination of `mfence` and `cflush` instructions to provide consistency and durability. Consistent and Durable Data Structures (CDDS) developed by Venkataraman et al. use versioning to provide atomic updates and rollbacks to build data structures in NVMs [347].

Instead of providing a general framework to develop any data structure, Yang et al. [365] and Chen et al. [79] focus specifically on building B+ trees, an important data structure in databases, in NVMs. Whole System Persistence (WSP) proposed by Naranyan and Hodson focuses on keeping data in DRAM persistent by taking advantage of a small residual energy in systems to *flush data on fail* (rather than *flush data on commit*) from CPU caches and registers on a power failure [254]. Other relevant works in the field of consistent and recoverable data include failure atomic-msync [270], Rio Vista [213], and RVM [297], which dealt with keeping application and operating system data consistent in the case of a failure. Other work looked into efficiently implementing transactional IO on non-volatile memories [88, 279, 298].

Performance: Many efforts focused on the design and implementation of fast storage stacks on modern NVM hardware and computing platforms. Initial efforts by Seppanen et al. [306] and Caulfield et al. [68] quantified the software overheads for fast NVM storage accesses. Woong Shin et al. advocated for shortening of the IO path by merging execution contexts and eliminating additional context switches in IO processing [314]. Ahmad et al. suggested to use interrupt coalescing to reduce the CPU load while processing interrupts in a virtualized environment [24]. Yang et al. recommended to use a synchronous IO completion path with polling with low-latency NVM devices [364]. Dong In Shin et al. proposed *dynamic polling*, where the polling interval is dynamically adjusted by monitoring the responsiveness of the NVM device [313]. Instead of using either exclusively polling (blocking the CPU) or blocking (high cost of context switches) IO, Wei et al. use speculation as a means to optimally use the processor for IO completion processing [355]. An application continues with its execution until it produces an externally visible side effect. At this time, the IO stack suspends its execution and processes IO completion. In this way, an application continues to do useful work without being held while IO requests are completed. Bjørling et al. looked into the scalability of the block layer of GNU/Linux [53]. They redesigned and optimized the block layer implementation in Linux for reduced contention on shared resources, increased parallelism, and scalability on multi-processor systems. In a more comprehensive work, Yu et al. implemented several of the aforementioned optimizations and analyzed their impact on the block IO performance of Linux [366]. They have implemented synchronous completion with IO polling, merging of discontinuous requests, optimization of IO scheduler, smart read-ahead logic, and avoiding lock contention in request queues. Others have looked into modeling SSD to reason about their performance [102, 103].

2.3 Distributed Storage

A part of this thesis deals with building a remote storage access stack using RDMA principles and operations. In this section, we provide the necessary background on classical distributed storage solutions, i.e., SAN based and NAS based, and comment on the RDMA usage of these solutions, why they have seen limited success, and the recent rejuvenated interest in RDMA networking.

2.3.1 Storage Area Network (SAN)

Storage area network (SAN) is a network that provides IO accesses to remote storage devices at a *block-level* granularity. A SAN can be an isolated specialized network for storage IO or can share an infrastructure network with other systems. The key benefit of the SAN technology comes from consolidating storage in a logically central administration from where storage capacities can be managed and provisioned to servers distributed across a whole data center. Once these servers get a remote storage device as a local block device over a SAN, they typically install a local client-side file system such as ext4 to storage and access files. Alternatively, in a virtualized environment, virtual machines can be given raw remote block devices for storage IO using SAN protocols.

One of the most popular block-level storage protocols for local storage access is Small Computer System Interface (SCSI). The SCSI protocol defines a set of commands to discover, configure, and access data from local storage devices such as spinning disks or solid-state drives (SSDs). Naturally, the popular and standardized SCSI command set can also be used for accessing remote storage. Two popular networking technologies that use the SCSI command set to access remote storage are Fiber Channel (FC) and Internet SCSI (iSCSI). We now provide more details about them.

Fiber Channel Protocol (FCP) was one of the early high-performance networked storage protocols that ran over Fiber Channel (FC) networks. The key aim of FC was at the easy management and configuration of storage devices while providing good performance. To achieve these goals, a single-purpose, low-overhead storage protocol, i.e., FCP, was designed. Remote storage devices were connected to a server using a dedicated FC host bus adapter (HBA). The HBA implements the fully offloaded FC network protocol in the adapter. This design ensured good performance with reliability and isolation. Modern FC infrastructure supports throughput speeds of 2, 4, 8 and 16-Gbits/sec. However, at the same time, FC technology is criticized for its expensive hardware, requiring dedicated network equipments, and specialized knowledge from administrators to manage FC networks. To relax the networking requirement, follow-up protocols such as Fiber Channel over Internet Protocol (FCoIP) (tunneling-based), Internet Fibre Channel Protocol (iFCP) (routing-based), and Fiber Channel over Ethernet (FCoE) are defined. These protocols allow packing FC frames into IP or Ethernet packets for transmission over commodity IP/Ethernet-based networks without requiring any expensive or dedicated networking equipment. However, due to network fragmentation (only 1500 bytes packets) and the lossy nature of IP/Ethernet-based networks, the performance of these protocols was slow. Recent developments in lossless Ethernet help in running loss-sensitive higher-level protocols such as FCP on top of Ethernet frames.

As an inexpensive alternative to FC, Internet Small Computer System Interface or iSCSI was developed. iSCSI is an Internet Protocol (IP) based storage networking standard that transfers SCSI commands over IP-based networks. From the start, it was designed to be compatible with IP/Ethernet-based networks and uses TCP to carry SCSI commands and data packets. iSCSI can be implemented in a software or a hardware device. Initial prototypes of the protocols were hampered by poor implementations and performance, but the gradual improvement of Ethernet speeds (from 1 to 10 to 40 and now 100 Gbits/sec) helped to narrow the performance gap between FC and iSCSI protocols. Apart from FC and iSCSI, other niche alternatives such as ATA (Advanced Technology Attachment) over Ethernet (AoE), Hyper SCSI (SCSI on Ethernet), and network block devices (NBDs) also exist. However, to date, their deployment in enterprise environment is limited.

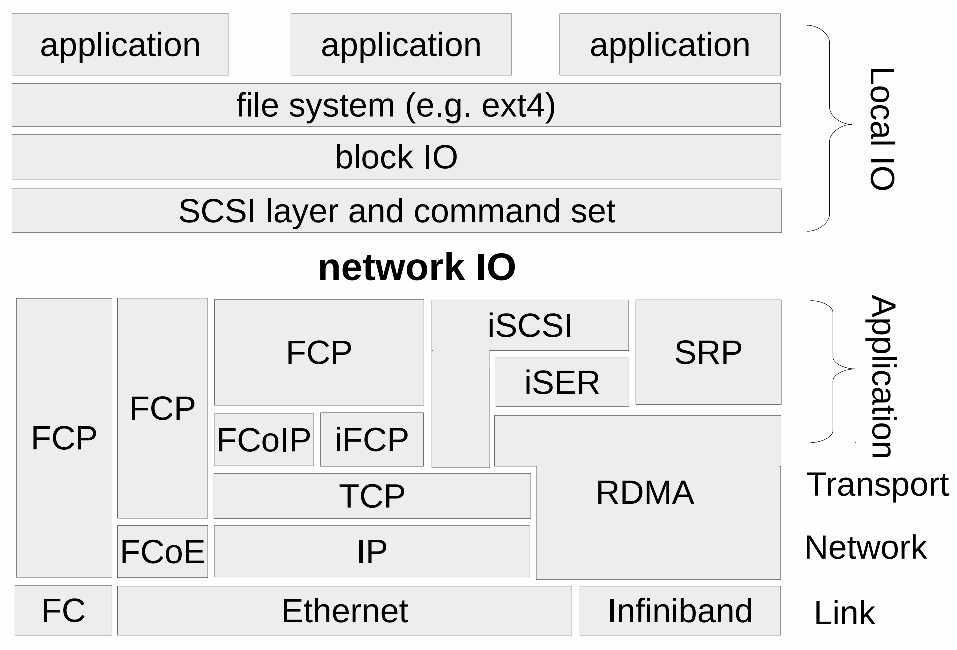


Figure 2.1: Traditional Fiber Channel Protocol (FCP) and various flavours of iSCSI SAN protocols in a layered architecture.

Multiple parallel efforts have also investigated integrating RDMA network operations in SAN protocols. Key goals for these works were to integrate transparently, reduce unnecessary data copies, leverage protocol offloading, and deliver higher bandwidth and lower latencies than what was possible with commodity interconnects such as Ethernet. iSCSI Enhancement for RDMA or iSER protocol is an extension of iSCSI transport to include RDMA networks. The iSER protocol integrates in the Data Mover (DM) architecture [73] of iSCSI which decouples data movement concerns from the rest of the iSCSI storage management protocol. Other DM protocols such as TCP-based, SCTP-based, or IB-based also exist. In contrast, SCSI RDMA Protocol (SRP) is a parallel effort to iSCSI to leverage RDMA operations for accessing remote SCSI storage devices. SRP natively packs SCSI commands (not iSCSI commands as done by iSER) in RDMA messages. SRP is a relatively new protocol and hence lacks substantial distributed device management and discovery services enjoyed by iSCSI and iSER deployments. Figure 2.1 shows the relations between the various flavours of iSCSI and FC protocols.

Modern NVM storage devices come with multiple host interfaces with NVMExpress (NVMe) being a standard interface for PCIe-attached NVM storage (Section 2.2.2). NVMe over Fabrics [240] is a recent effort to leverage RDMA for remote NVMe device accesses. It aims to improve performance by eliminating the heavy SCSI (and iSCSI) interface and leveraging light-weight, asynchronous, queue-based IO interface present in NVMe as well as in RDMA network interfaces. Other experimental RDMA-storage prototypes such as network-block device over Accelio messaging framework (ndbX) also exist [231]. However, Accelio's use of RDMA is limited to fast messaging.

2.3.2 Network-Attached Storage (NAS)

In contrast to the block-based access provided by SAN protocols, a Network-attached storage or NAS provides a native file-based access to data stored in remote storage devices. Many of the NAS protocol implementations integrate into the local operating system storage service to provide an illusion of a local file system containing files. For example, in Linux this integration takes place using its Virtual File System (VFS) layer. However, other solutions such as overlay-based file systems, user-space file system, etc., also exist.

Andrew file system (AFS) is a pioneering distributed file system design from CMU [159, 296]. Its design was influenced by its scale requirements and performance demand for networked clients. In order to reduce the network load and improve performance, AFS caches full files on the client-side local disks. When an AFS client opens a file, the server transfers the whole file to the client and the file is stored on the client's local disk. Any subsequent IO operations are performed locally on the cached copy of the file without any network IO. Upon closing the file, the content of the modified files are sent back to the server. All of these operations happen in a location transparent and independent way. The AFS server keeps track of client caches and actively notifies them in case the content of a cached file changes due to concurrent writes. This design was chosen on the basis that write conflicts were rare in the target deployment environment of AFS. Apart from having a simple design, AFS also supports other features such as multiple namespaces and administrative domains, Kerberos-based security, access-control lists, etc. However, AFS installations are complex to setup and administer. Due to the stateful nature of the AFS server, in an event of a crash (either on the server or a client), the recovery protocol could be complicated and could hamper performance during recovery.

Server Message Block (SMB) and its version, which is known as Common Internet File System (CIFS), is another NAS protocol that provides a shared file storage service [237]. SMB was originally developed by IBM around Microsoft MS-DOS and Windows operations systems, and the prototype was later taken over by Microsoft for further development. To let Windows clients access UNIX files and directories using the SMB protocol, Andrew Tridgell developed the Samba server software for UNIX servers. With the recent revision of 3.0 and the introduction of the SMB-Direct protocol, SMB can now use RDMA operations for network IO [238].

One of the most successful and popular NAS protocol is Network File System (NFS). NFS was originally developed at SUN Microsystems in 1984 [295]. NFS follows a server-client model where clients access data stored on a centralized NFS server using Remote Procedure

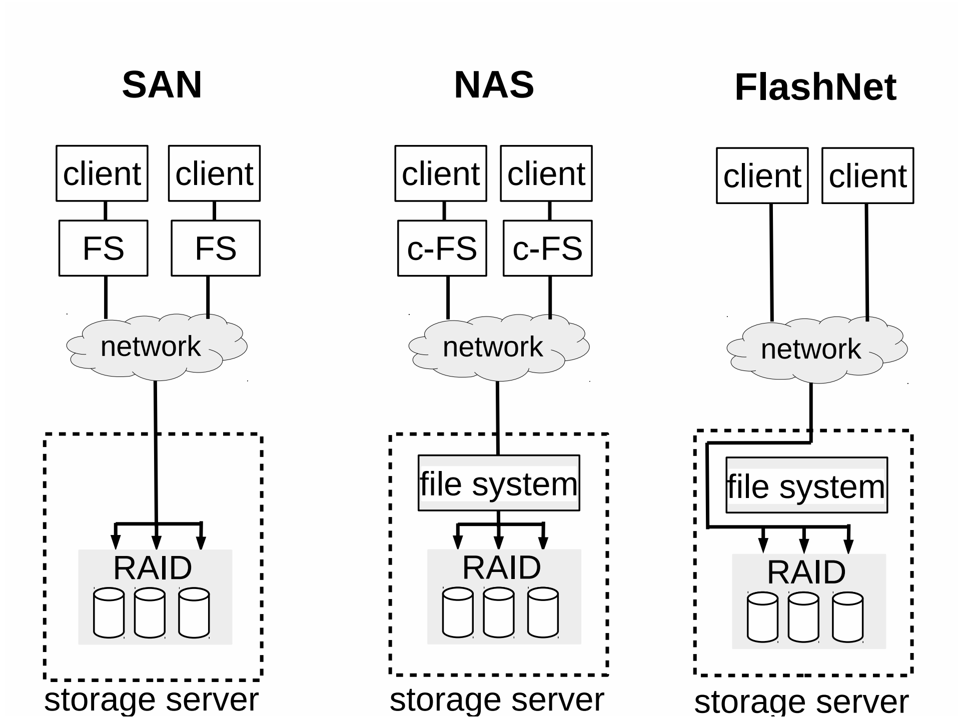


Figure 2.2: Comparison of data flows in SAN, NAS, and FlashNet approaches.

Calls (RPCs). The key design goal for NFS was a simple and fast server recovery after a crash. To achieve this goal, designers of NFS designed a (mostly) *stateless* protocol where the NFS server kept a small amount of distributed state about a client's file usage. The NFS protocol is defined in a RFC and is standardized [260]. Multiple implementations of the protocol exist with different flavours and enhancements. Client-side and server-side caching were introduced to increase performance. However, as NFS installations were deployed on an increasing scale, various issues with file locking, caching, ordering, and consistency semantics were discovered and gradually fixed in various revisions of the protocol. The latest incarnation of the protocol, v4, is influenced by the works from AFS and SMB and added support for stateful server operations for providing better performance and guarantees. Version v4.1 (pNFS) extended the protocol specification to include clustered, parallel server deployment. Due to its simplicity, the NFS remote file access protocol is also used as an access protocol for many high-performance commercial appliances e.g., Coho Data [96] and NetApp [256].

RDMA network IO is also used in NAS-based solutions to deliver high network performance to clients. Since version 4.0, NFS clients and servers can use RDMA transparently under the Sun RPC layer [67]. Other more general-purpose file systems such as PVFS, GPFS, and Luster, etc., also use RDMA, but in a limited capacity. RDMA network IO is usually retrofitted in these systems in the place of socket-based network IO. The general theme of this integration is to replace socket send/recv calls with RDMA send/recv calls. Data is either carried by send/receive calls or RDMA one-sided operations. The use of RDMA send/receive calls require explicit flow control management at the application level to ensure every received packet has a matching pre-posted receive buffer. And to use RDMA one-sided operations, these systems synchronize explicitly to fetch remote buffer credentials. Moreover, all these steps require applications to explicitly multiplex buffers, track IO requests, and manage connections – responsibilities not required by the socket API. All these requirements make "transparent-while-delivering-good-performance" integration of RDMA a tricky problem.

A more purpose-built RDMA-enabled file system is Direct Access FS (DAFS) [219, 220]. DSFA is implemented fully in user space rather than in the kernel, thus giving its applications the most benefits from by-passing the operating system. In a follow-up work, the authors also propose Optimistic RDMA (ORDMA) as an alternative to using RPC for improving performance of small messages by speculatively transferring data using one-sided RDMA operations [219]. DAFS delivered superb performance and clearly illustrated the performance benefits of RDMA-based network operations for a client-server file system design.

For comparison Figure 2.2, shows data flows in a SAN, NAS, and in a FlashNet setting. In SAN, data flows from a remote block devices to a client-side local file system, and is then eventually copied to application buffers. In a NAS setup, data flows from a server-side file system to a client-side local file system proxy (e.g., NFS client), and is then eventually copied to application buffers. FlashNet is a unified stack prototype that is presented in this thesis. In contrast to NAS and SAN based approaches, FlashNet uses RDMA principles to deliver data directly from remote storage devices to application buffers.

2.3.3 Commentary on the Usage of RDMA

Despite network performance being a key problem in distributed systems, the use of RDMA has only seen limited commercial success in SAN/NAS-type deployments so far. Multiple factors contributed to this state. This section summarizes the key observations and reasons behind why the use of RDMA has only seen limited success so far.

2.3.3.1 The OS-bypass Design that did not Bypass OS

SAN and NAS solutions are storage solutions which provide access to data stored in remote devices. Traditionally, storage as well as network IO are considered OS services which are implemented inside or with the help of a kernel for managing shared resources such the buffer cache. With the exception of DAFS, many RDMA-enabled solutions that we have presented so far implement RDMA networking inside the OS within the framework of a storage service (be it NAS- or SAN-based). Once data is brought into a host system using RDMA operations,

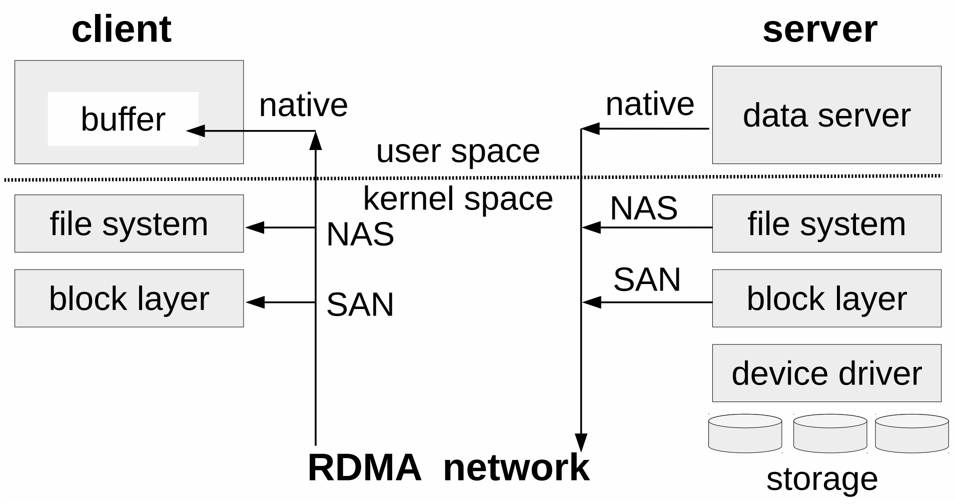


Figure 2.3: Data deliver targets for various storage/networking protocols that are capable of using RDMA networks.

it is processed and copied through multiple layers of abstractions inside the OS. Figure 2.3 shows this setup. For example, iSER and SRP protocols, which are SAN-based solutions that use RDMA, only deliver data efficiently to the block layer. Similarly, NAS-based solutions deliver data to the local client-side file system or file cache. The OS-bypassing architecture for user-space networking was designed to avoid this layered processing architecture with the goal of delivering data directly into the user buffer. All of these solutions fall short of this goal. The integration of RDMA underneath an OS-level storage service rendered one of the key principles of RDMA networking, i.e., OS-bypassing (or more generally *layer-bypassing*), useless. Consequently, only limited performance gains were delivered to end applications in such an architecture.

The FlashNet IO stack which is introduced in Chapter 5 uses native RDMA operations to transfer data from remote devices to local buffers while maintaining the benefits of having a file system service (similar to NAS) on the server side. The FlashNet configuration is shown in Figure 2.2c.

2.3.3.2 Limited API and Data Copies

The second major issue with the integration of RDMA has been its API. Since the BSD socket API for TCP was (is) the de-facto standard for networking for more than 30 years, RDMA networking was retrofitted in the networking layer of all applications. Consequently, many RDMA operations of vital importance to its performance such as apriori IO buffer registration and posting of receive buffers, were not possible with networking code designed for socket-based IO. There were two options to solve this issue. First, the RDMA networking stack could register IO buffers during the data transfer phase, and execute a costly registration operation and enforces additional metadata synchronization operations between a server and client. Or alternatively, it could perform network IO on pre-registered RDMA buffers and then copy data to application buffers in the end. Both options provide limited performance gains due to additional operations and overheads introduced in the end-to-end data transfer operation.

2.3.3.3 Marginal Performance Benefits

The third major issue behind the limited deployment of RDMA-enabled systems was its limited performance benefits in real world settings. Three factors contributed to this state. First, for long, the focus of the storage industry has been on cheaper, denser, and more reliable storage solutions. Hence, the raw storage performance improvements lagged significantly behind networks and CPUs. As the performance of most of the SAN/NAS-based solutions was dictated by the performance of spinning disks and not the network, putting fast RDMA networking did not deliver exceptional performance gains.

Second, due to the high disk IO latencies the key performance metric of any SAN/NAS solution was aggregate bandwidth. And with enough parallelism and large buffer sizes, many solutions, even those with an inefficient networking stack, managed to amortize networking overheads while delivering a good aggregate bandwidth. Hence, the high engineering cost of RDMA integration required more justification. Only recently with the availability of NVM, storage bandwidth and latencies became comparable to network's. Consequently, general interest in RDMA operations has revived.

Lastly, RDMA was touted to deliver superior performance than sockets by offloading and saving CPU cycles from network IO. However, the collective CPU performance (with multi cores) kept increasing and the networking stacks of modern OSes such as Linux managed to keep up with the network performance improvements and delivered the necessary bandwidth improvements. Consequently, the bandwidth gap between rigidly offloaded RDMA network and highly optimized software stacks were small.

2.3.3.4 General Complexity and Niche Hardware

A general problem with any offloaded solutions is its device API. The network offloading with RDMA was no exception. In his work, Mogual [244] criticises TCP offloading for being complex, buggy, and lacking a clear performance advantage. In a similar spirit, Magoutis [218] points out that most of the benefits of RDMA comes from direct data placement, which can be achieved without providing direct user-space access to networking hardware as done in user-space networking stacks. An unsupervised direct user-space access to share NIC resources entails a NIC to support right IO abstractions, security, isolation, QoS, and complex resource management — responsibilities which are traditionally done in the kernel. In essence, due to all aforementioned reasons, RDMA enabled hosts and networks only saw limited deployment in the early 2010s.

2.3.4 Recent Interest In RDMA

Since the late 2010s, RDMA and the prior research about high-performance networking from the 1990s have seen a rejuvenated interest from the commodity computing community. RDMA and related technologies were already pervasive in the high-performance computing (HPC) environment with a myriad of libraries, services, and abstractions being developed for HPC applications. As commodity computing became cheaper and more powerful, pioneering works from multiple groups in academia (e.g. Berkeley Network-Of-Workstations (NOW) project [28], BEOWULF cluster [324]) and commercial projects (e.g., Inktomi Corporation [126] and Google [100, 141]) illustrated the viability of high-performance, data-intensive workloads on commodity clusters as well. These workloads demand low latency and high bandwidth not just for storage, but also for in-memory computing. Factors such as cheap prices of DRAM storage and networking equipments, the emergence of Non-Volatile storage, and the relative maturity of RDMA devices with strong demands for high-performance computing on commodity clusters, etc., enabled a new breed of distributed systems. The seminal paper about a case to build a RAMCloud [264] from John Ousterhout and team played a key role in putting the focus back on networking performance [292].

Unlike previous efforts, which stopped RDMA integration too early in the stack (see Figure 2.3) and suffered performance losses, the next wave of integration efforts go deeper and closer to applications. These efforts hide the RDMA complexity and idiosyncrasies behind APIs of *infrastructure-level* services, and provide higher-level IO operations than just raw network or storage IO operations. Examples of these RDMA-enabled services include key-value stores [180, 183, 241, 326], distributed file systems [176], distributed data stores [262, 340], replication [369], databases [131], RPC service [327], and distributed computing [107, 255].

2.4 Conclusion

This chapter has provided an overview of the developments in the architecture, design, and implementations of modern networking and non-volatile storage devices and stacks. As evident from the discussion, there are no final words on the performance of these stacks and they will continue to evolve and improve in the foreseeable future. Furthermore, modern network and storage devices do not just deliver good results, but also offer many semantically rich IO operations. These developments force us to re-evaluate how we build distributed systems to provide services. In the next chapter, we focus on the Remote Direct Memory Access (RDMA) technology which provides the foundational abstraction for network and storage IO operations discussed throughout this thesis. The evolution, usage, and recent interest in RDMA are already briefly discussed in this chapter. The next chapter provides more detail on its basic design principles, abstractions, and supported operations. Based on that, Chapter 4 illustrates some striking resemblances in the development of high-performance storage and networking stacks and makes a case to unify them.

3 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) technology offers high-bandwidth and low-latency network operations for accessing data in remote memories. In this chapter we provide background information on RDMA technology, associated abstractions, networking operations, and application programming interface. Concepts and terminology presented in this chapter are referenced throughout the rest of the thesis.

3.1 Background

Remote Direct Memory Access or RDMA is a mechanism that enables an application to read or write remote memory locations without requiring any active participation from a remote application. It offers higher bandwidth and lower latencies than traditional socket-based networking in which both, local and remote applications, have to actively participate in network IO. A big part of this performance is achieved by bypassing local and remote OSes and CPUs when reading and writing remote memories. RDMA capabilities can be provided in a variety of ways. Specialized interconnects such as Crays Aries and Gemini networks, IBM's BlueGene networks, Fujitsu Tofu interconnect, etc., which can be found in modern supercomputers long supported RDMA capabilities in their end-host stack.

RDMA capabilities for commodity networks were discussed and supported by the Virtual Interface Architecture (VIA) specification. VIA was an industry effort from Intel, Compaq and Microsoft to standardize key ideas from extensive research done in the space of user-space networking with commodity networks. The first VIA specification was released in 1997 and multiple implementations followed soon, e.g., M-VIA [230], Berkeley-VIA [65], VIA on IBM RS/6000 SP network [38], etc.

VIA in its original form is a point-to-point connection-oriented protocol, which provides each application its own private, user-space mapped network endpoints. An endpoint (or a virtual interface (VI)) logically contains a set of DMA descriptor queues where the application can post prepared buffers for network operations such as send/receive-based messaging or one-sided RDMA operations. The kernel is only involved in setting up the network resources (buffer registration, VI allocation, connection establishment steps, etc.) but not in data transfer or network processing steps.

VIA was also supported by InfiniBand [232], an emerging standard (back then) which aimed to provide a unified interconnect for intra- and inter-system networking to eliminate interconnect bottlenecks in an end-to-end manner. The InfiniBand specification was open and hence, multiple vendors such as Mellanox, Voltaire (merged with Mellanox in 2010), QLogic (acquired by Intel), etc., implemented InfiniBand networking products where VIA thrived. With the success of InfiniBand, other commodity interconnects such as IP-based networks [156], and Ethernet [166], also picked up its specification. The end-host support and software stacks for these interconnects are consolidated under the Open Fabric Alliance (OFA). OFA provides OFA Enterprise Distribution (OFED) RDMA stack, which is the de-facto RDMA stack on Linux. The OFED API contains many features similar to the original VIA specification.

Although multiple specialized and commodity interconnects now support RDMA operations, the notion of RDMA should not be confused with the underlying interconnect used. All interconnects need end-host hardware and software support for implementing RDMA operations in a system. The work division between hardware and software to support RDMA for a particular interconnect and system architecture varies, and consequently, so does the end-host interface to applications. These are important factors in a particular implementation of an RDMA end-host stack and play a decisive role in determining the performance delivered. The follow-up discussion in this chapter and work presented in the rest of the thesis is set in a particular context of VIA-inspired RDMA interconnects and end-host interfaces. These VIA-inspired end-host stacks can trace their lineage back to academic projects such as Hamlyn [66, 358], UNet [352], SHRIMP [56], Application Device Channels [111], etc. In the following section we provide more details about the RDMA end-host networking stack in the context of the Linux/OFED stack.

3.2 Terminology

- **Traditional Networking Stack:** The traditional stack refers to the non-offloaded, classical networking stack implemented as an OS service inside *NIX operating systems such as GNU/Linux or OpenBSD. For this stack, the key application abstraction is a socket (a file descriptor) with send/receive (or read/write) calls to send or receive data. Unless explicitly stated, we refer to connection-oriented network transfers using the Transmission Control Protocol (TCP).

- **RDMA Interface:** An RDMA-based interface or RDMA interface refers to the queue-based, asynchronous, network communication interface used by applications to transfer data on *RDMA networks*. The abstractions and network operations associated with this interface are discussed in Sections 3.4 and 3.5.

- **RDMA Network/Transport:** An RDMA network or transport identifies one of the VIA-inspired commodity interconnects that are supported for the RDMA/Verbs end-host specification. The current generation of such transports includes InfiniBand [164], iWARP [156], and RoCEE [166]. Section 3.6 provides a brief overview of these RDMA networks.

- **RDMA Verbs and Verbs Interface:** *Verbs* represents a set of abstraction operations that define the semantics of application interaction with an RDMA-capable network controller (RNIC). The *verbs* interface actualizes these abstract operations for various RDMA transports.

- **RDMA or Verbs Provider:** An RDMA or a Verbs provider refers to the end-host entity that provides a complete implementation of the RDMA verbs interface. Though traditionally implemented in an offloaded manner on a NIC (e.g., Chelsio Terminator RNICs [76, 77]), a provider can also be implemented purely as a software entity inside an OS kernel (e.g., SoftiWARP [236, 339], or SoftRoCEE [15]).

- **Verbs Consumer or Applications:** A verbs consumer or an RDMA application refers to any entity outside the RDMA stack that uses its verbs interface for network IO. This entity can reside inside or outside (in the user space) of a kernel. An RDMA application within an OS kernel is called a kernel client.

- **RDMA End-Host Stack:** An RDMA provider together with the rest of the necessary hardware/software infrastructure is referred to as the RDMA end-host stack. RDMA end-host networking stack figures in this thesis are drawn for a hardware implementation of the RDMA provider. A software implementation would naturally involve the OS kernel in data copies, network, and API (verbs) processing as well.

- **RDMA Resources:** RDMA memory and communication resources that are allocated and used for data transfer operations are collectively referenced to as RDMA resources. These resources are discussed in detail in Section 3.4 and Section 3.7.

- **Local and Remote/Peer Hosts:** In this thesis, we always refer to communication within the scope of a connection-oriented model. In this model, a connection has two end-points or end-hosts. These two end points need to be connected explicitly with a connection establishment

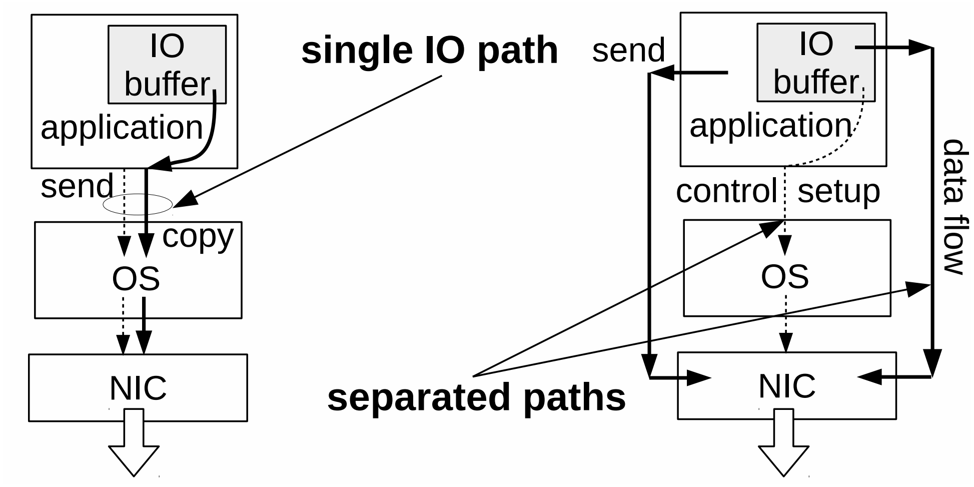


Figure 3.1: Illustration of a single-path and a separated path network architecture. The dotted lines represent the control transfer while the solid lines represent the data path.

procedure. Depending upon the narration, one side is referred to as the local and other as the remote or peer side.

- **RDMA Active and Passive Side:** The RDMA active side is the one end-point of the connection that actively initiates a network IO operation. The passive side only asynchronously waits for data or RDMA request arrival from the active side.

- **Source and Sink Buffers:** These buffers identify logical source and sink memory areas where data is transferred from and received into, respectively. The locations of these buffers change (either on local or peer host) depending upon the type of RDMA network operation used.

- **Buffer Registration or Preparation:** This term refers to the process of allocating, pinning, and registering a memory buffer (virtual or physical) with an RDMA provider. More details about this process are given in Section 3.4.2.

- **OFED Stack:** Open Fabric Enterprise Distribution (OFED) stack from Open Fabric Alliance (OFA) is the industry-standard, de-facto end-host RDMA stack implementation which is distributed with the GNU/Linux and Microsoft Windows operating system. For the work covered in this thesis, we use OFED RDMA stack.

3.3 The Data and Control Path Separation Principle

The key concept in RDMA-based network communication is the *path separation principle* that advocates to separate the slow *control path* from the fast *data path* of network IO operations. By separating these two, RDMA networks deliver high performance to applications by setting up all control resources between applications, operating system and network before the network IO processing starts. After the resource setup, applications communicate directly with the network controller for fast data transfers without requiring OS mediation for resource allocation and multiplexing. Consequently, the OS (and associated shared resource management overheads) is eliminated from the fast data path by design.

In contrast, the traditional networking stacks and operations only offer a single data and control intermingled IO path. Figure 3.1 illustrates this difference. In a single path architecture, the control transfer (which is triggered via a `send` call and includes resource allocation, preparation, translation, scheduling, etc.) is provided with the data as well. This data is copied and processed with the control path processing. The basic rationale behind this design lies in the architecture of early operating systems, which were designed to maximize the resource utilization by sharing systems resources. High resource utilization was necessary to offset the high price of computing as hardware was expensive. However, due to the demands from applications and general availability of cheaper commodity hardware, the idea of path separation principle and user-space networking (see Section 2.1.3) is now being revisited in the context of data center applications.

In a path-separated network architecture, sharing is eliminated by pre-allocating and exclusively assigning resources to applications on the control path before the IO processing begins. The slow control path entails allocation, control, and management of IO resources (e.g., connections, memory buffers, multiplexing mappings, IO channels, etc.) and extends all the way down to the NIC. These operations are executed with the help of a middleware entity, usually an operating system kernel, which mediates the allocation of resources, does appropriate security and limit checks, performs the necessary resource translations, and establishes the required mappings etc., for networking processing. In essence, the middleware creates an application-private (i.e., without sharing) view of the networking stack. These control operations are historically deemed slow due to their larger execution time (partially due to blocking/schedule-able nature of operations) than the network operations of the data path. On the data path, an application communicates directly with the network controller for network IO (e.g. sending data) by using previously established private communication channels. Data is also fetched separately from the control establishment after a network IO requested, is posted by the application.

3.4 Abstractions

In this section, we provide background information on the basic RDMA communication abstractions and resources. These resources, which are covered in the subsequent sections, are illustrated in the Figure 3.2.

3.4.1 Protection Domain (PD)

The key role of a protection domain is to ensure a secure access to RDMA resources. An application can create as many protection domains as it wants within a local RDMA provider. These protection domains are then used for the creation of RDMA IO resources such as memory regions (MRs) or RDMA queue pairs (QPs) (see the following subsections for their definitions). During RDMA IO processing, the local RDMA provider checks the protection domains of all involved resources. If an IO request accesses memory buffers which are not created within the protection domain as of the RDMA connection, the provider raises an error. Hence, conceptually, a protection domain is analogous to a process address space abstraction within modern operating systems.

3.4.2 Memory Regions (MRs) and Registration

Memory Regions (MRs) are application memory buffers that can be used as source or sink buffers in any RDMA operation. Memory areas, which are allocated using any standard memory allocation mechanism such as `malloc` or `mmap`, can be used as memory regions by *registering* them with the local RDMA provider. The registration process involves calling an RDMA end-host stack specific registration function with the virtual address, length, and access permissions of a memory area. On success, the function returns with an opaque 32-bit buffer identifier called *steering tag* or STag, generated by the local RDMA provider. The application uses this STag in network requests to identify IO buffers. All application buffers that are intended to be used for RDMA communication must first be registered with the local RDMA provider. However, prototypes, which relax this requirement also exist, e.g., UNet/MM [357] and Mellanox NICs [209]. These prototypes cache virtual-to-physical translations in the NIC and manage this cache with the help of the device driver and the OS. However, their APIs and implementations are not standard and their usage is imited.

The purpose of this registration process is three folds. First, during the registration process, the operating system performs the necessary access rights and limits checks about the memory consumption of the application. Second, the other necessary resources such as DRAM pages are allocated and installed into the page table of the process to ensure no page-faults during the network processing. The handling of page faults, which is a slow operation because it involves memory checks, allocation, and installation etc., is avoided on the data path. And lastly, associated DMA-descriptors are installed into a mapping table of the RDMA provider with a generated STag. The generated STag acts as a key into the memory buffer look-up table of the local RDMA provider to resolve the DMA buffers and steer data flows into/from them during network IO operations.

3.4.3 Event Channel

The RDMA end-host architecture follows an asynchronous event-driven architecture. An RDMA provider generates notifications for all network and connection related events. These notifications are then delivered to appropriate applications using an application-private event channel. In the OFED stack, these channels are implemented as a file descriptor where notifications are posted. Examples of these events include link up or down status of RDMA network, address and route resolution related events, connection establishment and termination, asynchronous error conditions, etc.

3.4.4 Queue Pairs (QPs)

A queue pair or QP represents the IO-request interface of a connection. It is equivalent to the socket abstraction in the traditional networking stack. A QP consists of a send queue (SQ) and a receive queue (RQ) where an application can post RX and TX IO requests (see Section 3.4.5). Internally, a QP is typically implemented as a shared memory region between an application and the RDMA provider. As with the other RDMA abstractions, QP is an application-private IO channel to the RDMA provider. An application can allocate and use multiple QPs concurrently. However, the actual number might be restricted by the security limits. Though the creation of QP is a slow control operation, posting of an IO request is a fast data path operation.

On the client side, an application creates a QP and invokes a `connect()` call to establish a connection to a server application. On the server side, the server applications creates a new QP while processing the corresponding "connect" event. The QP creation call takes the size of the send and receive queue as a parameter.

3.4.5 Work Requests (WRs)

A request to transmit or receive data is called a work request (WR). Send or receive WRs are posted on a QP from where they are picked up by the RDMA provider for processing. The representation of a posted WR in a send or receive queue is called a work queue element or a WQE. The *posting* of a WQE on a QP is a non-blocking operation. The IO processing notification for a WR is delivered asynchronously on a completion queue which will be discussed in the next subsection. An application can post multiple linked WRs in one posting to amortize posting overheads. The maximum number of outstanding send or receive WQEs depends upon

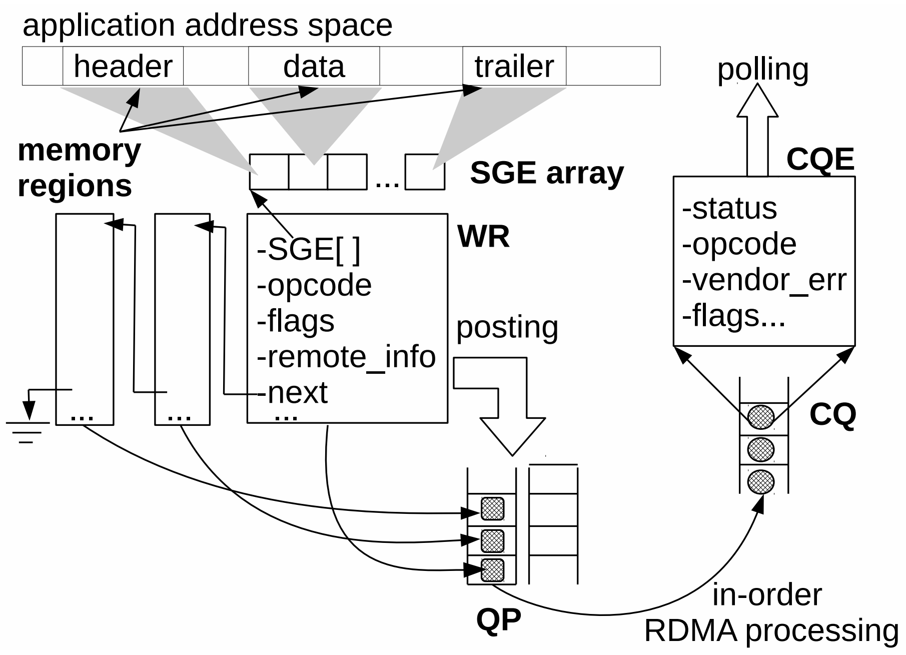


Figure 3.2: RDMA resources, namely memory regions, a queue pair (QP), work requests (WR), an scatter-gather element (SGE) array, a completion queue (CQ), and a completion queue element (CQE).

the size of the SQ or RQ, respectively, which were specified at the QP creation time.

A WQE typically consists of an opcode identifying the type of operation (Section 3.5), associated IO buffers and STags, the size of the operation, and the notification mechanism. The structure of a WQE supports scatter-gather elements (SGE) to identify local memory regions. Hence, IO buffers do not have to be contiguous in the virtual memory of an application. Having such capabilities support zero-copy data transmission operations where header, data, and trailer of a network operation can be built and transmitted from different buffers. Figure 3.2 shows a graphical view of SGEs, buffers, and WRs in a QP.

3.4.6 Completion Queue (CQ) and Channel

The IO processing notification delivery mechanism of RDMA has two associated abstractions with it. First is a notification channel, which is used by the RDMA provider to deliver "activity" notifications to applications. An activity might refer to the completion of one or more previously posted WRs. A completion channel is established between an application and the RDMA provider during the control setup by the operating system. Due to the operating system involvement in the management of completion channels, it is possible to share and aggregate IO notifications from multiple local RDMA providers into one channel. In essence, the completion channel is similar in the spirit to the event channels, but is for IO notifications.

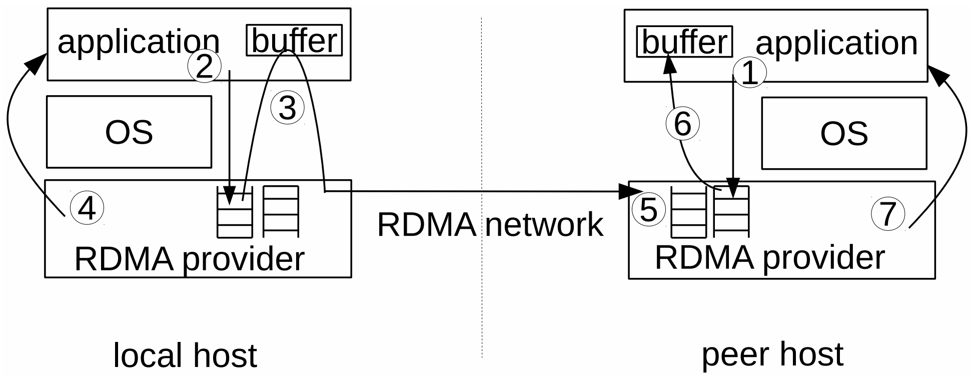


Figure 3.3: Illustration of a two-sided RDMA send/recv operation.

A second abstraction called Completion Queue (CQ) is used for delivering a complete report about completion of individual WQEs. For every WQE processed, the RDMA provider generates a Completion Queue Element or CQE. A CQE contains a full report about the processing of a WQE including its status, opcode, error code, vendor-specific information, QP information, application-provided identifier, and length of the processed IO, etc.

An application can get IO completion notifications in two manners. The first is a *blocking* mode where the application can block on the completion channel while waiting for a notification. Upon receiving the notification, the application is unblocked and it can then poll and *reap* CQEs from the CQ to check the status of WRs. The second mechanism is a direct, synchronous *polling* on the CQ without waiting for a notification first. Consequently, synchronous polling yields better network operational latencies at the expense of CPU cycles.

3.5 Network Operations

The traditional networking stack offers rendezvous-based network operations. In these operations, end-hosts operating system and applications of both connected hosts are involved in network processing. In contrast, RDMA-based networks offer multiple semantically rich network operations. These operations are typically classified in two categories, namely two-sided and one-sided operations. The numbers *two* and *one* represent how many applications are involved in the completion of a network operation. For example, one-sided RDMA operations can be executed completely by one initiating (or active) end-host and the two RDMA providers without involving the application from the remote host. RDMA providers of both end-hosts are always involved in the completion of an RDMA network operation. Figure 3.3 and Figure 3.4 illustrate these two modes.

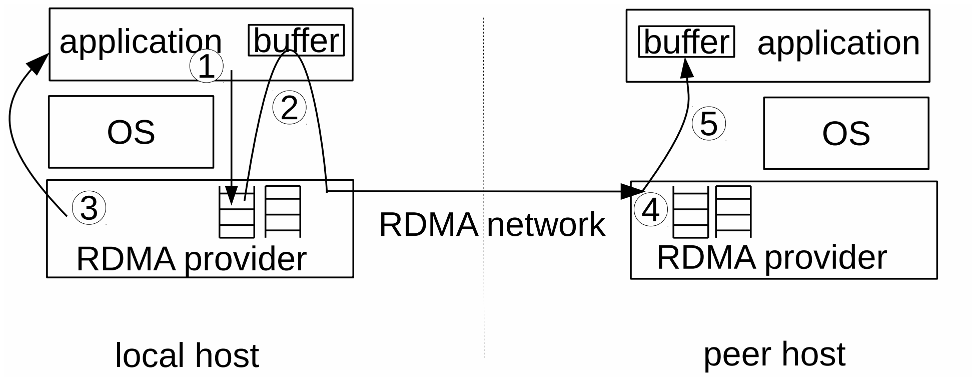


Figure 3.4: Illustration of a one-sided RDMA write operation.

3.5.1 Two-Sided Operations

Two-sided message-based send and receive network operations are analogous to the traditional stack's BSD `send` and `recv` operations. Figure 3.3 shows a high-level, step-by-step execution of a send and a receive operation. First, on the peer host the receiving application posts a receive WQ on the associated QP indicating where it wants to receive data (step 1). The receive WQ posting must happen before the transmitted data arrives at the peer host, otherwise the RDMA provider is free to drop data and generate an error event on the completion channel. Subsequently, the sender application posts a send WR (indicated by the operation opcode) on its QP (step 2). The local RDMA provider processes the send WQE and resolves the local source buffer from where the data is transmitted using a DMA operation (step 3). After send WQE processing, a corresponding CQE element is generated indicating the status of the send operation (step 4). When receiving the data, the peer RDMA provider extracts a receive WQE and processes it to resolve local buffer information (step 5). The extraction and processing of WQEs happen in a FIFO manner. Incoming data is then DMAed into the resolved buffer in a zero-copy manner (step 6), and a corresponding receive CQE is generated indicating the status of the receive WQE (step 7).

A typical application of two-sided send/recv operations on RDMA networks is a Remote Procedure Call (RPC) implementation. Using these operations, it is possible to deliver low per-request latency (10μsecs with iWARP) with a high non-pipelined aggregate throughput of 2–3 MOps/sec [327].

3.5.2 One-Sided Operations

One-sided RDMA operations only involve one active end-host in completion of an RDMA network request. Examples of these operations include RDMA read, write, and extended atomic operations. Figure 3.4 shows an example of an RDMA write operation from a local host to a buffer at a peer host without involving peer host's operating system or application. In a one-sided operation, the active side of the communication provides the complete information to execute the end-to-end data transfers in the posted WQE. To post an RDMA write request, an application not only specifies the local source data buffer (and STag), but also the remote sink data buffer and STag (step 1). Naturally, setting up an RDMA one-sided operation requires apriori communication (not covered by the RDMA/Verbs specification) between the local and peer applications to exchange buffer identifiers. The local RDMA provider then resolves and transmits data from the local buffer together with the remote/sink buffer identification STag (step 2). After processing the write WQE, a corresponding CQE element is generated indicating success or failure of the operation (step 3). The peer RDMA provider resolves the sink buffer for incoming data by matching the passed STag with its local registration table (step 4). Hence, no previously posted receive WQE is required to complete the write operation. After a successful buffer resolution, data is DMA'ed into the application buffer (step 5). A selective form of local notification can be generated about the processing of a one-sided RDMA operation on the peer side, but not all RDMA transports support that. The processing order semantics (completion order on a single QP) of these operations are the same as the two-sided operations.

One-sided RDMA operations are useful for efficient large data transfers. Due to no peer end-host involvement in processing of the RDMA network request, the performance of one-sided RDMA operations are generally better than the two-sided operations. However, the exact performance gap is protocol and implementation specific.

3.6 RDMA-Network Implementations

In this section, we provide a brief overview of three standardized implementations of the RDMA/Verbs specification. Multiple other proprietary implementations of the specification and in general of the RDMA operation also exist.

3.6.1 InfiniBand (IB)

In the late 1990s, multiple groups were competing to design a standardized system area network (SAN). The key requirement for such design was to be able to carry data with a very high performance from peripherals to peripherals. These designs were considered as PCI successors and aimed to replace the system IO bus. The InfiniBand specification emerged after the merging of two competing standards, Future I/O by Compaq, HP, and IBM and Next Generation I/O developed by Intel, Microsoft, and Sun Microsystems.

The InfiniBand architecture specification [165] details out the complete networking stack including fabric, transport, protocol, and the API to applications. It provides both two-sided messaging and one-sided RDMA operations. InfiniBand's API supported VIA interface and operations. Originally developed for dedicated High-Performance Computing (HPC) environments, today InfiniBand is used in shared data center computing environments as well.

A side-effect of the complete stack design is that the InfiniBand approach is not compatible with the popular commodity Ethernet/IP based networking infrastructure. InfiniBand has its own network, switch, and connection management logic which differs significantly from the IP-based networks. Despite these limitations, implementations of InfiniBand specification have provided high bandwidths and ultra low latencies for network IO over the years.

3.6.2 Internet Wide Area RDMA Protocol (iWARP)

To bring advantages of RDMA-based data transfer operations to IP-based networking, Internet Wide Area RDMA Protocol (iWARP) was proposed. The initial idea was seeded and prototyped by Buonadonna and Culler who combined the InfiniBand specification with IP-based networking [64]. They proposed Queue Pair IP where they implemented RDMA network operations on top of a TCP, UDP and IPv6 protocols, and demonstrated the viability of the iWARP approach.

The iWARP protocol runs on top of any reliable IP-based network protocol such as TCP or SCTP and is typically deployed on the TCP/IP stack. The on-wire packet format of the protocol is the same as any normal TCP/IP packet's. Hence, the networking infrastructure remains unchanged and decades of networking experience with the TCP/IP stack can be used to manage and provision networking infrastructure.

The iWARP protocol specification is covered in a series of RFCs [46, 95, 185, 195, 277, 283, 307, 308] and defines a direct data placement protocol (DDP) that can be use to encode RDMA operations as defined in the RDMA protocol (RDMAP) and associated metadata into the network headers. This metadata is then used to resolve source and sink buffers to perform a zero-copy, direct data placement into application buffers.

Initial performance concerns of the iWARP protocol stemmed from the fact that it involved integrated TCP/IP processing as well. Previous studies have already lamented the performance and benefits of TCP offload engines [128, 244]. However, the RDMA API helped to address the shortcomings of TCP offload engines and presented a different operational environment and network semantics than the socket API. Consequently, later prototypes of iWARP devices provided good performance and routinely managed to narrow the wide performance gap between them and the state-of-the-art InfiniBand performance. In his work, Recio presents a good comparison between InfiniBand and iWARP performance in the context of server IO and consolidation [284].

3.6.3 RDMA over Converged-Enhanced Ethernet(RoCEE)

RDMA over Converged-Enhanced Ethernet (RoCEE, pronounced “rocky”) is the latest entry in the list of networks that support RDMA operations. The key argument in the development of RoCEE has been the simplicity by collapsing layers of protocols while preserving the legacy Ethernet infrastructure. At the same time, RoCEE avoids complexity and resource requirements of the TCP flow and reliability control management in a large environment. RoCEE runs RDMA operations directly over raw Ethernet frames thus avoiding complex layered processing and implementation of networking and transport protocols. However, in order to cope with the reliable, in-order packet delivery, RoCEE needs reliable Ethernet extensions defined by the Data Center Bridging (DCB) Task Group [332]. Due to its collapsed networking stack implementation, RoCEE can deliver competitive or better performance than iWARP [234]. However, it has been criticized for its need for new Ethernet infrastructure that supports DCB extensions, the lack of routing infrastructure, and the absence of systems software to support it. The later v2 specification of RoCEE added support for routing.

3.6.4 Remarks

While providing almost semantically similar RDMA network operations, the three flavours of RDMA-networks differ in the performance delivered to applications. However, to the best of the author’s knowledge, no peer-reviewed publications exist that can conclusively establish design superiority of one stack over another. Performances of these stacks are highly dependent on their implementations, physical link speeds, and switching mechanisms.

For this thesis work we use iWARP-based networking infrastructure, though the work presented in this thesis is not specific or restricted to iWARP-based networks only.

3.7 Programming with the RDMA Separation Philosophy

In this section, we give a small example of an RDMA server client to illustrate how the separation principle is put into practice. The example shows how to write a server-client RDMA program that performs a send/recv operation. The example is developed using the OFED RDMA stack [16]. Whereever required, we provide more detail about the programming abstraction and interface. Comprehensive error checks are omitted for the sake of brevity. A more detailed programming-oriented discussion about these operations and other RDMA-related resources can be found in the RDMA-Aware Networks Programming User Manual [233].

3.7.1 Steps on the Client

As pointed out earlier, the end-host RDMA stack is an asynchronous event-based system. Events and requests for notifications are found everywhere in the code. And as a general rule of thumb, all explicit (in application knowledge) and implicit (done by OFED) resource allocations require corresponding free calls. A good example of an implicit resource allocation is a connection management event notification which is passed as a `struct rdma_cm_event` object. To deliver a notification, the OFED stack allocates a buffer to hold the event object which must be free explicitly by acknowledging the event by calling `rdma_ack_cm_event`.

An RDMA client takes the following steps in order to establish an RDMA connection on the reliable iWARP transport.

① Allocate an event channel to receive connection and network management related events.

```
struct rdma_cm_channel *cm_channel =
    rdma_create_event_channel();
```

② Create a connection management ID. A connection ID or `cmid` is an OFED abstraction that is used to hide connection management issues related to various underlying transports (IB, iWARP, or RoCEE). The `rdma_create_id()` takes an event channel, `cmid`, a context pointer, and a transport identifier. `RDMA_PS_TCP` identifies iWARP as the RDMA-network/transport provider.

```
struct rdma_cm_id *client_id = NULL;
int ret = rdma_create_id(cm_channel, &client_id, NULL,
    RDMA_PS_TCP);
```

③ Resolve address and route to a listening server IP and port. Like the traditional networking stack, the OFED RDMA implementation also takes a `struct sockaddr_in` to provide IP and port information.

```
struct socaddr_in s_addr;
s_addr.sin_family = AF_INET;
s_addr.sin_addr.s_addr = server_addr; /* provide server IP */
s_addr.sin_port = server_port; /* provide serve port */
ret = rdma_resolve_addr(client_id, NULL, (struct sockaddr*)
    &s_addr, 1000);
struct rdma_cm_event *cm_event = NULL;
ret = rdma_get_cm_event(cm_channel, &cm_event);
if (cm_event->status != 0 ||
  cm_event->event != RDMA_CM_EVENT_ADDR_RESOLVED) {
  printf("Failed to resolve the address, %d \n", -errno);
  exit(-errno);
}
```

```
ret = rdma_ack_cm_event(cm_event); /* ack the event */
ret = rdma_resolve_route(client_id, 1000); /* resolve route */
ret = rdma_get_cm_event(cm_channel, &cm_event);
if (cm_event->status ||
  cm_event->event != RDMA_CM_EVENT_ROUTE_RESOLVED) {
  printf("Failed to resolve the route, %d \n", -errno);
  exit(-errno);
}
ret = rdma_ack_cm_event(cm_event); /* ack the event */
```

④ Create RDMA resources — a protection domain (PD), a completion queue, and a queue pair (QP). The completion queue creation function (`ibv_create_cq`) takes a maximum number of elements on the queue, the completion channel, and a context as its parameter. Similarly, QP creation (`rdma_create_qp`) also takes the maximum number of send and receive queue elements as parameters.

```
struct ibv_pd *pd = ibv_alloc_pd(client_id->verbs);
struct ibv_comp_channel *comp_channel = NULL;
comp_channel = ibv_create_comp_channel(client_id->verbs);
struct ibv_cq *cq = NULL
cq = ibv_create_cq(client_id->verbs, 1, NULL,
          comp_channel, 0);
ret = ibv_req_notify_cq(cq, 0); /* request notification */
struct ibv_qp_init_attr qp_init_attr;
qp_init_attr.cap.max_recv_sge = 1; /* max recv SGE */
qp_init_attr.cap.max_recv_wr = 1; /* max recv WR */
qp_init_attr.cap.max_send_sge = 1; /* max send SGE */
qp_init_attr.cap.max_send_wr = 1; /* max send WR */
qp_init_attr.qp_type = IBV_QPT_RC; /* reliable transport */
qp_init_attr.recv_cq = cq; /* CQ for recv notifications */
qp_init_attr.send_cq = cq; /* CQ for send notifications */
ret = rdma_create_qp(client_id, pd, &qp_init_attr);
struct ibv_qp *qp = client_id->qp;
```

⑤ Connect to the server. This step generates a CONNECT event on the server side (see the next section, step 4).

```
struct rdma_conn_parm conn_param;
conn_param.initiator_depth = 1;
conn_param.responder_resources = 1;
conn_param.retry_count = 3;
ret = rdma_connect(cm_conn_id, &conn_param);
ret = rdma_get_cm_event(cm_channel, &cm_event);
if (cm_event->status != 0 ||
  cm_event->event != RDMA_CM_EVENT_ESTABLISHED) {
    printf("Failed to connect, %d \n", -errno);
  exit(-errno);
}
ret = rdma_ack_cm_event(cm_event); /* ack the event */
```

⑥ Prepare and register a data buffer with the RDMA provider (saved in the `verbs` pointer).

```
struct ibv_pd *pd = ibv_alloc_pd(client_id->verbs);
char *buf = malloc(1024); /* 1kB buffer */
struct ibv_mr *mr = ibv_reg_mr(pd, buf, (1024),
    IBV_ACCESS_LOCAL_WRITE);
```

⑦ Prepare and post a send WR on the QP. The `bad_send_wr` array is used to hold synchronously (failed during the posting) failed send WR(s). A send WR can be of type, send, read, write, or any other RDMA operation.

```
struct ibv_send_wr send_wr, *bad_send_wr = NULL;
struct ibv_sge send_sge;
send_sge.addr = mr->address; /* addr */
send_sge.length = mr->length; /* length */
send_sge.lkey = mr->lkey; /* STag */
/* wr */
send_wr.sg_list = &send_sge;
send_wr.num_sge = 1; /* number of SGEs */
ret = ibv_post_send(qp, &send_wr, &bad_send_wr);
```

⑧ Wait for completion of the posted send WR.

```
struct ibv_wc wc;
ret = ibv_get_cq_event(comp_channel, &cq, NULL);
ret = ibv_req_notify_cq(cq, 0);
ret = ibv_poll_cq(cq, 1, &wc); /* poll for 1 WC */
if (wc.status == IBV_WC_SUCCESS &&
      wc.opcode == IBV_WC_SEND) {
    /* ack the event */
    ibv_ack_cq_events(cq, 1); /* 1 event */
}
```

⑨ Actively call disconnect from the server.

```
ret = rdma_disconnect(cm_conn_id);
ret = rdma_get_cm_event(cm_channel, &cm_event);
if (cm_event->status == 0 &&
  cm_event->event == RDMA_CM_EVENT_DISCONNECTED) {
  printf("Disconnect successful \n");
  /* clean up local RDMA resources */
  ...
}
/* ack the event */
ret = rdma_ack_cm_event(cm_event);
```

On the client side, Steps 1–6 and 9 make the control path and Steps 7 and 8 make the fast data path.

3.7.2 Steps on the Server

A server takes the following steps in order to receive data from an incoming RDMA connection using the reliable iWARP RDMA transport.

① In a similar manner to a client, allocate an event channel in order to receive connection and network management related events. On this channel, create a listening connection management ID (cmid).

```c
struct rdma_cm_channel *cm_channel =
    rdma_create_event_channel();
struct struct rdma_cm_id *server_id = NULL;
int ret = rdma_create_id(cm_channel, &server_id, NULL,
    RDMA_PS_TCP);
```

② Bind and listen on a given (IP, port) for any incoming connection.

```c
struct sockaddr_in s_addr;
s_addr.sin_family = AF_INET;
s_addr.sin_addr.s_addr = INADDR_ANY;
s_addr.sin_port = server_port;
ret = rdma_bind_addr(server_id, (struct sockaddr*) &s_addr);
ret = rdma_listen(server_id, 1); /* backlog = 1 */
```

③ Wait until a new connect request arrives on the even channel, and then retrieve the new `cmid` of the client. The wait is done in the `rdma_get_cm_event` which is a blocking call.

```c
struct rdma_cm_event *cm_event = NULL;
ret = rdma_get_cm_event(cm_channel, &cm_event);
if (cm_event->status != 0 ||
  cm_event->event != RDMA_CM_EVENT_CONNECT_REQUEST) {
  printf("Failed on incoming connect, %d \n", -errno);
  exit(-errno);
}
struct rdma_cm_id *client_id = cm_event->id;
ret = rdma_ack_cm_event(cm_event); /* ACK the event */
```

④ Create a new protection domain, and allocate and register a new RDMA receive buffer with the RDMA provider (saved in the `verbs` pointer).

```c
 struct ibv_pd *pd = ibv_alloc_pd(client_id->verbs);
 char *buf = malloc(1024); /* 1kB buffer */
 struct ibv_mr *mr = ibv_reg_mr(pd, buf, (1024),
    IBV_ACCESS_LOCAL_WRITE);
```

⑤ Create a completion channel, a completion queue, and a queue pair.

```
struct ibv_comp_channel *comp_channel =
      ibv_create_comp_channel(client_id->verbs);
struct ibv_cq *cq = ibv_create_cq(client_id->verbs, 1,
      NULL, comp_channel, 0);
ret = ibv_req_notify_cq(cq, 0); /* request notification */
struct ibv_qp_init_attr qp_init_attr;
qp_init_attr.cap.max_recv_sge = 1; /* max recv SGE */
qp_init_attr.cap.max_recv_wr = 1; /* max recv WR */
qp_init_attr.cap.max_send_sge = 1; /* max send SGE */
qp_init_attr.cap.max_send_wr = 1; /* max send WR */
qp_init_attr.qp_type = IBV_QPT_RC; /* reliable transport */
qp_init_attr.recv_cq = cq; /* CQ for recv notifications */
qp_init_attr.send_cq = cq; /* CQ for send notifications */
ret = rdma_create_qp(client_id, pd, &qp_init_attr);
  struct ibv_qp *qp = client_id->qp; /* save QP pointer */
```

⑥ Prepare and post a receive WQE on the QP. This posting is done before accepting the connection to ensure that the incoming data always finds a pre-posted receive WR on the receive queue. The `bad_recv_wr` array is used to hold synchronously (failed during the posting) failed WR(s).

```
struct ibv_recv_wr recv_wr, *bad_recv_wr = NULL;
struct ibv_sge recv_sge;
recv_sge.addr = mr->address; /* addr */
recv_sge.length = mr->length; /* length */
recv_sge.lkey = mr->lkey; /* STag */
/* wr */
recv_wr.sg_list = &recv_sge;
recv_wr.num_sge = 1; /* number of SGEs */
ret = ibv_post_recv(qp, &recv_wr, &bad_recv_wr);
```

⑦ Accept the incoming connection and wait for a connection ESTABLISH event.

```
struct rdma_conn_param conn_param;
conn_param.initiator_depth = 1;
conn_param.responder_resources = 1; /* depth of IO queue*/
ret = rdma_accept(client_id, &conn_param);
ret = rdma_get_cm_event(cm_channel, &cm_event);
if (cm_vent->status !=0 ||
  cm_event->event != RDMA_CM_EVENT_ESTABLISHED) {
  printf("Failed to accept the connection, %d \n", -errno);
  exit(-errno);
}
/* ack event */
ret = rdma_ack_cm_event(cm_event);
}
```

⑧ Wait for a work completion (WC) notification for the recv WQE. A `struct ibv_wc` element contains all the information from a CQE.

```
struct ibv_wc wc;
ret = ibv_get_cq_event(comp_channel, &cq, NULL);
ret = ibv_req_notify_cq(cq, 0);
ret = ibv_poll_cq(cq, 1, &wc); /* poll for 1 WC */
if (wc.status == IBV_WC_SUCCESS &&
  wc.opcode == IBV_WC_RECV) {
  /* ack the event */
  ibv_ack_cq_events(cq, 1); /* 1 event */
}
```

⑨ Wait for a connection disconnect event.

```
ret = rdma_get_cm_event(cm_channel, &cm_event);
if (cm_event->status == 0 &&
  cm_event->event == RDMA_CM_EVENT_DISCONNECTED) {
  /* disconnect received, destrory all resources */
  ...
  /* ack the event */
  ret = rdma_ack_cm_event(cm_event);
}
```

Steps 1–5, 7, and 9 are the control preparation steps while steps 6 (posting) and 8 (WC retrieval) constitute the fast data path steps.

3.8 RDMA Performance Potential and Pitfalls

Despite recent wide-spread attention, translating the advantages of RDMA based network IO into application-level performance is challenging. In this section, we describe our experience and issues that we encountered while developing high-performance applications for RDMA capable network controllers (RNICs). Due to the offloaded nature of complete packet processing, many (un)related factors beyond the control of the systems software (e.g., the operating system) can significantly influence the performance gains for the end-application.

3.8.1 Motivation and Key Findings

To access the potential application-level performance gains when using RDMA, we conducted a simple request-response experiment for data transfer in a server-client configuration. For every client request, the server *prepares* a response buffer and sends it out on the network back to the client. We use RDMA and the traditional TCP stack in Linux for data transfers and compare their performances. Figure 3.5 shows (raw numbers in Table 3.1) the performance gains (seen by the client as improved request completion time) on the y-axis for different sizes

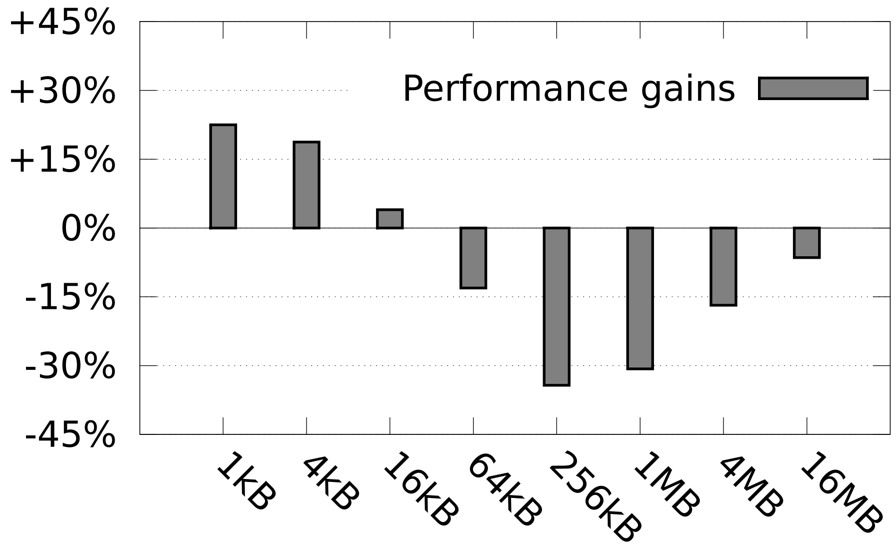


Figure 3.5: Potential performance gains of using RDMA network IO in comparison to the traditional BSD socket/TCP stack.

	1kB	**4kB**	**16kB**	**64kB**	**256kB**	**1MB**	**4MB**	**16MB**
TCP/sockets	40	48	75	153	350	1,133	4,781	20,877
RDMA send/recv	31	39	72	173	470	1,481	5,586	22,222
Gains in %	**+22.5**	**+18.7**	**+4.0**	**-13.0**	**-34.2**	**-30.7**	**-16.8**	**-6.4**

Table 3.1: Raw data for Figure 3.5. All numbers are in μsecs, representing the time it took to transfer a test buffer. The difference is calculated with respect to the baseline performance of TCP/socket's performance.

of the response buffer (on the x-axis). Positive numbers on the y-axis represent performance gains for the end-application using RDMA. Our experiment suggests:

(a) Networked applications can even lose performance when using network accelerators in particular circumstances. The performance implications of complex interactions among sophisticated CPU cores, last-level caches, and low-latency network controllers are highly machine specific and are hard to predict. For the same application, running on different generations of CPUs and NICs, one can observe a wide-range of performance fluctuations, including performance loss.

(b) Modern network latencies are getting closer and become comparable to architectural overheads. The overhead of coherence maintenance, cache misses, DRAM access latencies, and CPU stalls significantly influence the performance of an end-application operating in a low-latency network environment. Although the exact overhead is workload specific, it is affected by a number of characteristics such as buffer size, access pattern, etc.

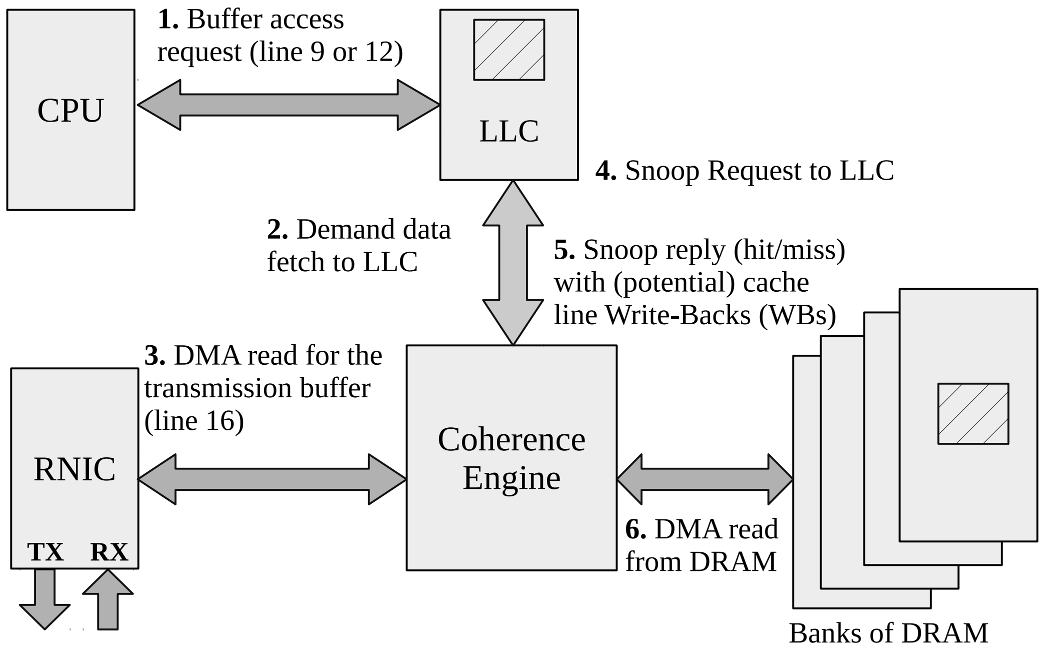

CPU
1. Buffer access request (line 9 or 12)
LLC
4. Snoop Request to LLC
2. Demand data fetch to LLC
5. Snoop reply (hit/miss) with (potential) cache line Write-Backs (WBs)
3. DMA read for the transmission buffer (line 16)
RNIC
TX
RX
Coherence Engine
6. DMA read from DRAM
Banks of DRAM

Figure 3.6: Interaction sequence among the CPU core, LLC, DMA access, and Coherence Engine. Line numbers refer to the code listing in Figure 3.7.

3.8.2 Benchmark Application and Experimental Setup

To understand our counter-intuitive findings where we observe performance degradation when using offloaded RDMA network processing, we start by designing a controlled request-response experiment between the server and the client as outlined in the previous section. Our analysis reveals that application-level latencies seen by the client are dominated by the buffer *preparation* step at the server. Hence, we further investigate the interaction among various entities involved in the buffer preparation and transmission steps, namely CPU, last level cache (LLC), and DMA access to DRAM. Figure 3.6 illustrates the sequence of interaction among the entities on an IO coherent architecture such as `x86`.

In our benchmark, the client constantly sends a request to the server in a tight loop without any pipelining. Upon receiving the request, the server prepares a buffer and transmits data in the buffer back to the client. The size of the buffer is variable. In our controlled setup, the *preparation* is a simple scan operation on the buffer. In a real-world application, the preparation step can involve more complex operations such as reading data from a persistent storage and then copying it into the buffer. Figure 3.7 shows the code which we implement within the `netperf` benchmark framework [9].

We now explain the preparation step in greater detail. Different buffer preparation configurations give us the flexibility to analyze cache and snoop protocols in a controlled environment

```
1 char dummy_buff[BUF_SZ], tx_buff[BUF_SZ];
2 /* Until timeout, keep receiving requests */
3 while(!time_out){
4   /* Receive the request from the client */
5   recv_request();
6   for(i = 0; i < BUF_SZ; i += CACHE_LINE_SZ){
7 #if SCAN_MODE == TOUCH
8     /* scan the transmission buffer */
9     scan(tx_buff[i]);
10 #elif SCAN_MODE == NO_TOUCH
11     /* else, scan the dummy buffer */
12     scan(dummy_buff[i]);
13 #endif
14   }
15 }
16 /* always send the transmission buffer */
17 send_buffer(tx_buff, BUF_SZ);
18 }
```

Figure 3.7: Server-side execution logic.

	NoTouch	**Touch**
Write Scan	Modified cache lines (M) from the dummy buffer.	Modified cache lines (M) from the transmission buffer.
Read Scan	Exclusive cache lines (E) from the dummy buffer.	Exclusive cache lines (E) from the transmission buffer.

Table 3.2: Content of last-level cache depending on the mode and the scan type. Modified(M) and Exclusive(E) cache line status represent the MESIF protocol states.

while keeping a uniform CPU load. On the server side, the *preparation* step has two modes: **Touch** and **NoTouch**. In the Touch mode, data in a transmission buffer is scanned using a `for` loop. In the NoTouch mode, a similar scan is done on a dummy buffer. The two buffers, transmission and dummy, are identical, but only the transmission buffer is transmitted on the network (see lines 15-16 in Figure 3.7).

Furthermore, the scan can be of two types: **Read Scan** or **Write Scan**. A Write Scan emulates a reader-writer sharing scenario, where the CPU writes and the network controller reads the buffer. A Read Scan represents a read-read sharing of the buffer. The scan access on the buffers (either transmission or dummy) brings the associated cache lines into the LLC. To maintain the IO coherence, transmission of the transmission buffer generates snoop requests for LLC. Table 3.2 summarizes the LLC content for different combinations of the modes and the scan types.

	Intel Xeon E7520
CPU cores	4×1.8GHz
QPI speed	4.8 GT/sec
L1 cache	64kB, 2.1nsec, 8-ways associativity
L2 cache	256kB, 5.3nsec, 8-ways associativity
LLC	18MB, 22.7nsec, 24-ways associativity
LLC type	Inclusive of L1 and L2 caches
Cache line size	64 Bytes
DRAM latency	131nsec
Prefetching	Next-line Prefetcher, enabled

Table 3.3: Architectural properties and configuration of Intel Nehalem-EX Xeon E7520 CPU.

3.8.2.1 Experiment Methodology and Hardware

We measure the single request completion time (the time between issuing a request and receiving the complete response buffer) at the client as the key performance metric. We use two network transport implementations for the buffer transmission - unaccelerated Linux in-kernel TCP/IP and an accelerated RDMA stack. The Linux stack runs on the host CPU together with the benchmark application. We calculate performance gains by comparing the request serving time between the two stacks. TCP performance is measured under a similar setup by using a modified `TCP_RR` test from the `netperf` test suit. The previously shown figure 3.5 compares the performance of the RDMA accelerated stack and the in-kernel TCP/IP stack under the Touch/Write Scan configuration for different response buffer sizes.

We perform our experiments on two identical IBM system x3690 X5 machines containing the Intel X58 chipset with Intel Xeon Nehalem-EX E7520 CPUs. Table 3.3 summarizes the architectural parameters for the CPU. Chelsio Terminator4 (T4) RDMA-capable Network Interface Controllers (RNIC) are used for RDMA network IO on the 10Gbps Ethernet. However, we repeated our experiments with Intel NetEffect network accelerator adapters and found no significant deviations in our findings. From this we concluded that our observations were not an anomaly of a particular RNIC implementation. We used Linux `perf` [12] measurement framework to measure the global coherence events as documented in the Intel manual [170]. Linux kernel version `3.7.0` is used in all experiments.

All experiments last 60 seconds, and are repeated three times. We omit reporting variance because the reported performance numbers have less than 5% standard deviation between the three runs. To avoid any multi-core coherence interference, all cores except Core0 are switched off. Core0 and RNIC are the only two entities in the system sharing the access to the DRAM. However, we have verified our results in presence of other cores and found no deviations in our findings. CPU pre-fetching is enabled for all experiments except for those presented in

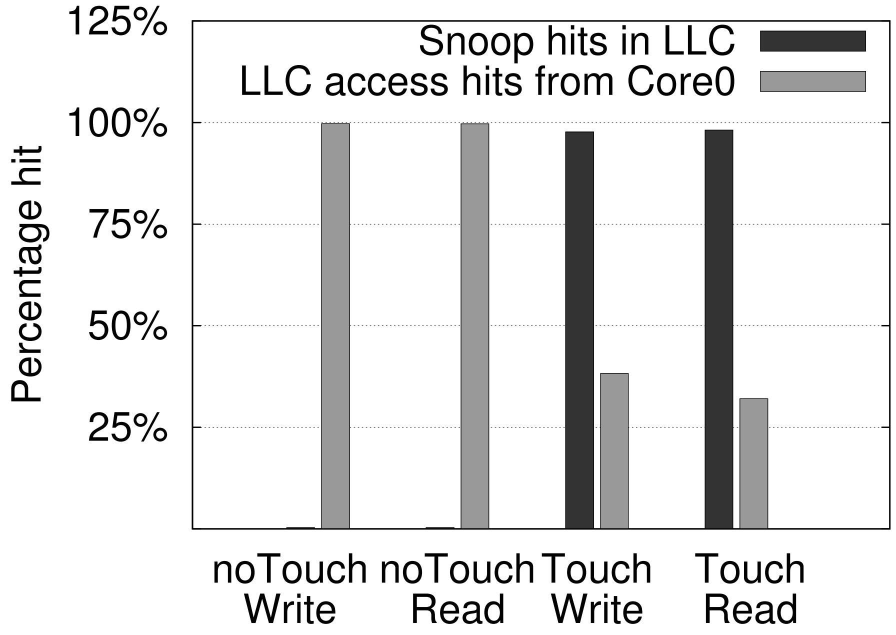


Figure 3.8: The snoop and the LLC hit rates.

Section 3.8.5.

In the next section, we present our results regarding various architectural overheads, and attribute costs to them on our systems. All data transfers in the next section use RDMA.

3.8.3 Result: Intention Mismatch between DMA Access and Cache Coherence

We start by analyzing the large performance penalties for touching the transmission buffer (as any real-world application would do). As shown in Table 3.4, the Touch mode access results in a 56% and 14% drop in performance for Write and Read Scans, respectively. In our experiment, the server always transmits the transmission buffer. To maintain coherence, snoop requests for the transmission buffer are generated when the DMA engine on the RNIC accesses the DRAM (see steps 3 and 4 in Figure 3.6). There are two possible outcomes of a snoop request: (a) a snoop miss, when the LLC does not contain snooped addresses, (b) a snoop hit, when the LLC contains snooped addresses. As the snoop requests are always generated for the transmission buffer, the Touch access has a high snoop hit rate. In the case of a snoop hit, the coherence engine must take appropriate actions to ensure IO coherence. Modified cache lines are evicted and written back (WB) to the DRAM to ensure that DMA access reads the latest content. In the case of clean Exclusive cache lines, nothing should be done. However, as we illustrate, the exact actions are implementation specific.

The performance drop for the Write Scan can be attributed to cache lines eviction and costly

	NoTouch	**Touch**	**Loss**
Write	300 μsecs	470 μsecs	56.6%
Read	275 μsecs	315 μsecs	14.5%

Table 3.4: Latency numbers for a complete request-response loop, measured at the client side for the different modes and scans on the server. The response buffer size is 256kB.

WBs to DRAM. However, the Read Scan on the transmission buffer unexpectedly also results in a performance drop. This behavior leads to a further investigation about snoop and coherence interaction. We measure the snoop hit rate by counting the snoop requests that hit the LLC. Similarly, we measure the LLC hit rate for accesses by the Core0 in the subsequent scan steps. Figure 3.8 shows our results. As expected, NoTouch access results in an LLC hit rate of almost 100%, with a negligible snoop hit rate. The Touch access results in a snoop hit rate of almost 100%. The high snoop hit rate evicts the cache lines and consequently, further access to the transmission buffer by the Core0 misses the LLC. The LLC misses are not capacity or conflict misses, as only cache lines in the Invalid state are filled. Further analysis reveals that the snoop requests are of type `REMOTE_RFO` (remote Request For cache line Ownership). This ownership request moves Exclusive cache lines to the Invalid state and discards them. This state transition resulted in mandatory cache line misses for the Read Scans.

The DMA-intention mismatch *may* be a multi-socket CPU specific behavior. The test CPU here is a Nehalem-EX CPU, which is a multi-socket CPU. A multi-socket CPU *may*[1] implement a different subset of the basic MESIF protocol than dual-sockets (Nehalem-EP) and single-socket (Nehalem-UP) CPUs. A specific implementation trades scalability (single, dual, multi sockets) with coherence maintenance overheads in hardware. Executing the key experiment from this subsection on a Nehalem-EP CPU resulted in the moving of the relevant cache lines from the Exclusive state to the Shared state because the DMA read snoop request was interpreted correctly by the coherence engine. However, we cannot confirm if this behavior is Nehalem specific or EX (multi-socket) processor specific due to the lack of enough data samples on different types of CPUs.

Summary: Write back of Modified cache lines is costly on Xeon E7520 due to high memory access latencies. However, more interestingly, E7520's coherence engine interprets a DMA read request during the data transmission as a `REMOTE_RFO`. This request for ownership forces the coherence engine to evict clean cache lines as well. The mismatch between DMA access intentions and coherence implementation results in a performance loss for the application where read-read sharing is expected.

[1]Personal discussion with an Intel employee.

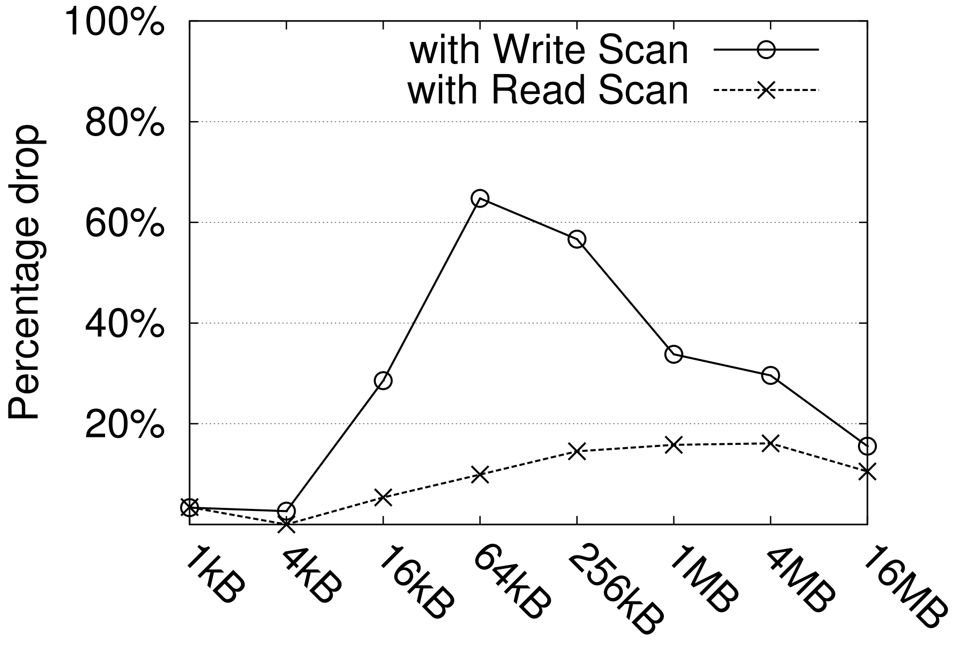


Figure 3.9: Performance degradation due to LLC misses and coherence overhead. The percentage performance drop is calculated by comparing the performances of the Touch accesses to the NoTouch accesses.

		1kB	4kB	16kB	64kB	256kB	1MB	4MB	16MB
	NoTouch	30	38	56	105	300	1,107	4,310	19,230
Write	Touch	31	39	72	173	470	1,481	5,586	22,222
	Drop in %	**3.3**	**2.6**	**28.5**	**64.7**	**56.6**	**33.7**	**29.6**	**15.5**
	NoTouch	29	38	56	101	275	974	3,777	15,873
Read	Touch	30	38	59	111	315	1128	4,385	17,543
	Drop in %	**3.4**	**0**	**5.3**	**9.9**	**14.5**	**15.8**	**16.0**	**10.5**

Table 3.5: Raw data for Write and Read scans for Figure 3.9. All numbers are in μsecs, representing the time it took to transfer a test buffer. The difference is calculated between NoTouch and Touch accesses.

3.8.4 Result: High LLC Misses and Coherence Overhead

In this section, we investigate the effect of the buffer size on the coherence overhead. The buffer size is directly related to the number of cache lines that need work for coherence maintenance. Large buffer sizes result in a large number of cache lines, and consequently add coherence overhead. For mandatory cache misses (in the case of Write Scan), accessing a large buffer from DRAM with a cold cache is also costly. Figure 3.9 (absolute numbers in Table 3.5) shows the effect of collective penalties of coherence overhead and cache misses. The y-axis shows performance degradation when Touch access is compared to NoTouch access. As shown in the previous section, the NoTouch access does not have any snoop-hits and maintains a high cache-hit rate. The Touch access results in snoop-hits and additional coherence maintenance work. For small buffer sizes, due to the small number of cache lines, the cost of coherence maintenance is small and relatively low compared to the overall network latency. As we increase the buffer size, this cost increases and becomes the dominant part of the overhead (4-256kB range). Further down the x-axis, with large buffer sizes the transmission cost on the 10GbE link becomes dominant and shadows the coherence overhead.

Summary: The shared access of the transmission buffer between the CPU core and network accelerator brings the accelerator into the memory coherence domain. Unlike the well studied (and optimized) behavior of memory sharing among many CPU cores, the performance implications of this shared access are not well understood. Different natures (inclusive or exclusive of L1 and L2) and implementations (topology, on- or off-chip) of last-level caches make reasoning about the performance a difficult problem. The architectural overheads stemming from cache misses, CPU stalls, etc., have now become comparable to the network latencies. As illustrated, the (potential) performance gains in a shared access environment can easily be eclipsed by high architectural overheads.

3.8.5 Result: Pre-Fetching Sensitivity for Buffer Access Patterns

Write back and (forced) eviction of cache lines result in mandatory cache misses. Because our benchmark is doing a sequential access (in a `for` loop) to the transmission buffer, the Next-line hardware pre-fetcher can fetch subsequent cache lines to avoid the high cache-miss penalty. However, real-world applications have complex data structure layouts in the transmission buffer, where parts of the buffers can be transmitted and received. Further, data can be accessed based on freshness, or urgent interest, e.g., only accessing the keys in a key-value pair. These types of accesses are strictly non-sequential and do not activate hardware pre-fetching. To understand the benefit of sequential access, we explicitly enabled and disabled pre-fetching in the BIOS. Figure 3.10 (absolute numbers in Table 3.6) shows our findings. Hardware pre-fetchers can help to accelerate the end-application performance when accessing the cold transmission buffer, but only under restricted access patterns. The gains from the sequential access (due to hardware fetching) can be as high as 60%.

Summary: Non-sequential access patterns that do not match any available pre-fetchers (adjacent-line, DCU streamer, etc.) will not get any performance boost. Unaccelerated network stacks get benefits from software pre-fetching hints (using `prefetch` family instructions) passed during protocol processing and data copying in the kernel. In contrast, with RDMA, where data is directly transmitted and received from userspace, side-effects of DMA access

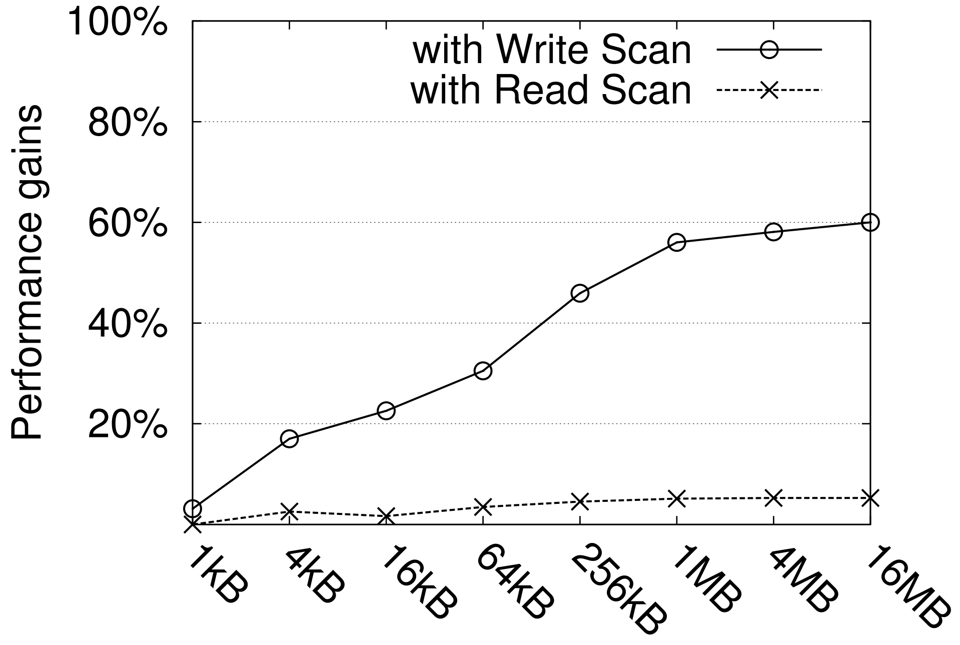


Figure 3.10: Performance gains due to Next-line hardware pre-fetching.

		1kB	4kB	16kB	64kB	256kB	1MB	4MB	16MB
	Disabled	32	47	93	249	869	3,367	13,333	55,555
Write	Enabled	31	39	72	173	470	1,481	5,586	22,222
	Gains in %	**3.1**	**17.0**	**22.5**	**30.5**	**45.9**	**56.0**	**58.1**	**60**
	Disabled	30	39	60	115	330	1,189	4,629	18,518
Read	Enabled	30	38	59	111	315	1128	4,385	17,543
	Gains in %	**0**	**2.5**	**1.6**	**3.4**	**4.5**	**5.1**	**5.2**	**5.2**

Table 3.6: Raw data for Write and Read scans with pre-fetching enabled or disabled for Figure 3.10. All numbers are in μsecs, representing the time it took to transfer a test buffer. The difference is calculated between pre-fetching disabled to enabled settings.

(e.g., cold cache) are completely visible to the application and cannot be avoided. Hence, various cache optimization techniques and large cache sizes are of little help, and factors such as DRAM access latencies start to dominate the performance of end-applications.

3.8.6 Analysis of Results

The distributed execution of application and network code on network accelerators is a radical departure from the traditional model, where everything is optimized to be accessed from a centralized host CPU. Hence, the interaction among off-chip non-CPU components becomes an important performance factor. These components include shared last-level caches, different coherency engine implementations, transport links (e.g., Intel QPI) to cores, and DRAMs, etc.

We realize that our investigation is processor and architecture specific, and therefore it would be wrong to make any final conclusions regarding the RDMA network technology. However, our findings do highlight (potential) architectural pitfalls, which are usually hidden from high-level applications while deploying network accelerators in large-scale environments.

Earlier works discuss network offloading specifically in the context of transparent TCP offloading that maintains the socket interface to applications [128, 244]. Our analysis, however, is not limited to TCP/Socket interface and has wider implications. As general IO offloaded devices and accelerators are becoming part of mainstream computing, IO latencies are rapidly becoming closer to architectural overheads. Our findings have further implications as RDMA IO interface and semantics are now being investigated even for GPUs [192, 261] and storage [341], for which we also have made a case in Chapters 4 and 5.

3.8.6.1 Impact on Networked Applications

Various high-performance NoSQL data stores [13, 121] have been proposed to serve multiple clients. Efforts have been made to transfer data by leveraging the capabilities of RDMA networks [183, 241, 326]. Such applications that reportedly enjoy performance gains with RDMA can also experience performance loss if used with a particular CPU or chipset in a low-latency environment. Overheads reported in Section 3.8.3 affect servers, those reported in Section 3.8.5 affect clients, and those discussed in Section 3.8.4 affect both. Other in-memory data stores [264, 367] are also susceptible to suffer performance losses. Also, in a shared environment cluster where storage and compute nodes are co-located, network RDMA operations of the storage application can potentially purge warm caches of the compute applications.

However, certain classes of applications are also less likely to be affected by reported overheads. Applications which have limited CPU-NIC interaction, such as media streaming where the CPU brings video data into the memory once, and network acceleration such as done by RDMA, which can be used to serve content repeatedly to multiple clients [132], are not exposed to the overheads. Another class of applications contains those where the non-network part of the application, either computation or disk IO, dominates the overall client latencies. These latencies are orders of magnitude higher (in msecs) than latencies reported in our setup.

3.8.6.2 Architectural Implications

In the previous sections, we have illustrated that architectural overheads can eclipse gains from RDMA networks in high-performance environments. The exact overhead cost is sensitive to a particular implementation of the coherence engine, cache-miss costs, write-back and DRAM access latencies, etc. Hence, application developers must now be aware of costs associated with low-level architectural events. With ever increasing complex NICs and CPUs internals, there is a growing confusion about performance [316]. Different processor vendors implement different variations (or even a subset) of cache coherence protocols. Implementation and cost of architectural features can be different even between different models of the same processor generation [32].

Intel addresses some of the issues discovered in this chapter. For a later generation of SandyBridge CPUs, Intel introduced Data Direct I/O Technology or DDIO [168]. With DDIO, incoming or outgoing DMA operations can access last-level CPU caches to read or write data. This mechanism solves the basic problems of mandatory cache misses for freshly received data and forced dirty cache eviction on DMA. However, the use of last-level cache as a data source or sink for IO devices also opens other problems such as how to avoid cache trashing due to excessive IO, how to selectively enable or disable this feature on a per device or per access basis, how to steer data to a particular shared last-level cache in a multi-socket setup, etc. Nonetheless, such an approach is definitely helpful, but requires a more generalized, configurable, or programmable solution than a system-wide static setup as currently supported in DDIO.

With the high residual transistor count on the CPU chips, it should now be possible to implement high-performance NICs on CPU chips [62, 173]. This NIC-CPU integration makes the network a first class citizen of CPUs with access to all on-chip resources such as caches and memory controllers. This access enables a better interaction between caches and network IO. Furthermore, network access to memories (DRAMs, caches or even NVRAMs) can be optimized (and reasoned about) using similar techniques for manycore CPUs. As there is no final word for high-performance network interfaces (hardware and software), it is a challenging task to design a single chip to meet all demands. However, demands for very high network performance (100Gbits/sec with less than 1μsecond latency [292]) necessitate this integration.

Another orthogonal issue is RDMA network integration in non IO-coherent architectures, e.g., ARM. In such architectures, understanding the interaction among non-CPU components (caches, DMA and coherence) is even necessary for the sake of correctness. The current OFED RDMA subsystem on Linux is broken for non IO-coherent architectures [338].

3.8.7 Summary

RDMA offers low ($\sim$1s μsecs) data access latencies together with very high data bandwidths (10-40-100Gbps) with a zero CPU load. Due to its unique performance potential, RDMA has (again) started to draw a lot of attention from the systems building community [107, 183, 241, 264, 325, 326, 339]. However, managing network resources in userspace does expose applications to low-level hardware details which are usually hidden in the operating system kernel. For example, to reduce cache pollution, Linux uses non-temporal copy instructions (e.g., `movnti`) to copy data from user buffers to SKBs. It also hides the cost of cache misses from applications by doing pre-fetching and data copy during the network processing.

3.9 Conclusion

In this chapter, we have provided the necessary background information on the RDMA technology, its implementations, networking abstractions, network operations, and programming interfaces. We then further documented our experience with RDMA network IO operations. We found out that a number of (un)related parameters can significantly affect the gains of applications when using RDMA network operations.

We identify performance factors that span the whole stack, ranging from low-level architectural issues (cache and DMA interaction, hardware pre-fetching) to the high-level application parameters (buffer size, access pattern) and attribute costs to them on our systems. Unfortunately, there is no silver bullet solution that guarantees performance improvements without any drawbacks or concerns. As we move toward a heterogeneous computing environment, the use of network offload/accelerator devices will become more common. This will change the decade old assumption that all processing and data access happens from the central CPU. Thus, instead of focusing on a high CPU core performance, system architects must take a holistic, system-wide approach toward network offloading/accelerator integration to achieve application performance boosts.

Although our findings are RDMA and CPU specific, they are illustrative of a growing confusion about performance when using network offloading/accelerator devices. Reasoning about RDMA performance requires a good understanding of CPU, NIC, and architecture internals. Complex off-CPU components, primitive performance monitoring facilities, ambiguous documentation of hardware, and a limited software support, etc. make RDMA performance analysis a very challenging task.

A general recommendation for RDMA practitioners is to benchmark and quantify their hardware for various micro-architectural features such as DRAM latency, LLC misses, pre-fetching misses, coherence overheads, etc. For the rest of the thesis, we use a cluster with servers equipped with CPUs where the cost of these micro-architectural features is low.

4 A Case for Unified High-Performance IO

Modern storage and networking devices have changed dramatically over the last decade. While end-host networking performance and capabilities have improved gradually and consistently over the years, Non-Volatile Memory (NVM) storage became part of today's mainstream computing in a relatively short span of time. The rapid emergence of NVM devices has also exposed many performance inefficiencies and bottlenecks in the design of traditional storage stacks and abstractions. To improve the situation, fragmented efforts have been made to look into lightweight, kernel-bypassing, low-latency, asynchronous, and directly-accessible storage stacks. Yet, there is a pressing need for a high-performance storage stack.

As discussed in Chapter 2, two decades ago when networking stacks were unable to meet strict performance demands of emerging parallel and distributed applications, multiple high-performance network concepts, network interfaces, operating system mechanisms, networking stack implementations, and application abstractions, etc., were developed. In this chapter, we now draw parallels between evolutions of storage and networking stacks to illustrate synergies between high-performance storage requirements and concepts from the networking domain. We identify common high-performance IO properties and recent efforts in storage to achieve those properties. Instead of reinventing the performance wheel, we present a case for developing a unified IO abstraction for high-performance storage *and* networking devices using modern, mature networking frameworks. We then discuss the key characteristic properties, opportunities, required support, and the open issues when applying networking concepts in the storage domain.

4.1 The Struggles of the Storage Stack

Slow storage has been the Achilles' heel for data processing systems. Historically, disk bandwidth and access latency have consistently lagged behind their capacity and packing improvements [116]. However, NVM storage offers unprecedented improvements over disks with multi-

	CPU Speed	Net BW	Storage BW
1980-2010	1000×	3000×	50×
2010-now	1-1.5×	4-10×	10-100×

Table 4.1: The widening CPU-IO performance gap between a CPU speed and bandwidth improvements of IO devices. In the last couple of years, performance (both bandwidth and latency) of IO devices continues to improve rapidly whereas a single CPU speed improvement has only seen marginal gains.

Gigabit bandwidth and access latencies in microseconds. This performance paradigm shift has been the most fundamental and significant change in storage since the advent of magnetic disks in the 1970s. Unsurprisingly, not much has changed in the way operating systems manage storage devices. The following factors motivate a need for reevaluation of the complete storage stack to support high data rates.

4.1.1 Rising CPU-IO Gap

Hardware landscape has changed considerably during the last decade. Modern IO devices are becoming significantly faster than CPUs. With stalled single CPU speed scaling, CPUs can no longer keep up with the high data rates from devices (see Table 4.1). As a result, traditional CPU-centric storage stacks, where the CPU orchestrates data movement from relatively slow disks to DRAM buffers, have started to show performance strains in high IO operation/sec (IOPS) environments. This CPU-bottleneck limits deliverable performance to applications, despite having orders of magnitude performance improvements in hardware. As NVM technologies continue to mature, this performance gap between CPU and devices will widen. Manycore CPUs come to rescue, however, the overhead due to locking, synchronization, and coherency, etc., limits the overall achievable IO performance [53]. Also, using multiple cores to satisfy high-CPU demands of IO operations is performance inefficient. As computing gradually moves toward Exascale, the performance efficiency [27] has direct implications for the amount of resources (CPU, storage, network), energy, and cost.

4.1.2 Software and Access Overhead

NVMs packed as *fast disks* have been the least intrusive and most economical way of integration so far. As the performance characteristics of underlying storage media have changed significantly, the traditional disk-based optimizations are now considered expensive, obsolete, and intrusive. For example, IO pre-fetching, buffer caching, request reordering and merging, etc., all require additional time and CPU cycles (which are limited) and, hence, may even lead to performance degradation with NVMs [68, 306]. Layering and multiplexing within operating

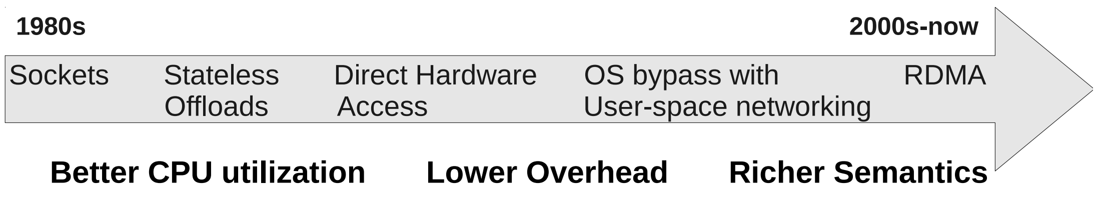


Figure 4.1: Evolution of high-performance network properties. The arrow does not represent any casual dependency or temporal precedence in the development.

systems, and virtualized environments put further penalties on the performance. With additional layers, the incurred overhead is non-linear which results in a rapid performance loss.

4.1.3 Restrictive APIs

Storage APIs are not designed to expose NVM capabilities to applications. Features such as accessing flash chips directly [137], virtualize storage space [181], parallel read/write ports [70], atomic updates [267], etc., can significantly simplify storage logic while improving deliverable IO performance. Furthermore, they also restrict passing useful information about the nature of IO across the layers to NVM devices. Useful access information like scratch-pad access (light-weight, single copy, no protection), log-access (write-append, random reads), or range invalidation can help significantly with better device management, and consequently performance.

4.2 A View from High-Performance Networks

This appetite for efficient high-bandwidth and low-latency access to data is not unique to storage. The networking community has always lived with stringent application demands for high IOPS. Over the last 30 years, various techniques and concrete implementations of networking stacks have been developed to match the periodic interconnect bandwidth and latency improvements. Simple optimizations such as checksum and segmentation offloads gradually delegated a part of the packet generation to network controllers [193]. DMA support was added to free CPU cycles from data movement [282]. Adaptive interrupt coalescing and device polling resulted in better device management under load [248]. These simple techniques freed precious CPU cycles and helped to close the interim CPU-network gap that arose from continuous interconnect improvements, e.g., Megabit to Gigabit Ethernet.

Though effective and helpful for high-bandwidth data transfers, these optimizations yielded little improvements in end-to-end application latencies. Latency requirements of high-performance applications necessitated more radical approaches. To reduce every potential overhead, these approaches favored a fresh redesign of the complete networking stack and devel-

	Networks	Moneta-D	Gordon	NVHeap	FusionIO
Efficient Hardware Access	yes	yes	yes	yes	yes
Operating System Bypass	yes	yes	N/A	N/A	proprietary
Zero Copy Data Movement	yes	no	N/A	possibly	no
Asynchronous IO Model	yes	yes	yes	N/A	yes
Synchronous Completion	yes	no	no	N/A	no
Rich IO and API	yes	no	no	transactions	proprietary

Table 4.2: Comparison of recent storage efforts to achieve high-performance properties. N/A denotes that the property is not the primary focus of the work.

oped novel host interfaces, interconnects, networking principles, and operating system mechanisms [111, 112, 352, 358].

As these high-performance stacks became popular, network architects soon identified a common requirement for rich network IO semantics from many applications. These semantics and network operations made the development of complex applications easier. Naturally, to reflect the gradual progress made in operating systems and networking hardware, networking API and interfaces also evolved. The holistic approach taken by networks helped in developing many key ideas that are now an integral part of any modern high-performance interconnect such as InfiniBand and iWARP.

4.3 Distilling Common High-Performance IO Properties

Given the recent rejuvenated interest in high performance IO, this is a timely discussion about the key principles and properties that enabled efficient IO for networks. Though these properties are inspired from experiences in networks, we argue that they are equally applicable to the storage domain as well. We discuss how recent efforts to integrate NVM are already exploring subsets of these properties (see Table 4.2).

4.3.1 Efficient Host Interfaces and Hardware Access

In Section 2.1.3, we discussed how high-performance networks manage to keep the host overhead minimal by *directly mapping the hardware resources* to applications as private channels or queues. As the overhead from disk based storage protocols (e.g., SCSI) and host interfaces (e.g., AHCI) becomes unbearable [199], research projects such as Moneta [69] and multiple commercial offerings [137] have started to look into directly accessible hardware with improved host interfaces. Moneta-D offers safe user-space access to directly accessible NVM devices [71]. Though it helps reducing the overhead associated with issuing IO requests and notification delivery, these efforts lack the generality of user-space networking. For example, Moneta-D does not have a generic mechanism (similar to a completion channel for RDMA networks) for asynchronous completion notification to userspace. It is not possible to batch notifications or share resources between multiple IO channels. Other industrial efforts such as NVMe, which do provide multiple queue-based interfaces to a device, but only allow trusted kernel clients such as a file system.

4.3.2 Operating System Bypass

By separating data from the control path and pre-allocating IO resources, high-performance networks involve operating systems in selective managerial tasks such as resource accounting. Enforcement of the security policies takes place in network hardware. Recent storage research has looked into similar techniques to avoid unnecessary operating system involvement with request scheduling, batching, reordering, dynamic resource allocation, and security enforcement. Moneta-D pre-allocates DMA buffers, directly posts requests, and offloads file permission checks to a capable storage hardware [71]. Though the operating system is still involved in DMA buffer management, file check offloading, and permission evictions, etc., it is kept out of the IO loop between the application and hardware. The storage stack of the Arrakis operating system, a recently proposed state-of-the-art system, completely eliminates the kernel involvement in the IO path by assigning a virtualized storage interface controller (VSIC) to applications [274].

4.3.3 Zero-Copy Data Movement

High-performance network controllers maintain sufficient contextual meta-data to multiplex and securely DMA data directly into application buffers. They also support arbitrary application buffer layouts, offset calculations, and scatter-gather IO. Together with directly accessible IO hardware and operating system bypassing, the CPU is now completely decoupled from the fast data flow. Efforts have been made to achieve zero-copy storage, but they are either limited (small number of user accessible DMA buffers) or restricted (aligned layout of user buffers). Zero copy storage is possible with mmap'ed files as application buffers, but not achievable using other memory allocation methods such as malloc.

4.3.4 Asynchronous IO

High-performance storage interfaces such as epoll are based upon the *readiness* instead of the *asynchronous-notification* model. This model does not provide sufficient concurrency to exploit full device potential and makes optimizations such as request batching and selective notifications very difficult. In multi-stage environments, where data passes through multiple storage devices, asynchronous IO also gives better IO scheduling opportunities for a smooth end-to-end data flow. Recent efforts such as Moneta-D [71] and MegaPipe [153] have advocated the use of the asynchronous model with private channels.

4.3.5 Polling and Synchronous Completion

A synchronous completion allows posting IO requests and reaping completion notifications without context switches. Networks support non-blocking posting of batch requests with polling completion within the same user-context. For low-latency networks, this method delivers better application latencies at the expense of higher CPU utilization. However, as emerging NVM device latencies will fall below context switch latencies, this approach proves to be more favorable for storage as well [364]. Like high-performance networks, recent efforts in storage have also looked into supporting adaptive switching between blocking and polling for completion [313].

4.3.6 Rich IO Operations and APIs

Modern interconnects support operations such as remote data read and writes, fencing, atomic compare and swap, atomic add, scatter-gather IO, etc. Such hardware primitives make complex application development simpler. Similar experiences are also reported by the storage researchers in [181, 267]. NVHeap [89] (and other concurrent works [347, 351]) saves the heap-state of an active application on NVM in a novel way and provides transactional support to access it. As the NVM integration has been transparent, these approaches do not provide much control over IO.

4.4 A Case for Unification

The current unified abstraction of *files* is not aimed at high performance and has plenty of performance overhead due to the need for global synchronization, inefficient IO memory management, and the lack of useful hints in multi-core environments [153]. One potential solution is to redesign the complete storage stack from scratch. However, as modern high-performance networking stacks offer very mature and stable implementations of the desired key high-performance properties (see Table 4.2), in this chapter we propose using them to access and transfer data to/from NVM devices. This gives storage architects the opportunity to reuse the developed frameworks without undergoing a similar evolution.

Similar to the networks, high-performance storage interfaces can provide directly user-space mapped hardware IO queues or channels for request postings and completion notifications. Operating system and user-space device libraries provide support for setting up direct NVM

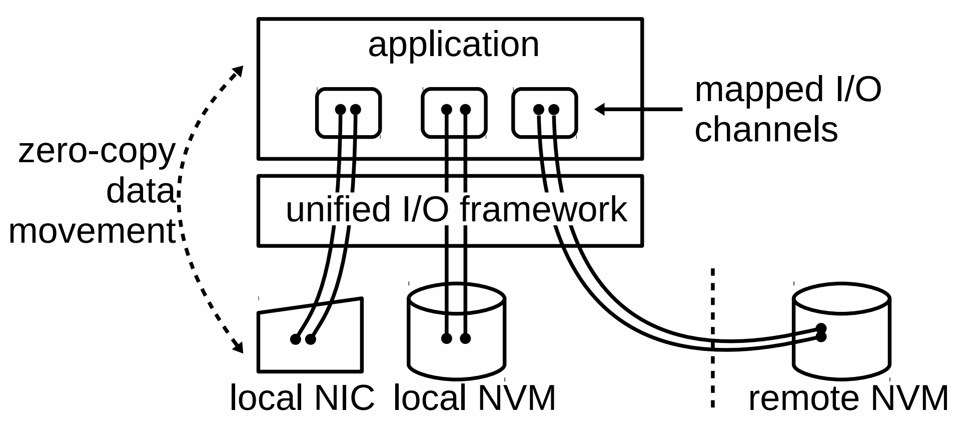


Figure 4.2: Illustration of the unified IO stack that can deal with network, local NVMs, and remote NVMs under a common framework.

device access from the user-space. The devices implement multiple IO request opcodes and associated completion semantics.

The access abstraction is a simple byte-addressable storage address space. Byte-addressable abstraction does not impose any data structure and is expressive enough to support a wide variety of higher-level storage systems such as hierarchical file-systems, databases or object stores. These systems are responsible for the translation of higher-level storage objects, such as a file or a database column, to a specific device address range. After the translation, the data transfer happens directly between the device and the user-space buffers. The data transfer is done in a manner similar to a remote memory read or write operation. The following factors – lined out in Subsections 4.4.1-4.4.3 – further support our case for a unified stack for common device management and data access.

4.4.1 Unified Operating System Support

The unification blurs the traditional boundary between network and storage and enables a common evolution of high-performance IO frameworks. A single stack provides uniform IO semantics and guarantees across multiple kinds of devices (see Figure 4.2). This unification also simplifies the implementation of IO mechanisms inside an operating system. Both, network and storage, need capabilities to directly access hardware, use adaptive notifications (callbacks, blocking, polling, etc.), share IO memory management with applications, etc. Looking beyond the performance properties, networking stacks also have everything from device detection, configuration management, capabilities discovery, to efficient memory management, etc. These services largely simplify the device management.

4.4.2 Ready to Use

Any modern high-performance network stack implementation can be used as a drop-in replacement to access storage. The replacement allows storage to reuse interfaces, data structures, APIs, and even (up to a certain extent) concrete implementations of a stack. Many networking semantics have an immediate appeal for storage applications. Features such as remote read/writes can be used to access data from storage. Fencing and ordering among IO operations ensure proper consistency guarantees (similar to a remote memory) for storage. Multiple storage devices can form a multi/broadcasting group to implement replication. QoS can be implemented by using multiple network traffic classes. Furthermore, end-to-end semantics of the interface/API ensure light-weight data access even in multi-layer access environments, e.g., virtualization.

4.4.3 Favorable Advancements

Lastly, recent architecture and systems advancements also facilitate this unification. The byte-addressability nature of NVMs (e.g., PCM) makes data access as simple as reading remote memory. This fits nicely with remote memory access semantics of RDMA. However, an NVM device itself does not have to natively support byte addressability as long as it understands the RDMA access model. With the revised host-interfaces [10], NVMs can now be directly accessed via networks, further blurring the gap between local and remote storage. Repartitioning storage responsibilities between application and hardware also makes it possible to reuse standard user-space networking stacks.

4.5 Discussion

4.5.1 Operating System Support

Efficient, streamlined IO execution requires support from both operating systems and hardware. As the key responsibilities of an operating system - abstraction, multiplexing, and layering - seem too prohibitive for high-performance, we must revise the responsibilities between devices and the operating system. Much of IO management complexity from within operating systems can now be delegated to applications and hardware. Instead of micro-managing, an Exokernel-like [140] approach for OS design is more desirable. For example, instead of performing fine-grained IO scheduling within an operating system, as has been done traditionally, it should be involved selectively in coarse-grained decisions such as when to schedule an IO request class (e.g real-time, or backup) or controlling parallelism and concurrency within hardware for QoS. Hardware does a better job in fine-grained scheduling of individual IO requests. Additionally, operating system developers must remove the stigma associated with IO offloading. An operating system must be able to efficiently manage, extract, and communicate necessary contextual information about IO to the devices.

Protection in a shared environment can be provided by generating access capabilities with the help of IO devices. For example, an RDMA device generates an access identifier tag during the memory registration for a data buffer. This tag must be presented by an application for any access to the data buffer to verify access rights.

4.5.2 Hardware Support

Network and storage devices must be able to allocate IO channels, generate protection identifiers, install user contexts and memory mappings, etc. To be efficiently managed by the operating system, device vendors need to come up with a standard communication interface for management and configuration such as OpenFlow [229] for switches. Efforts have been made recently to standardize the storage host-device interface [10, 14]. Interestingly, NVMExpress [10] shares many key performance properties with high performance networking devices such as directly accessible hardware resources, doorbell write to issue a command, multiple request and response queues, capability discovery, interrupt coalescing, configurable data block size, etc.

To avoid unnecessary operating system involvement, devices must be educated about logical IO primitives with gradually increasing complexity. As we discussed earlier, high-performance interconnects such as InfiniBand already support rudimentary forms of these operations such as atomic fetch and add, atomic compare and swap, etc. We believe that with a minimal set of basic integrated operations (e.g., locking, logging, atomicity, serialization, etc.), it should be possible to build higher-level complex primitives such as transactions or replication (using multicasting) without bloating the IO stack.

4.5.3 Open Issues

Multi-stage resource allocation: Resource allocation in the control path of high-performance networks is a multi-stage process. Different IO resources (with associated states) are allocated at various stages of a connection setup, e.g., open channel, route discovery, device resolution, connect, accept, etc. However, storage has a simple single-stage (e.g., open a file) access process. Reserving storage resources in a single step may lead to wasteful resource usage. To avoid overcommitment, additional access pattern and range related information must be passed to the storage. However, due to the lack of support in network interfaces to pass this information, it requires further development of new APIs.

Hardware multiplexing: Modern high-performance controllers can typically maintain 64K to 1M active contexts. However, high-level storage primitives such as files can be in the billions. This will require some coarse-grained multiplexing support from the operating system such as the one found for the virtual memory subsystem. By installing hardware contexts in the *storage page-table*, the operating system can move out of the way of normal IO processing. This mechanism maintains the operating system control over IO resources without sacrificing the performance. However, this functionality will require support from the storage hardware, e.g., generating storage faults for an invalid access to files.

File semantics: Files are shared more often than sockets. Depending upon the mode, file sharing can lead to different consistency semantics. For example, accessing a shared file using a common request queue among multiple applications can potentially provide serialization guarantees, but this may not be possible with different request queues or may require different IO opcode. The packet oriented nature of network APIs makes development of stream-based storage applications difficult.

IO failures: Direct-access zero-copy IO has visible side-effects in the case of a failed operation. The byte-addressable nature of NVMs makes data corruption detection even harder. Hence, a more sophisticated and precise error reporting and cancellation framework is required. One possible solution is to maintain error and log data structures in DRAM, thus, if there is a failure the operating system can still perform error diagnosis on it.

4.6 Conclusion

Storage stacks are at a familiar crossroad. Performance of IO devices are improving at a much faster rate than the speed of a single CPU. Over the last 30 years, networks have undergone an evolutionary transformation to support high-performance IO. In this chapter, we argue that storage does not have to repeat the same steps as the networks had to and wait another 30 years to undergo the same transformations. Storage developers can directly use abstractions, frameworks, and interfaces developed by high-performance networks. This unification instantly enables efficient, light-weight high-IOPS access to NVM devices. In the next chapter, we present the design and implementation of FlashNet, a software devices that unifies flash management with RDMA operations.

5

FlashNet: A Unified High-Performance IO Stack

Modern distributed data processing frameworks such as Apache Spark or Hadoop routinely analyze Terabytes of data stored across hundreds of storage servers. Consequently, the performance of these frameworks depends considerably on the IO performance of the many *storage and network* devices involved. As discussed in the last chapter, modern IO devices have undergone a rapid evolution during the last decade. Ethernet, the most popular network technology, supports 10, 40, and 100 Gbits/sec data rates with single-digit microsecond link latency. With comparable advancements in Non-Volatile storage such as Flash, storage devices now also offer multi-Gigabits/sec bandwidths with μsecs access latencies. This rapid evolution has also put tremendous pressure on traditional IO stacks. Overheads stemming from maintaining IO abstractions, scheduling, context switching, cache flushes, contention, execution of generic OS, etc., have contributed significantly to the loss of IO efficiency. Stalling CPU speeds have only made things worse recently, up to a point where IO stacks have become the new IO performance bottleneck.

In response to these trends, several efforts have been put in place to improve the IO efficiency, typically by limiting or eliminating operating system (OS) involvement in the data path [274]. However, these efforts exclusively either target the network [44, 153, 178, 224] or the storage stack [53, 71, 314, 364], but not the combination of both. As a result, access to storage over the network still requires the application to orchestrate the data access by engaging the OS and the file system multiple times for every IO operation. Furthermore, the established solutions in this space such as NFS or iSCSI do not deliver network data efficiently to client buffers, and hence fall short of providing full performance to IO-intensive applications.

In this chapter we present FlashNet, a unified IO stack architecture that enables fast and efficient networked data accesses from remote flash devices. FlashNet builds upon the data and control path separation principle of Remote Direct Memory Access (RDMA) networks and extends it to storage with the help of a file system and a flash controller. This extension establishes an *end-to-end* data path which reduces the unnecessary application and OS (including file system) involvement in network-storage IO processing. The unification of RDMA and flash access in FlashNet is not just cosmetic. While processing IO requests, FlashNet identifies the source/sink files of a network-storage IO request by leveraging RDMA's buffer tagging mechanism and operational semantics. RDMA message fragment headers are used to pre-fetch data from flash to hide high (w.r.t the network) storage latencies. RDMA network data access patterns, frequencies, and usage are used for a better flash device management. These steps result in a better flash management while delivering high performance for remote data accesses.

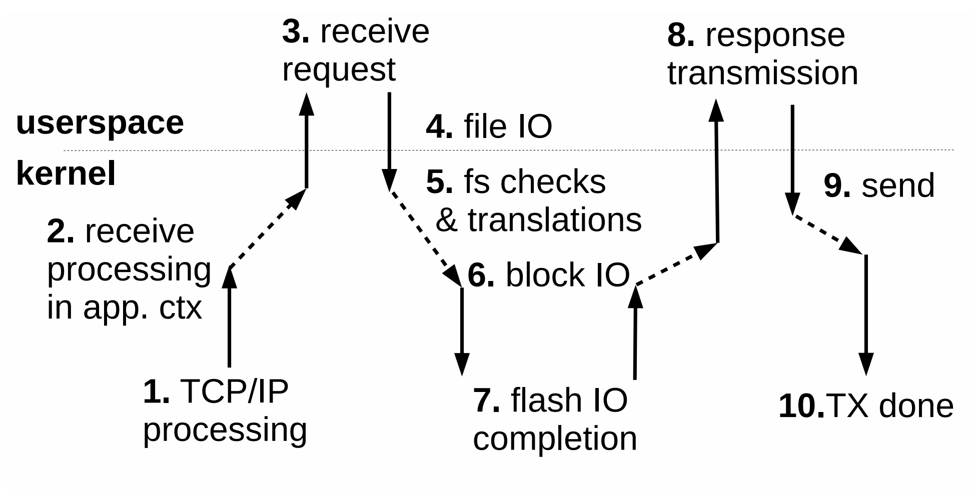
FlashNet is designed to be fully RDMA-compatible, thus enabling hybrid RNIC/FlashNet deployments. Consequently, applications that have previously used RDMA to efficiently access remote memory require minimum changes to access data to/from flash storage using FlashNet.

Our specific contributions in this work include (a) extending and unifying the path separation philosophy of RDMA networks for remote flash storage accesses; (b) building FlashNet, a unified IO stack as a proof of the unification concept; (c) evaluating FlashNet in a distributed setting, highlighting its raw performance, IO efficiency, deployment opportunities with a hybrid FlashNet-RDMA NIC (RNIC) setup.

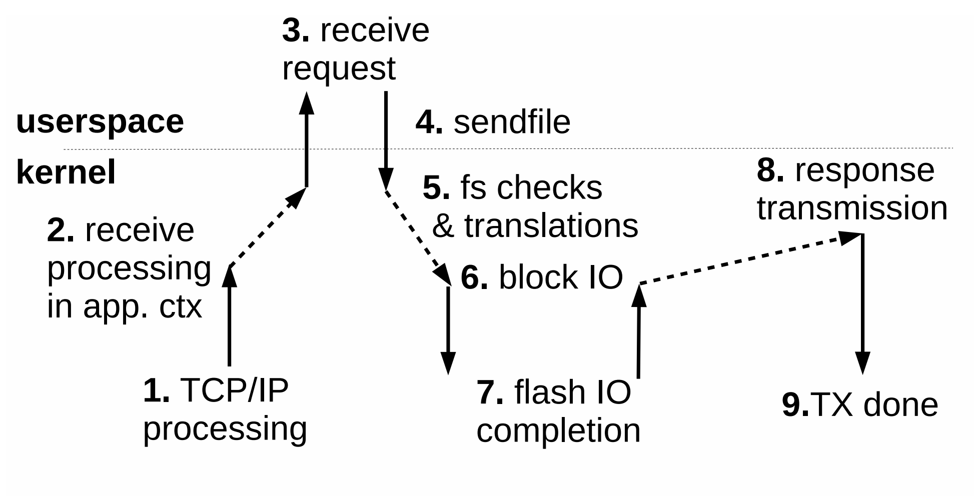
5.1 The Cost of High-Performance IO

Setup: In contrast to the dedicated, appliance-based approach to serve storage, in this work we consider a setup where off-the-shelf, general-purpose machines are used to run storage servers. Apart from running a storage server, these machines may also be involved in hosting and executing other associated services such as web servers, distributed data processing, and resource management frameworks, etc. Hence, due to the general applications of these servers, these systems require full-fledged operating system support to manage, multiplex, and schedule systems resources (e.g., CPU, IO, memory, etc.). This deployment scenario is typical of modern day data center infrastructure.

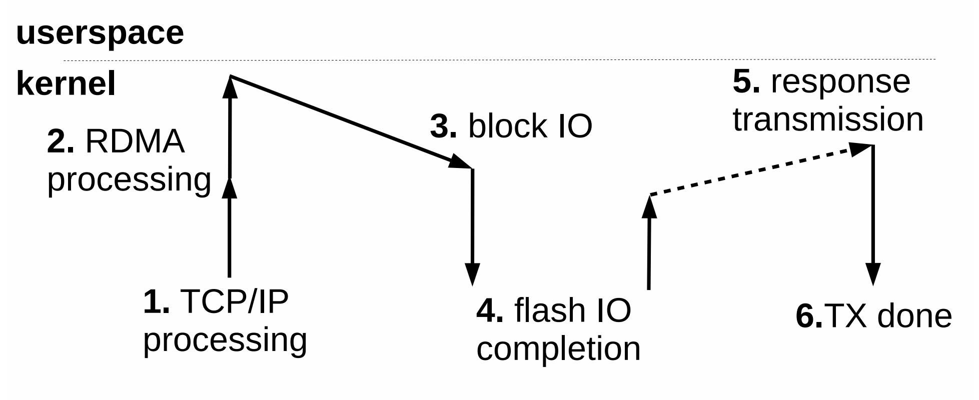
We start by quantifying the overhead associated with the constant application and OS (file system, generic code, checks, locks, etc.) involvement in processing of net-storage requests. Consider a client-server environment where a storage server is responsible for processing client requests to access data from a storage device. A typical example of such a deployment is a datanode in HDFS, or a key-value server. Hence, the peak performance of the storage server depends upon the efficiency of both the network *and* the storage stack.



(a) Socket with file IO.



(b) sendfile.



(c) FlashNet RDMA read.

Figure 5.1: IO execution paths of a single net-storage read request with (a) socket `send/recv` and file read IO; (b) socket `send/receive` and `sendfile`; (c) FlashNet's RDMA `read` operation. Dotted lines represent a possibility of a context switch during the execution.

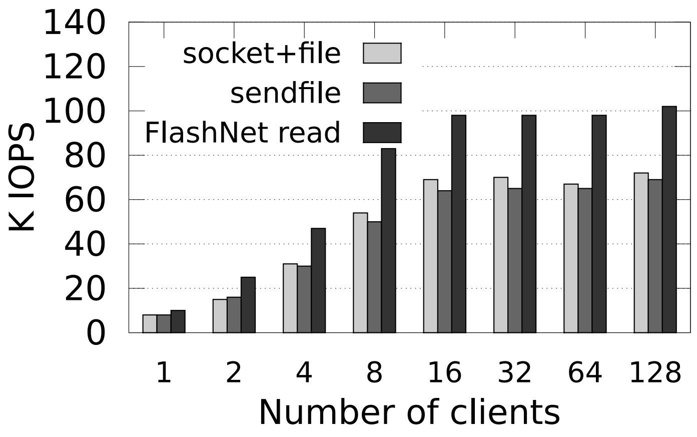


Figure 5.2: 4kB IOPS with one server CPU core enabled.

We therefore measured peak net-storage IO operations per second (IOPS) per core delivered by the server. The single core configuration emphasizes the CPU cycles/IOPS requirements of the evaluated IO configurations. We perform measurements using the netperf benchmark [9] where we added support for reading data from a remote flash device using TCP sockets and RDMA-based FlashNet operations. The netperf server and the clients run on a 9-machine testbed (see Table 5.3), connected via a 40 Gbits/sec network, running Linux version `3.13.11`. The storage server runs on a dedicated machine containing a PCIe-attached enterprise-level flash device. Clients run on the rest of the 8 machines and repeatedly request the server over the network to read 4kB data from a large file stored on the ext4 file system on the remote flash device.

To represent a data-dependent workload where a client cannot issue the next request until the last one has finished, clients have only one outstanding request at a time. The netperf server forks a new server process to handle requests for every connected client. We report performance numbers for three IO configurations:

1. *Socket with file:* The server uses `send/recv` on a TCP socket for network IO and a `read` syscall for file IO from the flash device in the direct mode (`O_DIRECT` flag). In this approach, the application is involved in both control and data operations. (see Figure 5.1(a)).

2. *sendfile*: A similar approach to the previous setup, but with the server using the `sendfile` mechanism to eliminate its involvement from the data transmission. Note that the application still remains involved in the control loop processing of the net-storage request that involves reading a client's request from the network buffer and instructing the kernel to initiate the transfer by calling `sendfile`. (see Figure 5.1(b))

	socket + block IO	sendfile	FlashNet read
networking	19.3%	17.6%	20.6%
storage	7.3%	8.2%	0.8%
IO drivers	6.7%	5.9%	6.4%
scheduling	15.8%	14.3%	8.4%
architecture	22.8%	20.3%	23.9%
kernel	12.8%	15.7%	17.9%
memory management	4.5%	7.5%	4.9%
IO logic	4.7%	4.7%	11.7%
miscellaneous	6.1%	5.8%	5.4%
Total	**100%**	**100%**	**100%**

Table 5.1: Breakdown of the CPU cycles spent in various routines and operations. Key performance gains of FlashNet comes from saving the cycles in scheduling, storage, and spending more time in IO logic processing logic routines.

3. *FlashNet RDMA read:* Here, the server application *prepares* the flash-backed file ahead of RDMA accesses. Clients use one-sided RDMA read operations to read data directly from the flash device. This approach completely eliminates any file system and application involvement from the data and the control loop processing of an IO request. (see Figure 5.1(c))

Figure 5.2 shows the peak IOPS delivered with a single core at the storage server in the aforementioned three configurations. Both the socket with file IO and the sendfile mechanism approach deliver 65-70K peak IOPS/core. In contrast, under similar circumstances FlashNet is able to deliver up to 98K IOPS/core, which is 40-50% better than the other two approaches. While delivering the peak IOPS to 128 clients, the CPU core is fully utilized at 100% in all three configurations.

To further quantify the overhead, in Table 5.1, we break the CPU cycle usage down into the following categories:

- *network:* RDMA, TCP, IP, Ethernet processing, SKB management related routines.
- *storage:* the file system, the generic VFS, and the block layer related routines.
- *IO drivers:* network and flash device drivers routines.
- *scheduling:* scheduling, and context switch related functions.
- *architecture:* spinlocks, IRQs, atomics, bitops, data copy, etc.
- *kernel:* kernel routines timer, workqueues, softirqs, etc.
- *memory management:* memory management related functions.
- *IO logic:* application or FlashNet related routines.

- *miscellaneous:* miscellaneous routines that we cannot classify.

The three dimensions that stand out from the table are storage, scheduling, and the IO logic. The storage column includes routines from the ext4 file system, the generic VFS layer, and the block layer, etc. Most of these routines are executed for *every* network-storage request in the configurations (a) and (b). Furthermore, the current IO stack architecture experiences very high context switching and scheduling related overheads. This is because most of the IO resources (e.g. network sockets, memory buffers, etc.) are tied to the process abstraction. These resources need to be valid during the IO processing and hence require the application process to be scheduled for IO processing and data movement orchestrations. As a result, not many CPU cycles are left for actual application processing which can lose up-to 1/3^{rd} of the potential peak performance.

Multiple efforts tried to deal with the issues related to excessive OS and application involvement in either network or storage IO flows in isolation. For example, the use of Remote Direct Memory Access (RDMA) has been proposed to completely eliminate OS and applications from the network data flows. On the storage side, projects such as Moneta-Direct [71] and FusionIO's ioMemory SDK [135, 136] eschew the OS in the data access path in the favour of a leaner and faster access to flash storage. The FlashNet architecture is built on similar principles and goes a step further by building an *end-to-end* data path for network-storage transfers where applications and file systems are eliminated from the data path.

5.2 Design of FlashNet

FlashNet is a unified software stack that consists of three logical components, namely a flash controller, a file system, and an RDMA controller[1]. These components work together to eliminate IO inefficiencies in an *end-to-end* manner when data flows between a remote flash device and a client buffer. Figure 5.3 shows the setup and interaction among these components. The design of FlashNet is guided by three principles:

1. *Eliminate application involvement from IO flows:* As identified before, overheads from direct and indirect application involvement limit data flows. FlashNet leverages the path separation philosophy of RDMA networks and extends it to storage devices to completely eliminate an application's involvement from data transfer orchestration in an end-to-end manner.

2. *Reduce storage overheads:* FlashNet eliminates the file system and much of the generic OS and the VFS code from the extended data transfer path by designing a file system that uses

[1]We use separate controller names to highlight their roles in the overall FlashNet architecture.

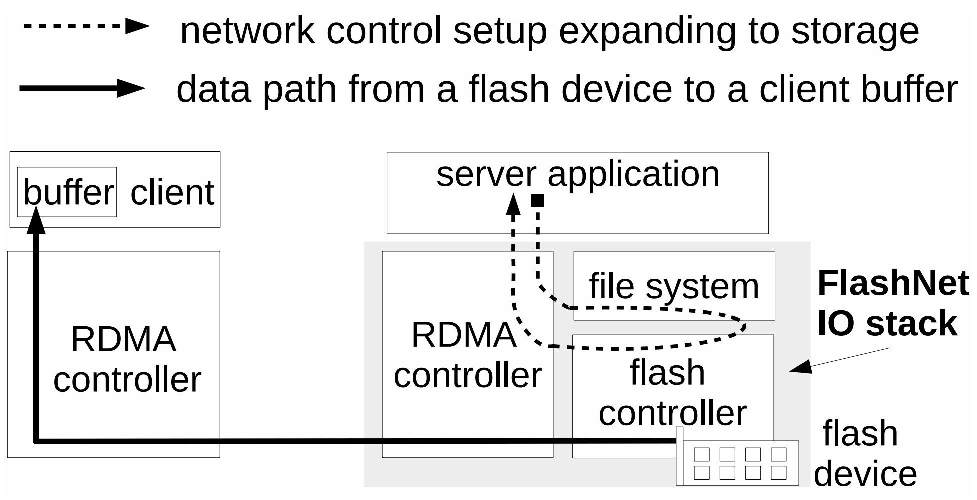


Figure 5.3: FlashNet stack illustrating the local network-storage control setup path and the *end-to-end* data flow path between a flash device and a client buffer.

a simple file layout. Hence, the file system and associated actions (e.g., inode look ups, checks, location translations, etc.) are eliminated from fast data paths between the network and storage controllers.

3. *Keep application interface simple and clean:* FlashNet extends the basic file and `mmap` based IO interfaces to provide *RDMA-ready memory-mapped* files as the basic IO abstraction. Using existing standard mechanisms, FlashNet plugs storage buffers into the application address space and lets the application manage them in a unified manner with other DRAM buffers.

5.2.1 The Flash Controller

A key part of the FlashNet architecture is its flash controller design. A flash controller manages the performance and packaging idiosyncrasies of flash devices. However, prevalent embedded flash controller designs are too restrictive for the FlashNet architecture due to multiple reasons. First, with high-speed networks, a flash page containing hot data may experience bursts of concurrent small writes from the network within a small time frame (a few μsecs). Even though the networking stack contains pertinent information, which could be useful to absorb the bursty nature of network IO, there is no standard way to pass this information to the flash controller for better flash management. Second, flash devices are exposed as conventional block devices where the logical block management is tied with the flash storage management. Previous research in the field has demonstrated that decoupling these two can lead to performance improvements with a much simpler file system layout [181]. A simpler file layout enables removing the file system from network data transfers. And lastly, the one-controller-per-device design cannot leverage multiple flash devices present in a system.

To alleviate the above mentioned restrictions and to jointly optimize the flash controller with the rest of the stack, we have designed and implemented a software-defined flash array controller that builds on top of virtualized flash storage works from Weiss et al. [356] and Josephson et al. [181]. Our controller decouples the logical block management from the flash storage management and exports a large 64-bit block address space using a virtualized Flash Translation Layer (FTL). The FTL dynamically maps flash logical block addresses (LBAs) to physical block addresses (PBAs) over a virtual device array made out of one or more flash devices. An LBA entry in the FTL contains the location of the data on a flash device and its location in a DRAM buffer (if the data is present in the system). This design ensures that all concurrent accesses are given the same DRAM page or PBA locations. As required for RDMA accesses, data in the block address space is accessed through a native *byte-addressable* get/put interface (Table 5.2). Furthermore, the get/put interface also provides the necessary heat and frequency information to the flash controller for better flash management.

Dirty data is always written out-of-place while keeping track of new LBA to PBA mappings in the FTL. Updates to the FTL are appended to the flash device asynchronously with the data and are synced when instructed by an application or a remote RDMA access. The controller uses a log-structured allocation strategy to allocate PBA blocks across multiple devices. It ensures uniform wear leveling, and employs advanced data placement and efficient garbage collection (GC) policies to reduce write-amplification. Under non-uniform (i.e., skewed) workloads, the controller segregates data into data streams based on their update frequency to reduce data relocation overheads [160]. Ideally, a number of data streams equal to the update frequencies that the workload exhibits should be chosen. In practice, however, the supported number of data streams is limited by hardware or metadata resources. Our controller performs a three-level data segregation scheme based on (a) the logical origin of data blocks; (b) their age in the system; (c) their frequency and heat of updates (provided with the help of the RDMA network controller). The GC policy is a greedy policy augmented with a recurring write pattern detection that will not evaluate a block for relocation if it is expected to see more overwrites due to sequential writes, for example.

5.2.2 Contiguous File System (ContigFS)

Files and file systems have been the de-facto standard of saving data on storage devices since the inception of UNIX in the 1970s. However, As storage devices get faster, the constant and unnecessary involvement of a file system in every aspect of IO (local or networked) operations generates a significant amount of overhead [68, 71, 201]. In the IO path, one of the key actions that a file system executes is the translation of a file offset to a device block location. With the current extent-based file layouts, it is not possible to eliminate or even reduce the file system involvement from the IO path due to their sophisticated extent management logic. However, with the virtualized 64-bit FTL address space, we can vastly simplify the file system design by using a range-based rather than an extent-based file layout. A range-based file layout stores a complete file in contiguous device block addresses. This layout enables a trivial file offset to device location translation by adding the file offset to the start location of the file on the device. This translation can also be done by the network controller, hence removing the file system from the networked IO. As an alternative, raw block IO can be used to remove the file system from the IO path. However, this option also eliminates other highly desired file system properties such as hierarchical naming and access control.

To realize our idea, we design a POSIX file system called Contiguous file system or ContigFS that does contiguous file allocations on top of the virtualized FTL address space. The files that are stored in a contiguous LBA address range, can grow, and shrink by manipulating their mappings in the FTL address space. The design of the file system and metadata management for hierarchical namespaces is done in a very similar spirit to the Direct File System [181]. A large segment (one TB) from the beginning of the virtualized FTL is reserved for the file system metadata and directory layouts. The files are allocated after this segment in the virtualized FTL.

Like any other file system, ContigFS provides the full file system API to applications. ContigFS files are *RDMA-ready* and can be *memory-mapped* by using the familiar `mmap` call. The use of the virtual memory abstraction is easier than the separate two-tier (memory and storage) approach towards data management. Many in-memory applications already do their own memory management. Acquiring memory through a file-backed memory region is an intuitive extension to the process. For example, MongoDB manages data in a single unified virtual address space using file-backed `mmap` calls [4]. Memory segments obtained from file `mmap` calls can be used to provide durability as well.

In absence of flexible APIs to manage the kernel page cache, ContigFS co-manages (with the flash controller) its own pool of DRAM pages to exert full control over data staging, sharing, and management logic to/from flash devices. Data is staged for access and dirty data is written out from pages in the DRAM pool. Pages from this DRAM pool are also given to serve the page faults in mmap'ed memory regions. In a similar spirit to the IO-lite system [269], ContigFS ensures (in a collaboration with the flash controller) that there is only a single physical and consistent copy of data in the system that is shared between the storage controller, the network

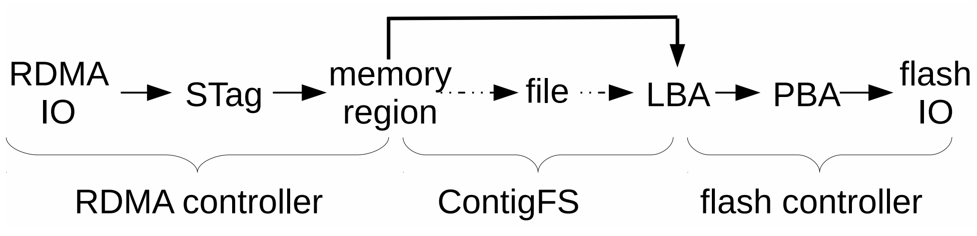


Figure 5.4: Stitching of the IO path abstractions from network to flash while eliminating the file system from the data path.

controller, and applications.

5.2.3 The RDMA Controller

The RDMA controller of FlashNet extends the data and control paths [352, 358] of RDMA networks to include the file system and the flash controller as well. Similarly to the original path separation idea, applications, the file system and to a large extend the OS, are eliminated from the extended data flow path. In order to achieve this, FlashNet set ups necessary abstraction translations in advance from an RDMA access to the data location on a flash device.

A key operation on the extended control path is the RDMA buffer registration process. In the buffer registration process, every data source or sink buffer is pre-registered with the RDMA stack to generate a buffer identifier called Steering Tag or STag. This STag is used in subsequent RDMA operations to identify network source or sink buffers without involving the application to steer data flows. FlashNet uses the same mechanism to identify files and offsets to resolve data locations which are involved in a network operation. ContigFS files, which are involved in RDMA network operations, are registered with the FlashNet RDMA controller by passing their mmap'ed area. At this point, with the help from the ContigFS, the RDMA controller translates the memory area to the start LBA of the file. As files are contiguously allocated in the LBA address space, further offset calculations during RDMA network operations are done entirely by the RDMA controller and then passed to the flash controller for reading/writing data from/to involved LBAs. Hence, on the fast data path, the file system is eliminated and the two device controllers talk to each other to manage data flows. Figure 5.4 shows the end-to-end translation process between these abstractions on the extended control and the data path.

The traditional RDMA buffer registration scheme, where pages are immediately pinned, does not allow scaling beyond the system DRAM size. In order to support TBs of flash storage [134] with GBs of DRAM, the RDMA controller of FlashNet supports lazy memory registration. During the lazy memory registration (happens only for memory segments backed from a ContigFS file), the RDMA controller only allocates necessary metadata, locks, and data

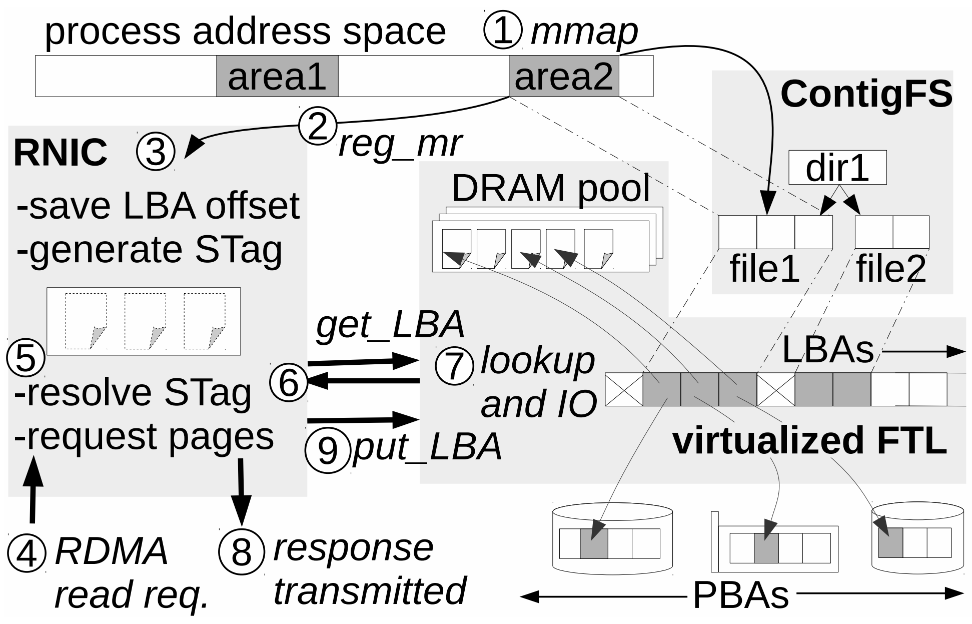


Figure 5.5: The life of a FlashNet net-storage operation. Steps 1–3 and 4–9 constitute the extended control and the extended data paths in the FlashNet stack, respectively.

structures to hold DRAM page pointers, but does not pin pages. The pages are populated on demand on the extended data path when an RDMA request accesses them. These pages are also shared concurrently (without creating copies) between an application, the network, and the storage stack using RDMA buffer ownership rules.

The use of the RDMA API further brings byte granular, low-latency, high-bandwidth flash accesses into the Remote Memory Access programming model [158]. As we discussed in the last chapter, many of the key performance properties, such as userspace-mapped IO queues, batched IO requests, asynchronous IO processing, synchronous polling for completion notifications, etc., have been explored and shown to be useful in the context of local flash accesses as well [71, 273, 313, 364]. FlashNet provides a high-performance IO interface around these unified RDMA properties [341]. This holistic approach collapses the rigid layer structure and thus, efficiency is gained by creating fast, non-blocking, asynchronous end-to-end data flow paths.

5.2.4 The Life of a Unified IO Operation

To demonstrate how various components come together, in Figure 5.5, we present an example where a server process serves data to a client from a file stored on ContigFS using an RDMA read operation. The server process starts by mmaping the file into its address space (step 1). Upon receiving the `mmap` call, ContigFS does sanity and permission checks of this mapping. The server then registers the mmap address with the FlashNet RDMA controller to prepare it

```c
/* open a file from mounted ContigFS */
int fd = open("/mnt/contigfs/file0", flags);
if (fd < 0)
  goto open_fail;
/* map the file into the process AS */
void *faddr = mmap(NULL, len, prot, flags, fd, off);
if (faddr == MAP_FAILED)
  goto mmap_fail;
/* register this address */
struct ibv_mr *flash_mr = ibv_reg_mr(pd, faddr, len, access);
if (flash_mr == NULL)
  goto reg_fail;
/* initialize rdma_msg using flash_mr */
struct ibv_send_wr rdma_msg={...};
/* post the rdma_msg on the qp for IO */
ret = ibv_post_send(qp, &rdma_msg,...);
return ret;

/* Error handling */
reg_fail: munmap(faddr, len);
mmap_fail: close(fd);
open_fail: exit(-errno);
```

Figure 5.6: Example of RDMA code using the FlashNet stack.

for RDMA operations (step 2). The controller resolves the passed region to be a ContigFS-file region and hence, only translates mappings and saves the LBA address of the memory region by adding the mmap offset to the starting LBA address of the file (step 3). The controller then generates a valid STag and returns it to the server process, which distributes the Stag to clients (not shown). Steps 1–3 constitute the extended control path in FlashNet.

On the extended data path (steps 4–9), upon receiving an incoming RDMA read request, the RDMA controller first resolves the target memory buffer using the STag present in the request. The controller then calculates the LBA address of the request by adding the offset (present in the RDMA request) to the previously saved base LBA address of the registered region (step 5). The flash pages in the identified region are then populated with the help of the flash controller (steps 6 and 7). Upon completion of the RDMA request processing (step 8), the involved LBA pages are given back to the flash controller (step 9).

5.3 Implementation of FlashNet

The components of the FlashNet stack are implemented as kernel modules in Linux kernel version `3.13.11`. The implementation of the flash controller is based upon the SALSA software flash controller [174] which is written from scratch using the Linux kernel device mapper framework. The SALSA controller is extended to support in-place DRAM page-sharing among multiple entities. ContigFS is implemented as a standalone kernel module that hooks into the VFS layer of the Linux kernel. FlashNet uses the open-sourced SoftiWARP RDMA controller [236, 339] to provide RDMA capabilities while maintaining the compatibility with current RDMA RNICs enabling hybrid hardware-software deployments. SoftiWARP was enhanced to interact with ContigFS and the flash controller to support lazy memory management. The unified RDMA software device appears as an RDMA device within the Linux RDMA OFED framework [16]. The whole FlashNet framework does not require any changes to existing infrastructure code including the kernel and drivers.

RDMA-ready applications require minimum changes to access data from a remote flash device using RDMA operations. To enable RDMA accesses on remote files, applications must acquire memory from `mmap`ing ContigFS files. The code snippet in Figure 5.6 shows how applications can obtain RDMA-ready flash buffers using `mmap` operation on a file from ContigFS (lines 2–6), and register the obtained memory area with the FlashNet RDMA controller (line 10). After the registration step, there are no application-visible differences in the use of the registered memory region (mr) (lines 13–16), represented by `flash_mr`. A server application can distribute the STag obtained by this method to clients for remote RDMA read and write operations to the flash device.

In the following sections, we describe the extended control path, the page population and RDMA processing operations on the extended data path, and the synchronization guarantees that are given by the system.

5.3.1 The Extended Control Path

In line with the path separation philosophy, a FlashNet application must first create files (if not already present), `mmap`, and register file-backed mmap addresses on the extended control path ahead of remote accesses. The extended control path is implemented in the file system and the RDMA controller.

5.3.1.1 Files and `mmap` Management

ContigFS manages all file management related operations. It splits the virtualized FTL address space to save file system metadata and file data separately. When a newly created file is given a valid size via a `ftruncate` call, ContigFS allocates a contiguous LBA range on the virtualized FTL to save the file. Our file system implementation is very similar to the Direct File System [181], however, with the current prototype we do not reserve LBA ranges to provide

API functions	**Description**
check_vma(va, len)	resolves a ContigFS `mmap`'ed area
get_LBA(lba, len, flags, cb)	gets DRAM pages
put_LBA(lba, len, flags)	puts pages in a LBA range
sync_range(lba, len, flags)	writes data and/or the FTL to flash

Table 5.2: The flash controller API for data management.

large unassigned LBA ranges to files. The size of an LBA block is configurable (default is 4kB). A file can grow and shrink by `realloc`ing its LBA address space area. And if necessary, files can be relocated in the FTL address space without physically moving the data on the devices by updating the FTL mappings of the new LBA range to the old PBA entries. The old LBA space is marked free and managed using the buddy memory allocation technique [194].

An `mmap` call from an application is relayed to ContigFS with a valid virtual memory area (VMA) within the process address space by the Linux kernel. With this VMA, ContigFS registers itself as the page fault handler for this area to provide the same DRAM pages to the application as seen by the network. At this stage, ContigFS does sanity and permission checks for the `mmap` call. Since DRAM pages are shared, the current implementation of ContigFS only supports the `MAP_SHARED` flag.

5.3.1.2 RDMA Lazy Memory Registration

To generate a valid STag, the application registers the VMA provided by the `mmap` call to a FlashNet RDMA device (enumerated by the `rdma_get_devices()`, a regular RDMA API call). Upon receiving the registration request, the FlashNet RDMA controller checks if the passed VMA range belongs to a ContigFS file. In that case, the RDMA controller translates the passed virtual address to a file start LBA address (by walking on the `vm_file` and `f_mapping` structures). The mmap offset is then added to the start LBA of the file and saved as the start LBA of the mmap'ed memory region. The RDMA controller then allocates all necessary per-VMA meta-data, and generates a valid 32-bit STag, but does not pin pages by calling `get_user_pages().`

The controller maintains its own view of memory mappings outside of the flash controller's FTL mappings. This design ensures that the same ContigFS file pages (or even a single byte within the same page), which are registered with different RDMA semantics, are treated properly during concurrent RDMA accesses. Furthermore, by doing so, it reduces pressure on the FTL management logic and the RDMA controller can use more effective data structures that are best suited to its needs. These VMAs are populated on-demand, in a lazy manner when an RDMA operation accesses them using the get/put API provided by the flash controller.

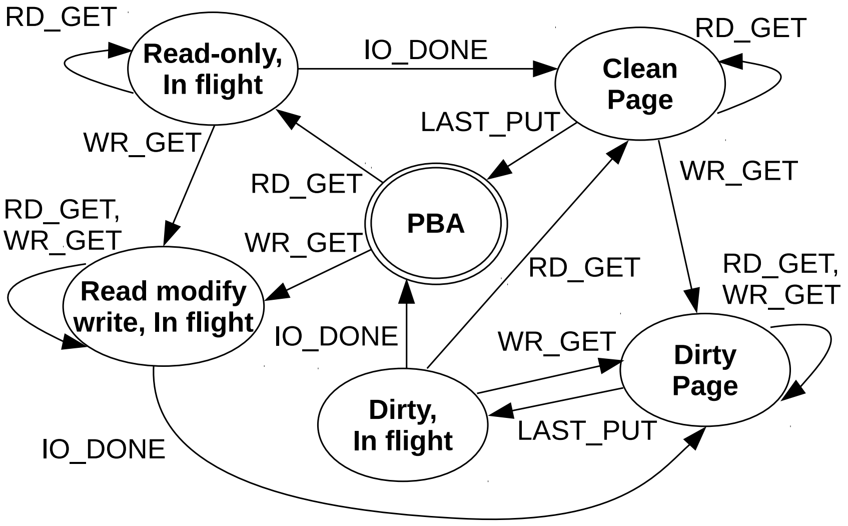


Figure 5.7: The state machine of a flash LBA page.

5.3.2 The Extended Data Path

On the extended data path, the flash and RDMA controllers interact using simple non-blocking `get_LBA()` and `put_LBA()` interfaces (Table 5.2)) to lazily populate pages in ContigFS-backed VMAs. As pages become ready for processing, callbacks are issued on per-page basis. The same interface is also used by ContigFS to serve page faults in `mmap`'ed areas. With the early identification of IO buffers via STag, FlashNet's IO processing path has no stalls (no context switches, or scheduling points) with *run-to-completion* type processing to deliver high performance [44].

5.3.2.1 Processing of get/put LBA Requests

Although the get/put API of the flash controller is byte addressable, these requests are broken down internally into the flash page size (typically a multiple of the system page size, default 4kB) for processing. To ensure that no two entities in the system see different data, the flash controller implements a state machine with atomic transitions for every flash LBA page. The state of an LBA page is stored with its FTL mapping. Figure 5.7 shows the LBA page state machine executed by the flash controller to resolve its status.

An uninitialized flash LBA page starts in an `Invalid` state. At the time of the first writing, the controller picks a PBA address, maps it to the LBA page, and updates the LBA entry in the FTL from a `Invalid` to a `PBA` state (not shown). For any read (`RD_GET`) and write (`WR_GET`) get requests for a LBA range from the RDMA controller, the flash controller checks the state of the LBA pages involved to determine if a page has previously been brought into the system (i.e. contains an LBA entry in the FTL that points to a DRAM page address or a device PBA). For flash LBA pages that are not in the system DRAM (indicated by the `PBA` state), the flash controller issues DMA requests for them. This action results in moving of the pages into a `Read-only In-Flight`, or `Read-Modify-Write In-Flight` status for read and write requests, respectively. When the DMA is finished (`IO_DONE`), and the data of a flash page is brought into a system DRAM page, the DRAM page is atomically installed in the FTL and the LBA state is changed accordingly. After this transition, asynchronous callbacks are issued to the RDMA controller to notify the page status change. Any subsequent read or write get requests on this page will be given the same DRAM page location while maintaining the usage counter on a per-page basis. These counters are used by the garbage collector to identify the hot and cold LBA pages.

For LBAs which were already in DRAM, the flash controller immediately issues callbacks to the RDMA controller with a valid DRAM page pointer. Consequently, depending upon the status of the pages involved in a request, callbacks can be issued in any order. These out-of-order callbacks enable the flash controller to (1) finish a `get_LBA()` request as soon as possible without stalling on the first unavailable flash page; (2) leverage parallelism in the flash hardware by issuing requests to parallel devices/ports; (3) efficiently recycle DRAM pages between get and put calls on a page granularity.

After network processing, the RDMA controller issues `put_LBA` to put down references on the pages involved. Concurrent small network writes are absorbed by the same DRAM page, and only the last put call triggers a dirty data write out. The current flash controller prototype does not cache dirty data. A selected form of time-bounded caching is done for pre-fetching. The pre-fetching logic is similar to the `get_LBA()`, but without the callbacks.

5.3.2.2 Processing of RDMA Requests

The FlashNet's RDMA controller (based upon SoftiWARP) provides complete iWARP RDMA features, semantics, and operational ordering. Applications post RDMA TX and RX requests and get IO completion notifications on memory-mapped RDMA queues which are maintained in the kernel. FlashNet uses non-blocking kernel TCP sockets with per-core kernel threads to perform asynchronous network IO on behalf of a user process. As a further optimization, FlashNet uses RDMA fragment header information to pre-fetch flash pages to hide the high (w.r.t. network) access latencies of flash devices.

The network processing for flash data can be stalled if a flash LBA is not ready. In that case, a `get_LBA` request is placed, and the connection is marked stalled. A stalled connection is resumed after receiving callbacks form the flash controller. As explained before, these callbacks happen in any order. However, FlashNet's RDMA semantics implemented on top of TCP sockets necessitate in-order data delivery and processing. This means that data in an application buffer is filled in a sequential increasing order from the lower bound to the upper. Since RDMA networks do not re-order packets (when presenting to an application), the sender must also send data in a sequential increasing order. This network IO semantic inhibits us to send flash pages if they do not complete in order. For example, for a 3-page flash IO request, data from pages 1, 2, and 3 must be transmitted in this sequence. If pages 2 or 3 finish before 1, then the network transfer cannot start and must wait until page 1 is done. This strict processing order introduces unnecessary delays even when some data is ready for transmission.

As a first solution to this problem, we thought to keep track of page ordering in an additional RDMA header. However, we quickly discovered that this approach has multiple disadvantages. First, as the number of callbacks are variable (depend upon the number of pages involved), it requires a memory allocation to keep track of callbacks on a per-connection basis, which goes against the principle of path separation in RDMA. Second, the out-of-order delivery of data will introduce a complicated tracking of the current IO status on the receiving side. Lastly, it breaks the compatibility with the current RDMA hardware. Hence, we opted for the *out-of-order* `get_LBA()` completion with *in-order* network processing in FlashNet.

For in-order data processing, we take advantage of the fact that RDMA requires pre-allocation and registration of flash VMAs. At the time of registration, FlashNet allocates an atomic counter on a per-page basis to store the validity of the page. In a callback, this counter is increased and before calling a put on the page, the counter is decreased. FlashNet checks the *readiness* of the region by scanning for the longest sequence of non-zero atomic counters and only processes data in that ready region.

RDMA Transmission Path: FlashNet uses non-blocking kernel sockets with per-core kernel threads to perform asynchronous data transmission on behalf of a user process. As the transmission path is synchronous (i.e. the total transmission size of an RDMA message is known at the beginning), the thread prepares the complete flash message buffer VMA in one go. It then scans the flash area to prepare a full message fragment (typically the `tcp_mss` size). The ready flash data (minimum of `tcp_mss` size, RDMA message size remaining, and ready flash data) is then pushed to the kernel socket.

The transmission processing can be stalled if (a) the write space of the socket is exhausted and `-EAGAIN` is returned; or (b) if a not-ready flash page is encountered (indicated by zero counter). In the first case, the Linux kernel TCP stack issues a callback (`sk_write_space` on `struct sock*`) when more socket write space is available. In the latter case, the callback is issued from the flash controller when a flash page becomes ready. These callbacks resume the TX processing, and the transmission loop is repeated with checking the readiness of the flash data buffer to prepare the next RDMA message fragment.

One outstanding issue on the transmission path is when to put down the reference on a flash page. The Linux TCP stack does not expose the point up to which data has been transmitted. In the case when `kernel_sendmsg` is used, involved flash pages can be freed immediately. In the case of a zero-copy transmission using the `tcp_sendpage` interface, FlashNet periodically probes the state of the TCP connection, and calculates how many bytes have been transmitted successfully based on the TCP sequence and acknowledgement numbers [133]. The involved DRAM pages are then put back to the flash controller. However, this feature remains experimental and is not evaluated in the thesis.

RDMA Receive Path: Unlike the transmission path, the receive path of FlashNet is asynchronous where the total incoming RDMA message size is not known apriori. The flash buffer preparation is done in a response while more message fragments are received. The FlashNet receive-side processing happens in the TCP softirq context in the `sk_data_ready` upcall by calling `tcp_read_sock()`. After receiving a message fragment header, FlashNet extracts the STag and resolves the flash buffer area. It then proceeds to prepare the area to receive the current fragment payload (denoted by length and offset of current SKB). After the preparation, it checks the readiness of flash pages to receive the message fragment. It then copies the ready data (minimum of fragment size and flash ready pages) using `skb_copy_bits()` and reduces the reference count for the pages involved. This eager action of putting flash pages allows FlashNet to continue receiving data in a flash region that is bigger than the system DRAM capacity. If all flash pages are ready, then the RX processing finishes in the TCP softirq upcall. If some or none of the flash pages are ready, then, after copying the appropriate amount of data from the SKB, the RX processing is marked stalled and the RX context is queued on the wait list.

The RX processing is resumed on a kernel workqueue that synchronizes with softirq upcalls to receive data. The kernel workqueue is scheduled when more flash pages become ready. The receive processing path of the kernel workqueue is very similar to what we have described before for softirq upcall. Upon receiving the last fragment of an RDMA message, the receive context is marked complete.

5.3.3 Data Synchronization

To allow a remote client to wait until data is written on a flash device, FlashNet defines semantic STags. A semantic STag uses the current STag structure, but allows applications to use the lower 8 bits of the 32 bit structure in a special way. These lower 8 bits are masked by the

CPU	Dual Xeon E5-2690, 2.9 GHz cores
DRAM	128 GB, DDR3 1600MHz
NIC	Chelsio T5 Ethernet 40Gbits/sec
Flash	1.3 GB/sec (read), 680 MB/sec (write) peak read IOPS: 360K, Chip latency: 50μsec
Switch	IBM RackSwitch G8332, 32x40GbE ports

Table 5.3: 9-machine testbed configuration for FlashNet.

application using `sync` or `async` flags provided by FlashNet. With this extension, the `sync` RX processing path of FlashNet does not process further incoming RDMA write operations until the previously pushed data is written to the flash device. `async` processing continues with the RDMA write processing as soon as DMA requests are queued. Irrespective of how the processing of an incoming RDMA write is done, the client-side RDMA queues always remain asynchronous.

5.3.4 On-Going Efforts

Persistent RDMA Credentials: RDMA semantics are currently defined within the scope of a process. When a process dies, the associated RDMA resources are destroyed. However, the FlashNet stack brings persistence to RDMA buffers. Hence, persistent storage services built on FlashNet, such as a distributed file system or an object store, may require persistent RDMA credentials. We are investigating how to preserve or regenerate RDMA credentials (registered flash regions, STags, etc.) across a process or the system restart.

Current RDMA semantics: From its design choices, the current FlashNet prototype is bound to the RDMA protocol and semantics. RDMA up to now, however, is defined for volatile addresses only. Consequently, RDMA does not provide any global ordering, consistency, and synchronization guarantees in presence of concurrent accesses or failures. More specifically, ordering guarantees for writes (in case of power or software failures) while being able to exploit the internal parallel of flash devices is a challenging problem. New hardware primitives [92] and mechanisms [347] are being proposed for directly-attached NVM devices in presence of caches and multicore CPUs. We envision similar facilities for remote NVM accesses. Hoefler et al. [158] discuss many of the coherence and consistency challenges associated when accessing remote flash storage in a remote memory access (RMA) setting. FlashNet is a first working step in this direction and we plan to tackle these issues in our on-going work.

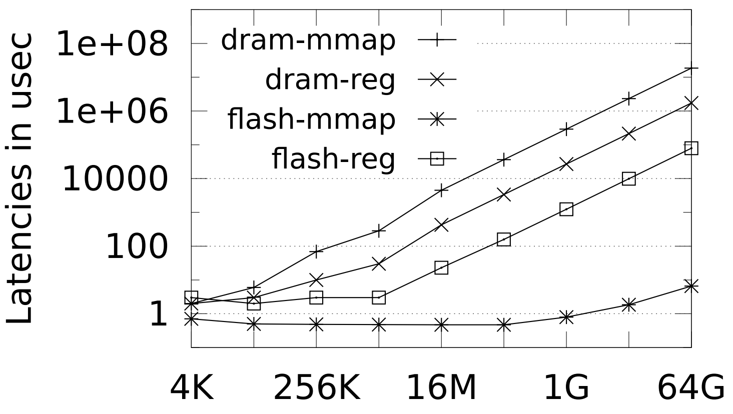


Figure 5.8: mmap and buffer registration cost.

5.4 Evaluation

We evaluate FlashNet on a cluster of 9 machines connected via a 40 Gbits/sec network (Table 5.3). On one machine that contains an enterprise-level PCIe-attached flash device, we run the FlashNet stack. The remaining machines run networked clients that repeatedly ask the server for data from the flash device. The clients use unmodified SoftiWARP to connect to the FlashNet devices at the server (except for Section 5.4.5 where the clients use real RDMA hardware). All numbers reported here are measured on the Linux kernel 3.13.11 and are the average of 3 runs, each lasting 60 seconds. FlashNet experiments are done with the files on ContigFS while other configurations have used the ext4 file system. We have also repeated the experiment with a NVMe-attached flash device and found no significant deviations in our findings.

The key performance highlights are:

- FlashNet delivers up to 50% more IOPS per core than the traditional way of transferring data, which involves a mediating application orchestrating data transfers between the IO stacks. Peak IOPS performance of FlashNet (346K) is within 96% of the flash capacity.
- By eliminating unnecessary OS and application involvement, FlashNet helps to reduce the network access latency of a 4kB flash page by 20–44%.
- By efficiently managing the flash device, FlashNet reduces the write amplification by 38–83% under skewed write workloads.
- In a hybrid FlashNet-RNIC setup, the flash-to-DRAM data transfer latencies are 40–45% better than those of the traditional interfaces.

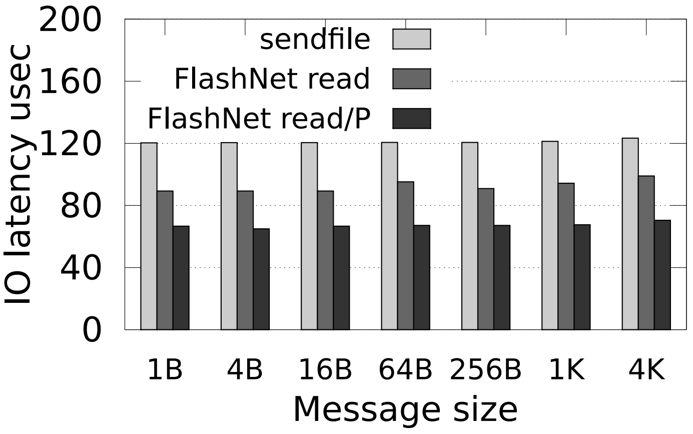


Figure 5.9: Read latency of net-storage operations.

5.4.1 The Cost of Flash Buffer Registration

We first evaluate the cost of a flash buffer registration. The key difference between the standard RDMA memory registration and the FlashNet registration is the lack of page allocation and pinning costs from the latter. However, per-page metadata, locks, states, etc., are still allocated. Figure 5.8 shows the cost of `mmap`ing a file on ContigFS, as well as the cost of registering the obtained mmap area with the FlashNet RDMA device. Naturally, FlashNet's buffer registration is faster than the corresponding DRAM buffer registration. We only show anonymous DRAM buffer registration to exclude the cost of flash IO for ext4 file `mmaps`. FlashNet can register and prepare 64GB of flash area in 85μsecond (6μsecs for mmap, and 79μsecs for registration). In contrast, a DRAM buffer registration takes almost 20 seconds, where the majority of the time is spent in the mmap call (zeroing the buffer by the kernel). At the moment, the cost of the buffer preparation is proportional to the number of 4kB pages involved. We plan to support multiples of huge pages (2MB) in the future.

5.4.2 Single Client Performance

In this section, we evaluate the bandwidth and latency of network-storage read IO operations from the perspective of a single client in a single core setup. The client has only one outstanding request at a time. Figure 5.9 shows the latency numbers for three different configurations, namely `sendfile`, FlashNet read, and FlashNet read/P when it is optimized for the low latency IO using polling. On our setup, the three configurations of `sendfile`, FlashNet, and FlashNet with polling transferred a 4kB page in 123.4, 99.1, and 70.4 μsecs, respectively. We could not measure the socket using the file IO configuration because the direct IO mode does not allow reading less than the sector size (512B). But for 4kB reads, the socket with file IO

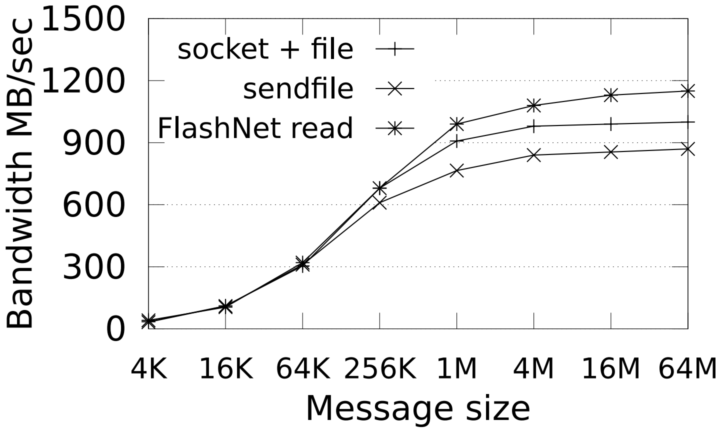


Figure 5.10: Bandwidth of read net-storage operations.

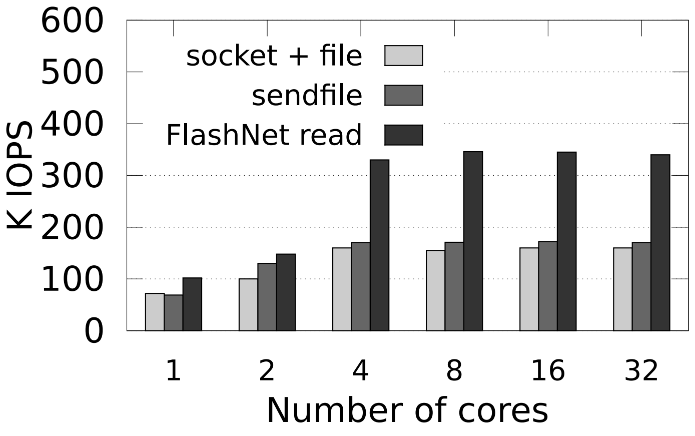


Figure 5.11: 4kB IOPS scaling with number of cores.

configuration shows similar performance numbers as the `sendfile`.

Figure 5.10 shows the observed bandwidth number for various IO sizes. As the IO transfer size is increased, FlashNet performs better than the other two configurations. Around a buffer size of 32MB, FlashNet reaches the device read bandwidth limit.

5.4.3 CPU Core Scaling

In this section, we revisit our key experiment from section 5.1 and evaluate the effect of increasing numbers of CPU cores. We kept the number of concurrent clients constant at 128. Figure 5.11 shows our results. All three configurations scale between 1 and 4 cores and then stop scaling further. FlashNet's performance of 346K IOPS is very close to the device limit of 360K IOPS, and is achieved with using only 4 cores. However, the other two approaches do not

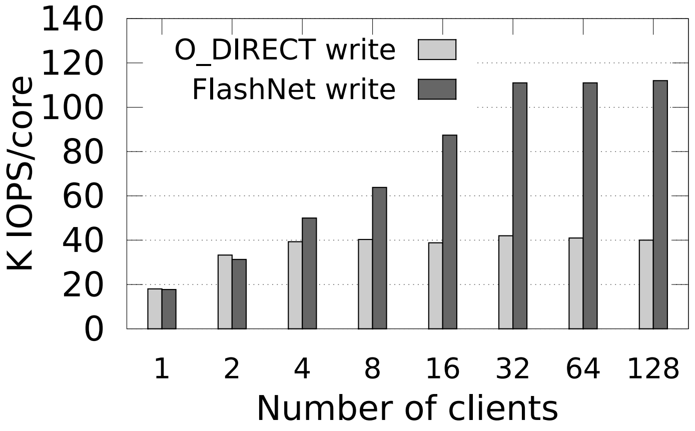


Figure 5.12: Peak 4kB synchronous write IOPS/core.

scale beyond 170K IOPS despite increasing the core count to 32.

5.4.4 Write Performance

In this section, we present the write performance of FlashNet in the presence of *synchronous* write operations where networked clients do not issue a next write request until the last write has been written out to the storage. We compare the performance of FlashNet against a `O_DIRECT` direct write to the ext4 file on the flash device. A 4kB synchronous write took 53μsecs for a direct write, and 56.1μsecs for a FlashNet write. We suspect this additional cost comes from our write processing path in the interrupt context, which in case of a synchronous write requires context switching to a kernel thread to wait for the IO completion. For 4kB asynchronous networked writes, however, FlashNet schedules the IO writes immediately, in which case the write completes in 38μsecs.

Figure 5.12 shows the scaling of peak write IOPS as we scale the number of concurrent clients. Clients only have one outstanding request at a time, and the server runs on a single core. At 128 clients, FlashNet delivers almost $3\times$ more IOPS per core for a ContigFS file than a synchronous direct write on the ext4 file. We also measured the performance of a `fsync` syscall and observed a performance that is worse than the one of direct file writes.

Figure 5.13 shows the write bandwidth as we increase the data size. FlashNet consistently outperforms direct writes to ext4 files by a margin of 30–70%. It reaches the device write bandwidth limit with 64kB buffers.

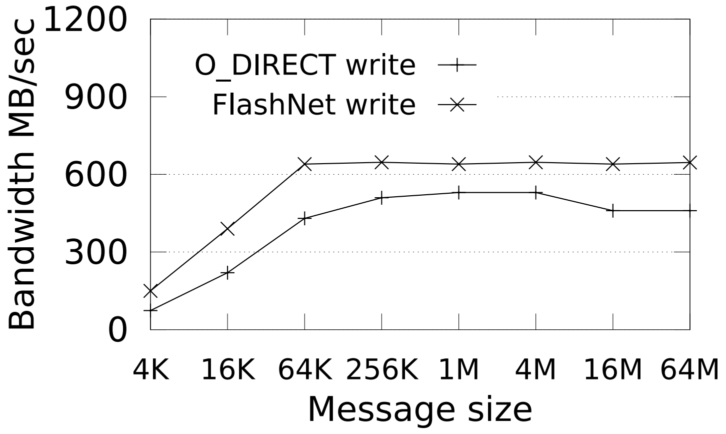


Figure 5.13: Synchronous write bandwidth.

	TX-side	RX-side	4kB	Diff.
Read	FlashNet FlashNet	SoftiWARP T5 RNIC	70.4μs 60.9μs	13.4%
Write	SoftiWARP T5 RNIC	FlashNet FlashNet	56.1μs 53.7μs	4.2%

Table 5.4: IO latencies of various hybrid configurations.

5.4.5 Hybrid IO Configurations

One key advantage of keeping the iWARP packet format is that FlashNet is compatible with the current iWARP RNIC hardware. In this section, we evaluate this hybrid setup using Chelsio's T5 RNIC. We equip one of the clients with T5 RNIC and measure read and write latencies for a 4kB page. For a read configuration, FlashNet transmits data from a flash page to a client containing the T5 RNIC. For a write configuration, the T5 RNIC is used to transmit data from the client to the FlashNet server. As shown in Table 5.4, the use of a hybrid FlashNet-T5 setup improves IO latencies by another 4–13%.

5.4.6 Remarks about the Flash Management

The flash controller segregates data in three levels based on their heat, age, and origin during data placement. As described earlier in Section 5.3.2.1, the get/put API of the controller enables it to use the usage counters to extract a page heat (i.e., update frequency) information about a page. A high `get_LBA` count represents a hot data page. The data segregation and greedy GC scheme of the flash controller exhibit significant reduction in write amplification in the common case where the write access pattern exhibits skew. Under Zipfian write workloads of 80/20 (80% accesses to 20% flash area) and 95/20 (95% accesses to 20% flash area), the flash controller reduces write amplification by 38% and 83% and improves the flash lifetime by $1.6\times$ and $5.9\times$, respectively.

5.5 Related Work

In this section, we cover the large body of related work in the field of high-performance IO integration. Table 5.5 summarizes our discussion here.

Inefficiencies in end-host networking stacks have first been discussed in the 1990s and have led to the design of high-performance networking stacks such as U-Net [352] and Hamlyn [66, 358]. RDMA networks such as InfiniBand or iWARP are the latest incarnation of these principles and are being used both in supercomputers as well as in data centers running cloud workloads [107, 241, 264]. In parallel with these networking efforts, the storage community has proposed several works to eliminate inefficiencies in end-host storage stacks. These efforts include offloading permission checks to hardware [69, 71], shortening of the IO path by merging execution contexts [314], coalescing interrupt to reduce the CPU load [24], polling [313, 364], and speculation [355]. The recently proposed NVMExpress [10] SSD interface and the block layer optimizations for Linux [53] take a very network-alike approach (parallel queues, polling, IO descriptors, etc.) towards reducing flash access overheads. In a more system-wide approach, Exokernels [117, 182], and the recently proposed Arrakis [274] and IX [44] OSes contain high-performance IO stacks. These OSes eliminate overheads from the kernel by limiting its role to resource management only. Nonetheless, these isolated stack-specific efforts do not address end-to-end data transfer issues solved by FlashNet.

The easiest way to eliminate the datapath overheads when accessing storage is to use the `sendfile` interface. However, as demonstrated earlier, this interface still requires the application to be involved for cross-stack control loop transfers and only solves the transmission-side issue. Similar to FlashNet, BlueGene Active Storage (BGAS) [119, 304] work uses RDMA semantics and operations for accessing a *local* flash storage. Network-Attached Secure Disks (NASD) [142, 143] work from CMU eliminates many server-side CPU and the OS overheads by connecting storage disks directly with the network controller. However, it relies on custom disk and networking hardware, and does not reduce overheads at the client side. More traditional approaches, such as block IO in the form of SAN (e.g. QuickSAN [72]) or distributed file systems such as Strata/NFS [96], also provide fast access to data stored on remote non-volatile storage devices. However, their performance benefits are limited to the boundary of the provided IO abstraction. For example, QuickSAN implements SAN block access protocol and Strata provides a file interface on top of NFSv3. Hence, the data stops short (at the block or the file level) of being delivered in the final application buffer. FlashNet delivers data directly by establishing

Table 5.5: Comparison of related work for end-to-end cross-stack data transfers.

	Remote Flash Access?	**Application Involved?**	**Byte-level IO**	**High-Perf. Properties**	**Application Interface**	**Limitations/Comments**
`sendfile`	yes	yes	yes	no	file with socket	only deals with the TX side issues
`mmap` + `msync`	yes	yes	yes	no	shared buffers	application-driven IO placement
NASD [142]	yes	no	yes	no	dist. object store	only deals with the server-side issues
QuickSAN [72]	yes	no	no	no	block IO	benefits limited to the block layer
NFS [67, 96]	yes	no	yes	no	NFS file IO	apps. must copy data to/from fs bufs.
BGAS [119, 304]	no	N/A	yes	yes	RDMA queues	limited to local flash access
Corfu [37]	yes	no	yes	no	Dist. shared log	uses custom flash access protocol
RDMA efforts [107, 264]	no	N/A	possibly	no	KV, Dist. mem.	do not involve remote storage access
FlashNet	**yes**	**no**	**yes**	**yes**	**RDMA queues**	**uses RDMA to access remote flash**

an end-to-end data flow path from application buffers to storage devices. Furthermore, none of these systems provide other highly desired high-performance IO properties [153, 158, 341] such as asynchronous IO with selective notifications, direct polling of userspace mapped queues for low-latency, etc. More specifically, the FlashNet approach differs from iSCSI in its treatment of storage hierarchies. FlashNet eliminates the traditional two-level of memory and storage hierarchy, and brings data into a single, unified storage tier. This unification enables clients to access remote data directly by using RMA network primitives without having to manage separate storage operations. Further differences lie in the provided abstraction. In comparison to the iSCSI's block-based interface to remote storage, FlashNet provided a byte-addressable remote memory interface to remote flash devices. Cray's BrustBuffer (BB) [211, 271] uses flash and NVM storage in various roles such as for temporary shared storage, for absorbing bursty IO, for fast check-pointing, for file system cache, for swapping, etc. In this context, FlashNet can be used to transfer data from remote flash storage for BrustBuffers. FlashNet itself does not restrict in what capacity a remote flash storage should be used. Rather it provides efficient means to transfer data using RDMA network operations. Mojim [369], a recently proposed system, takes a similar stand as FlashNet on increased software overheads in high-performance IO. It uses RDMA for NVM data replication and can benefit from the unified DRAM/flash buffer approach taken by FlashNet.

Many other distributed systems involving NVM storage, such as RAMCloud [265], FaRM [107], KV stores [183, 241], etc., also leverage RDMA for networking. However, they utilize their own server-client architecture that is very different from a file system interface provided by the FlashNet on the server-side. Distributed file systems, such as DAFS [219, 220], DFS [101], and NFS [67], GlusterFS [6], etc., also use RDMA and present a file system interface. Commercial systems such as Violin Memory, use RDMA to access flash in an appliance setting with the SMB protocol [17]. NVMe over RDMA transports NVMe command sets to a remote server using RDMA protocol and can even directly transfer data to a RDMA device using peer-to-peer PCI transactions [40]. However, these efforts lack generality as they use RDMA with the scope of a file system or the block device without delivering the benefits of RDMA operations to client applications. For example, it is not possible to share in-kernel RDMA transport buffers of NFS with an application to avoid data copies when performing a file IO. DAFS presents a new user-space, file system interface to clients, where as FlashNet presents RDMA queues with all supported RDMA operations. This design enables a unified, single-tier storage hierarchy (e.g., Multics) encompassing storage and memory, where remote clients make no distinction between them [158]. Though, a client-side file system interface on FlashNet warrants a DAFS-like approach and protocol extensions [101].

5.6 Conclusion

In this chapter, we have presented FlashNet, a unified software IO stack that provides direct, high-performance, byte-granular access to remote flash storage. It achieves this performance by adopting the well-known path separation principle from RDMA networks and extending it to include a file system and a flash controller. This extension enables a single-tier storage hierarchy in data centers where the data can be staged transparently in remote DRAM or flash while still being accessible using high-performance RDMA operations.

On the backdrop of the recent surge in interest to leverage RDMA in distributed storage and computing, in the next chapter, we focus our attention on extending the path separation philosophy of RDMA in a distributed environment.

6 RStore: A Direct-Access In-Memory Based Data Store

In this chapter, we present RStore, a DRAM-based data store that extends RDMA's separation philosophy to a distributed environment to deliver high-performance data accesses. It achieves the separation at the API level by having distinct calls for allocation and access. This setup lets applications pre-allocate and pre-fetch expensive RDMA resources in a distributed environment before the data processing phase begins. RStore is built on two design principles. First, *decouple resource allocation from its abstraction binding*. This decoupling allows RStore to manipulate and reuse expensive resources (e.g., RDMA buffer allocation and registration) independently from the provided storage abstraction. Second, *keep the IO path thin and fast*. A fast and thin IO path helps to deliver fast data access to applications with their own synchronization logic. RStore further exploits the availability of multiple NICs by striping data across servers. As a result, RStore delivers high performance that is very close to the RDMA-network limits.

The basic storage abstraction in RStore is a flat 64-bit distributed access *namespace*. Applications interact with a namespace using RStore's API that resembles the familiar memory-mapped IO (`mmap` and friends). Using the API, applications can reserve, allocate, and map storage capacity in any namespace. To illustrate the power of the abstraction, we have developed two different applications on top of RStore's API. Our first application is a distributed in-memory graph processing framework called Carafe, which imports, processes, and stores graph data, metadata, and messages in RStore. In our experiments, Carafe outperforms state-of-the-art systems by margins of $2.6 - 4.2\times$ when calculating PageRank. Our second application is a distributed Key-Value sorter called RSort. It stores data in RStore and leverages high-bandwidth data access to deliver high sorting performance. RSort can sort 256 GB of data in 31.7 sec, which is $8\times$ better than Hadoop TeraSort in a similar setting.

Our contributions include (a) design and implementation of a DRAM-based data store where RDMA philosophy is integrated as a first-class citizen in an end-to-end manner; (b) illustration of the system capability by developing two different types of applications using RStore's API

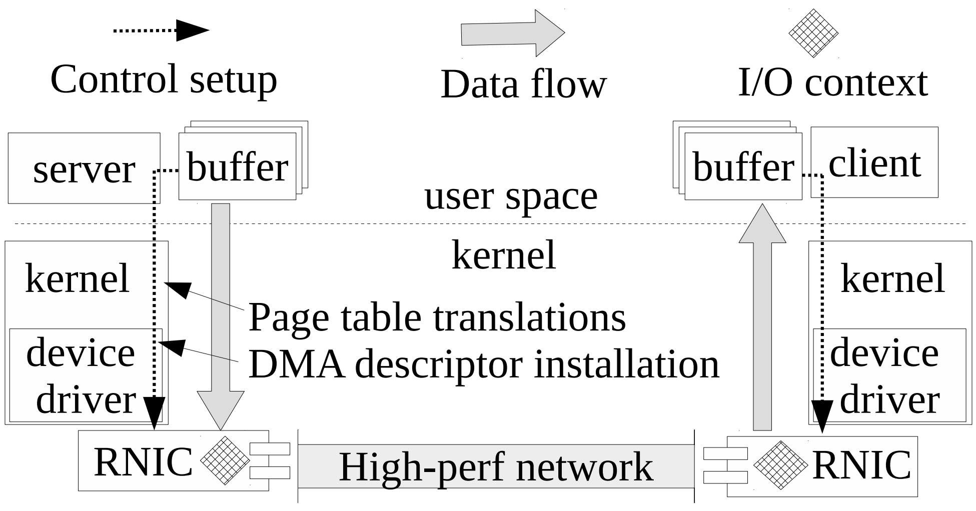


Figure 6.1: RDMA communication model showing the control setup and the data flow.

that allows pre-fetching and pre-allocation of resources out of the performance critical path in a distributed setting; (c) quantification of the distributed control setup cost of RDMA at scale (1000s of connections, TBs of DRAM).

6.1 Opportunities and Challenges with RDMA

The demands on big data analytics platforms to provide real-time performance has led to memory-centric architectures where more and more data is kept in DRAM for fast access. In this context, several in-memory storage systems such as RAMCloud [264], FaRM [107], key-value stores [13, 121], etc., have been proposed. Consequently, due to the elimination of slow disks, the network has become the new performance bottleneck.

To improve the network performance of in-memory storage systems, researchers have advocated using Remote Direct Memory Access (RDMA) technology [107, 180, 183, 241, 264, 326]. As discussed in Chapter 3, RDMA networks like InfiniBand or iWARP provide high-throughput/low-latency with very low CPU overhead by separating the *control path* (or setup) from the *data path* (or access). The fast data path operations offer one-sided network IO semantics and can be offloaded to network controllers. The philosophy of path separation, together with rich IO semantics and hardware offloading, is what gives RDMA its performance benefits.

By deploying RDMA in a limited capacity, one still gets marginal benefits from the offloaded protocol processing and high link rates of specialized interconnects (such as 56Gb/s on InfiniBand). However, the full potential of RDMA is closely tied to its separation philosophy that sets up resources a priori to eliminate overheads from memory management, multiplexing and data copies, etc., during fast data accesses [129]. Figure 6.1 shows the data flow and control flow in a server-client setup.

Extending this philosophy to a distributed environment mandates a careful network and storage resource management, whose cost, without thoughtful consideration, can easily eclipse any potential performance gains from using RDMA-capable networks [130]. Despite previous efforts, leveraging the full performance of RDMA in a distributed environment still remains a challenging task due to multiple reasons.

The first and foremost question is how to extend the separation philosophy of the RDMA networks to a distributed data store. Traditional APIs (such as Key-Value or Files) have narrow interfaces that do not allow resource setup requirements to be captured prior to an access. In the absence of sufficient information, a system cannot leverage RDMA operations to deliver the highest performing IO stack.

Second, in a distributed data store that might offer separation, how can a data-processing application leverage it? What is the right abstraction? The right IO abstraction can help applications, which can identify, create, and pre-fetch necessary distributed objects upfront to gain significant performance gains when performance really matters.

Lastly, distributed RDMA resource management is a complex task. It includes "when and how" to open RDMA connections to servers, share offloaded resources, register buffers on multiple RNICs, etc., in a distributed setting. In comparison, previous RDMA integration efforts have only dealt in part with these challenges. For example, RAMCloud [262] delivers low-latency IO for small objects, but does not let applications manage their resources. On the other hand, FaRM [107] gives the possibility to pre-allocate and manage transactional objects, but does not deliver the lowest possible latency because of object layout adjustments in the IO path.

6.2 RStore

RStore is a distributed in-memory data store that delivers high performance by extending RDMA's separation philosophy from a point-to-point setting to a distributed environment. In this section, we describe its goals, design principles, and implementation details.

6.2.1 Scope and Goals

We consider a rack-scale deployment of RStore, where data is imported from a persistent storage, processed by thousands of cores while the working set is held in distributed DRAMs, and finally, results are written back to the storage. The high-level goals of the system are:

- *Efficient distributed setup path:* Even though the RDMA setup cost is a part of the separated

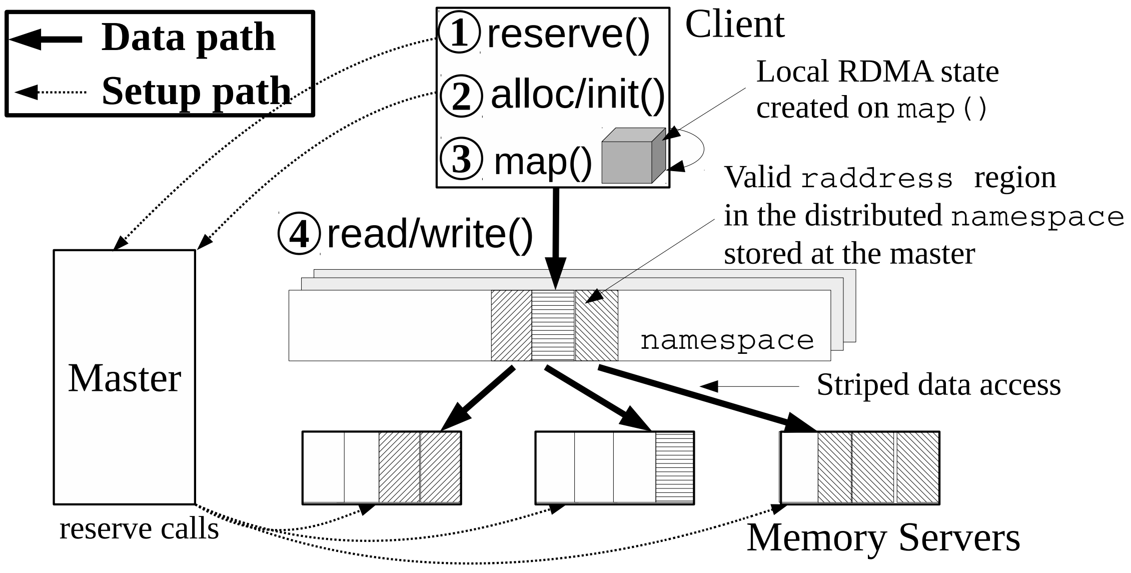


Figure 6.2: Design and components in RStore.

control path, it should not be prohibitively high. This cost is one of the primary concerns with RDMA deployments in distributed settings.

- *Light-weight abstraction:* The proposed abstraction should be intuitive, light-weight, and general-purpose. This property is key to building applications with different consistency and performance requirements.

- *High performance:* The system should be able to deliver high bandwidth and low latency to a single client by leveraging all networking hardware.

However, in our pursuit of achieving these goals, we sacrifice a few system properties from an implementation point of view. RStore's fault-tolerance handling is primitive. It does not strive for data durability in the case of arbitrary memory server failures. Memory server replication or fast recovery [262] can be used to deal with this issue, but we have not yet done so. We have implemented a copy-on-write mode (see Section 6.2.7) to provide a strong atomicity guarantee under concurrency and failures. This mode, however, is not evaluated in the thesis. We first start here by discussing the abstraction and design principles, and how they help us to achieve RStore's goals.

6.2.2 Components

RStore has three main components: a master, a set of memory servers, and a client-side user library. The master is a logically centralized entity that acts as the system arbiter and stores all metadata in DRAM for fast servicing. The metadata consists of created namespaces, allocated memory regions and their locations, permissions, active client states, etc. Data is distributed and stored in DRAMs of participating memory servers. Their primary responsibility is to allocate and prepare DRAM buffers to be accessed by RStore applications. Applications interact with RStore using a client-side API. Applications and memory servers communicate with the master using a fast and scalable Remote Procedure Call (RPC) implementation using RDMA two-sided send/recv operations [327]. Figure 6.2 captures the interaction among the components.

6.2.3 Abstraction and Principles

The basic access abstraction in RStore is a flat, 64-bit, byte-addressable storage address space called *namespace*. RStore's applications create, join, and allocate storage capacity in multiple namespaces. An allocated memory segment is identified by a globally visible <address, length> tuple, called `raddress`. Applications use `raddress` segments to store and share distributed data structures, tables, data-blobs, etc. RStore is built on the following two design principles:

1. *Decouple resource allocation from its abstraction binding:* The distributed control path in RStore consists of setting up two types of resources: (a) systems IO resources; (b) metadata associated with RStore. The first involves allocating memory, registering buffers with RNICs, opening up RDMA connections, etc. The latter involves binding these resources to application-visible RStore namespaces and addresses. We made RStore's distributed setup path efficient by decoupling these two resources. Applications can manage costly system resource allocation separately from the performance critical path, which may encompass relatively cheaper metadata manipulations in RStore. For example, by reserving TBs of DRAM memory independently from binding them to an `raddress`, applications can quickly allocate and free `raddress` segments without having to allocate and pin memory buffers repeatedly.

2. *Keep the IO path thin and fast:* RStore does not interpret or impose any structure on the stored data. Consequently, it avoids the overhead of the implicit synchronization associated with data structure accesses [23]. Distributed applications with explicit coordination logic among workers do not require additional synchronization overheads from the storage. While maintaining the same considerations as RDMA, RStore translates IO requests to RDMA operations and avoids context switches, memory allocations, data touch operations (e.g., copies or layout adjustments), etc., during a data access. It delivers large aggregate bandwidth by striping data across multiple RNICs without overwhelming a single server or link.

Table 6.1: RStore client API. The `reserve()`, `alloc()`, and `map()` calls constitute the control path in the distributed setting. The `read()` and `write()` calls are fast data-path calls, in which no global or local resource allocations happen.

Object	RStore API calls	Description
Master	`connect(struct sockaddr*)`	Connects to the RStore master
	`create_ns(string name)`	Creates a namespace with a string identifier, returns Namespace *obj
	`join_ns(string name)`	Joins an already created namespace, returns Namespace *obj
	`delete_ns(string name)`	Destroys a passed namespace
Namespace	`reserve(u64 size)`	Allocates and prepares DRAM on memory servers for RDMA access
	`release(u64 size)`	Releases memory reserve from a namespace
	`alloc(u64 size, int flags)`	Allocates a globally visible address segment, returns `raddress` *obj
	`free(raddress*)`	Deletes an `raddress` allocation segment at the master
RAddress	`init(namespace*, u64 addr, u64 size)`	Initializes an `raddress` object to a valid global address and length
	`map(int flags)`	Allocates local network/memory resources, returns a local void* ptr
	`unmap()`	Frees all resources associated with this `raddress` mapping
	`read(void *p, u64 size)`	Reads a mapped memory region for a passed length
	`write(void *p, u64 size)`	Writes a mapped memory region for a passed length

6.2.4 RStore API and the Path Separation

RStore achieves the separation using discrete API calls to set up and access resources in a distributed environment. There are three explicit calls, namely, `reserve()`, `alloc()`, and `map()` to manage resources within a namespace. Following the decoupling principle, `reserve()` allocates DRAM buffers for RDMA accesses at the memory servers, `alloc()` binds them to an `raddress` at the master, and `map()` associates the `raddress` to local memory at the clients. An allocated `raddress` region is physically served by DRAMs from multiple memory servers. Multiple applications can concurrently map and access the same `raddress` region. These three calls collectively constitute the distributed control setup (or resource setup) in RStore. After the control setup, fast data-path calls, i.e., `read()` and `write()` in mapped regions, do not involve any resource allocations. Table 6.1 gives an overview of RStore's API and lists its main objects and associated calls.

6.2.5 Distributed Memory Management

RStore manages distributed DRAMs internally in a granularity of *chunks*. Applications, however, can allocate and map `raddress` regions of any size. A list of free chunks, allocated chunks to `raddress` regions, their access permissions, mapping types, reference counts, and active client states, etc., are maintained as system metadata at the master.

Storing and accessing data in RStore are multi-step processes. First, an application must reserve sufficient DRAM capacity by calling `reserve()`. If the free chunk list at the master has sufficient capacity, the RPC returns immediately; otherwise the master chooses a set of memory servers to reserve the memory capacity requested. The current implementation uses a primitive round-robin policy to distribute the load uniformly. Upon receiving a `reserve()` RPC call from the master, the memory servers allocate and register chunks of DRAM to an RDMA device and communicate RDMA credentials (STag, virtual address, and length) back to the master. These chunks are added to the free chunk list.

Second, an `alloc()` call stitches together reserved memory chunks from different memory servers (for best parallelism) under a single distributed `raddress` region. This globally visible binding between an `raddress` region and the memory chunks is created and stored at the master. The newly created `raddress` region together with the location of its chunks is returned to the client as the result of the `alloc()` RPC. For a previously allocated region, a client can initialize an `raddress` object by calling `init()` with a valid <address, length> range. The call fetches the chunk locations from the master. The master maintains appropriate reference counting on objects to avoid freeing them while they are still mapped and in use at clients. When a client's connection aborts, the master cleans up the associate states and references.

These metadata manipulations are atomic as they are serialized by the master using appropriate locks.

Lastly, a valid `raddress` object requires a local mapping before the IO operations. Similarly to the POSIX `mmap()` call, a `map()` call returns a local DRAM address, which is made RDMA-ready by the RStore's client library. A client uses this address for staging and modifying data in RStore. When a valid `raddress` object is destroyed, it notifies the master for reference cleanup.

6.2.6 IO Operations and Synchronization

The fast data path in RStore consists of `read()` and `write()` calls. These calls do not involve any data touch operations or resource allocations anywhere in the system, which is the key to deliver high performance to applications. A `read/write()` call is divided at the chunk boundary and individual chunk IO requests are then translated into one-sided RDMA operations for zero-copy network transfers. The byte-granular nature of IO operations fits perfectly with the message-oriented nature of RDMA operations: an RNIC understands message boundaries and only notifies an application when the complete message has been sent or received. Hence, the transfer time depends on the IO size, rather than on the mapped memory size.

With its key focus on separation and performance, RStore does not provide any form of global IO synchronization. RStore's clients must coordinate among themselves to define the concurrency access model. We argue that this is not an unusual feature as many distributed applications tend to have their own synchronization mechanism using external services, e.g., Zookeeper or DARE SMR [278]. We illustrate in Section 6.5.1 how one can achieve global barrier coordination and develop applications using RStore.

6.2.7 Copy-on-write for Machine Failures

The zero-copy architecture of RStore modifies data in-place. Hence, a failure of a writer client leaves data in an inconsistent state. To provide atomic updates in case of a memory server failure, we are implementing a copy-on-write (COW) type `raddress` segment (indicated by a flag in the `alloc()` call). In the COW-mode, a small amount of per-chunk metadata (8 bytes) is maintained at the memory servers to communicate the chunk state to clients. This metadata is accessed during IO operations using RNIC's scatter-gather capability.

While mapping a COW region, a writer client synchronizes with the master to get a time-bounded (default: 10 sec) lease for new memory locations in a multiple of chunks for every write operation. The master enforces a policy of one writer with concurrent readers by rejecting new leases for an already mapped COW region to another writer. After local modifications, the data is written out in multiples of chunks to the new chunk locations. Upon a successful `write()`, the writer notifies the master to update the `raddress` metadata to point to the newly written chunk locations. If any of these steps fail, the master retains the last known chunk locations for the `raddress` region. When a lease expires (owing to a client's failure or inactivity), the master garbage collects the lease's chunk locations. After a successful write notification from the client, the master marks the per-chunk metadata at the old locations as tainted.

A concurrent reader in a COW-mapped region can be in one of three states. First, it reads clean data while a write is in progress. As the new data is written out of place, this read returns the last known consistent value for the data. Second, the reader sees a subset of chunk locations as tainted while the master is updating the per-chunk metadata. Third and last, the reader sees all per-chunk metadata as tainted. Note that in the last two cases, the presence of tainted metadata only notifies the reader about the availability of new data, but the old data locations are not over-written or invalidated as the new data is written out of place in new locations. If it chooses so, a reader can still work with a consistent copy of the data stored at the old locations. Alternatively, the reader, upon detecting the tainted metadata, can proceed to obtain the new data chunk locations from the master. When an old chunk location has no active client mappings, the master uses it for new allocation requests.

6.2.8 Discussion on RStore API and Abstraction

RStore's unique memory-like API has some critical advantages that differentiates it from state-of-the-art systems:

- **Explicit Distributed Resource Management:** RStore's API gives applications control over RDMA and memory setup. Using this explicit control, applications can allocate, pre-fetch, and prepare stateful objects associated with TBs of DRAM by calling `init` (or `alloc` for a fresh allocation). These calls fetch chunk locations from the master and can be called separately from the `map` call, which involves local memory commitment. Furthermore, applications can use this hierarchical setup of control calls (`reserve, alloc, map`) to progressively distribute the resource allocation cost in a most efficient way as suited to their workload requirements.

- **Unified Network and Application Buffers:** By using an explicit `map` call, RStore integrates network and application buffers. As explained, the `map` call returns a local `void*` pointer to a memory buffer, which is made RDMA-ready by the client-side library. This buffer is known to the network for IO and to the application for data access, thus eliminating a data copy which is typically done to move data between network and application buffers. Hence, RStore's IO stack is a true zero-copy stack on both, TX and RX sides. Furthermore, due to the data copy elimination, one-sided RDMA operations in RStore deliver data directly into the application buffers with highest bandwidth and lowest latency.

- **Expressive Memory API:** The raw byte-addressable memory abstraction of RStore is the most expressive general-purpose storage abstraction. This memory abstraction enables the building of distributed linked data structures with pointer arithmetic for offset calculations. The `map` call, which supports mapping partial address ranges, allows large memory objects to be partially mapped and updated. Both of our applications use this facility to build and access distributed data structures. This would not have been possible on object or Key-Value stores that only permits full object updates with their `get/set` API calls. For example, it is not possible to update a pointer at a particular offset in a value of a key in a Key-Value store.

6.3 Implementation of RStore

RStore is implemented in Linux (for `3.13.11` kernel) in about 15K LOC[1] of C++ which contains code for the master (4K), memory servers (1K), the application-side library (3.5K), and common subsystems including RDMA (6.5K) These components are logically separated and hence can run on a single physical box as well. The implementation follows the best practices of RDMA resource caching and sharing as recommended in the literature [107, 327].

6.3.1 System Booting

How does the first client in RStore discover where the data has been allocated to and stored? To facilitate this discovery process, RStore fixes the first allocation address in any namespace to a pre-defined, globally known address called `RSTORE_INIT_RADDRESS`. This address can also be queried from the master. This discovery mechanism is similar to reading the partition table of a storage device at a fixed block address.

We now explain how this fixation also facilitates the initial synchronization between multiple clients in a distributed setting. When multiple clients boot simultaneously and want to coordinate, they all try to `map()` the first allocation address with an agreed-upon, pre-defined finite-length metadata. If the `raddress` region was not allocated before, a client fails to map the region with an "Invalid Address" error. Upon receiving this error, the client proceeds to `alloc()` the region. As the `alloc()` call is atomic, only one client succeeds and others fail,

[1]Generated using David A. Wheeler's 'SLOCCount'.

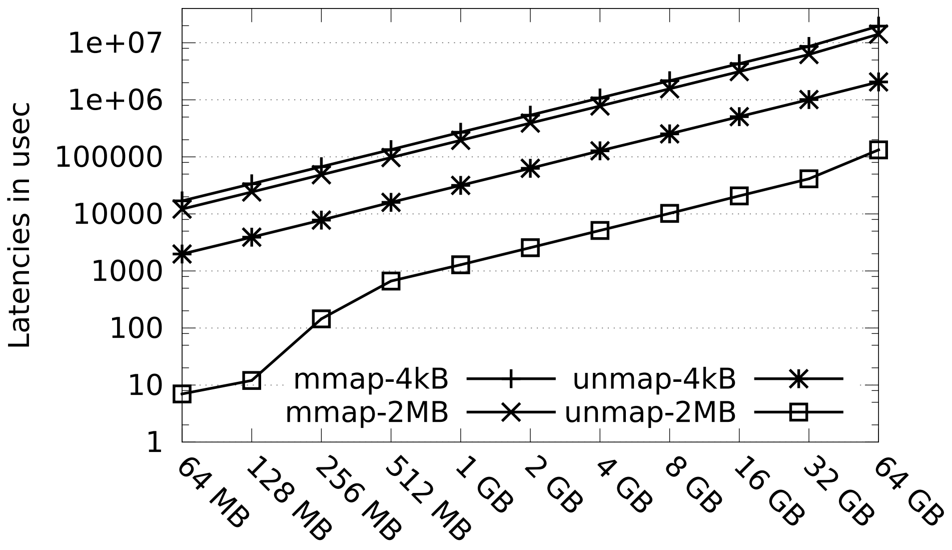


Figure 6.3: `mmap` and `unmap` costs in Linux with 4 kB and 2 MB page sizes.

with an "Address in Use" error. Upon receiving this error, clients re-attempt to map the address region and succeed. At the end of this bootstrap synchronization, each client has a metadata region mapped to their local memory which they can read or write to get updates from other clients. This facility is used to develop our Global Barrier Coordination service described in Section 6.2.7.

6.3.2 RDMA Resource Caching

Although memory management and RDMA objects management are parts of the control path, they have prohibitively high setup cost for a low-latency environment. An RDMA connection takes 2-3 msec to connect and allocate associated connection resources. Independently from RDMA operational costs, it takes more than one second to allocate 4 GB of DRAM (page population included) in Linux (see Figure 6.3). Allocating memory from huge page pool performs a bit better due to the dedicated nature of page management done by the Linux kernel where huge pages are reserved beforehand and are only allocated on a special request indicated by the `MAP_HUGETLB` flag.

RStore caches and reuses RDMA resources (memory buffers and connections) to hide the high cost of control setup for repeated accesses. RDMA-ready memory (which includes populating virtual memory regions and registering the pages with the RNIC) is managed using a simple user space buddy allocator. Calling `unmap` on an mapped memory region in RStore does not immediately release resources until a configured threshold is reached. As RNICs have a limited number of offloaded on-NIC resources (e.g., number of memory regions, connections,

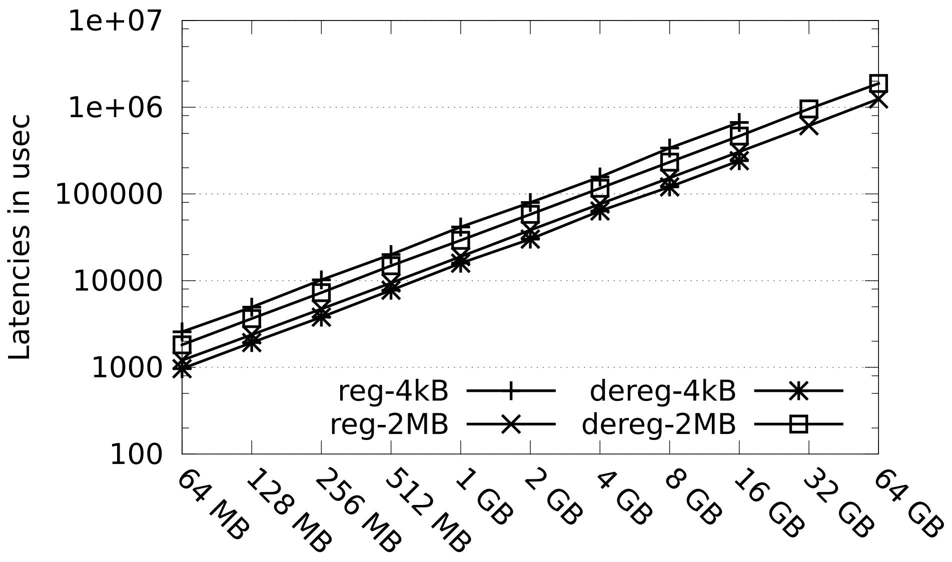


Figure 6.4: Memory registration and de-registration costs with 4 kB and 2 MB page sizes. The 4 kB page size does not allow registration beyond 16 GB buffer size.

QPs, and completion queues), RStore optimizes the number of new unique resources created. RDMA connections to a memory server are shared across multiple mappings. All connections on an RNIC device share the same completion queue for notifications. We quantify the effectiveness of resource caching in Section 6.4.2.

6.3.3 Hugepages and Memory Registration

The amount of metadata generated during a memory registration call is proportional to the number of pages. The size of this metadata determines how much memory can be registered as this metadata is stored on the on-chip memory of the RNIC. For 4 kB page backed memory buffers, we can only register up to 16GB of DRAM with T4 RNICs.

To reduce per-page entry metadata overhead and increase memory registration limits, RStore uses hugepage (2 MB on `x86_64`) backed memory. Hugepages allow RNICs to coalesce the contiguous physical pages buffer lists into a single DMA mapping. Figure 6.4 shows the cost of memory registration for 2 MB hugepages. The memory registration costs are consistently better for hugepages than 4 kB pages. But 4 kB page-backed buffers are faster to de-register. We do not know the exact cause of this behavior from the device.

CPU	Dual Xeon E5-2690, 2.9 GHz 16 cores
DRAM	256 GB, DDR3 1600MHz
NIC	3 Dual port Chelsio T4 iWARP RNICs
Network BW	60 Gb/s (3 $\times$ 2 $\times$ 10 Gb/s)
Network Latency	9.6 μsec, 8B RDMA read latency

Table 6.2: 12-machine testbed configuration for RStore evaluation. All machines are connected via two IBM BNT G8264 switches.

6.4 Performance Evaluation

The performance evaluation is done on our 12-machine iWARP/Ethernet testbed as shown in Table 6.2. For all benchmarks, clients and memory servers (72 servers in total, 6 per machine) are co-located on every machine. The master runs on a separate, dedicated machine. The chunk size is set to 64 MB. The key performance evaluation highlights are:

- **Efficient distributed control setup path:** RStore reserves and allocates 1.15 TB of DRAM in 22.1 sec (52.1 GB/sec). Clients need 19.9 sec to map this memory in their local DRAM.
- **High performance by maintaining the separation philosophy in a distributed setting:** RStore delivers close to bare-metal latency (12 μsec, in comparison the iWARP network latency is 9.6 μsec) and high aggregate cluster bandwidth (705 Gb/s for 12 servers, 58.7 Gb/s per server).
- **High application performance using RStore's API:** Carafe, our distributed in-memory graph processing framework, is 2.6-4.2$\times$ faster than state-of-the-art systems in calculating PageRank on a graph containing millions of vertices. RStore sort (RSort) sorts 256 GB of data in 31.7 sec, which is 8$\times$ better than Hadoop TeraSort in a similar setting.

6.4.1 Cost of the Distributed Control Path

In this section, we quantify the cost of the distributed control setup path, which consists of `reserve()`, `alloc()`, and `map()`. Figure 6.5 shows the performance for a single client. The y-axis shows the latency of the operations in comparison to the buffer size (on the x-axis). The `reserve()` and `map()` calls are the most expensive calls as they involve costly memory allocation operations (see Linux `mmap` cost in Figure 6.5). The `reserve()` call benefits from spreading the memory allocation across multiple memory servers.

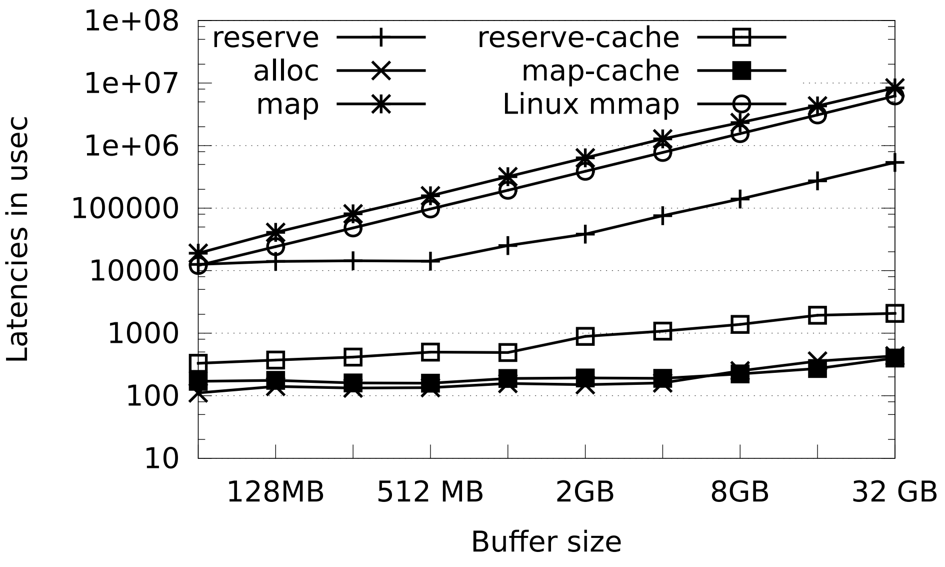


Figure 6.5: Control setup cost for a single client.

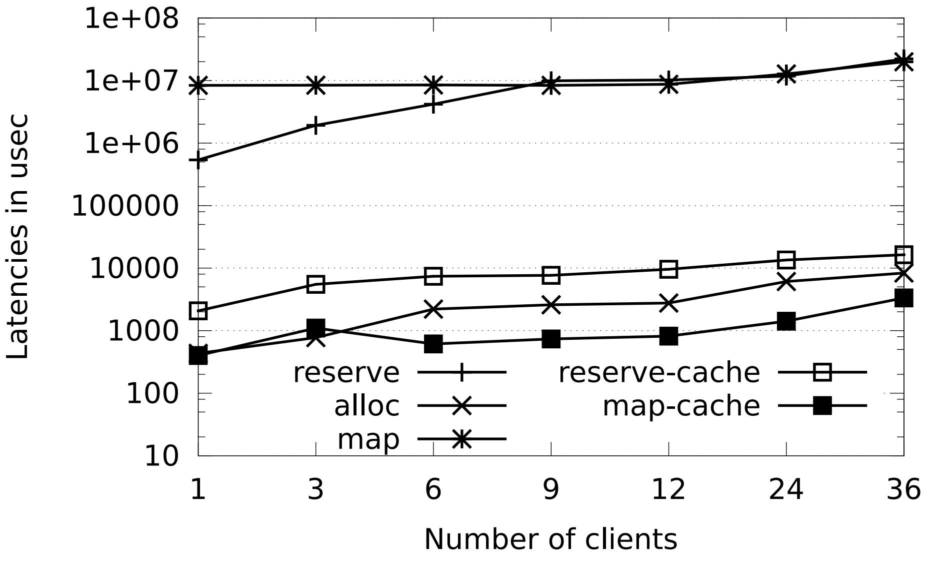


Figure 6.6: Scaling of the control path setup cost.

Figure 6.6 shows the scaling of the control cost in comparison to the number of clients. A variable number of clients (shown on the x-axis) concurrently starts to reserve, allocate, and map 32 GB memory regions in the client's own namespace. As shown in Figure 6.6, the setup cost scales gracefully, and 36 clients can prepare 1.15 TB of DRAM in approximately 22.1 sec. Mapping this memory takes 19.9 sec (18.7 sec are from concurrent `mmaps`), and involves opening, in total, 2,592 (72 servers $\times$ 36 clients) RDMA connections between machines.

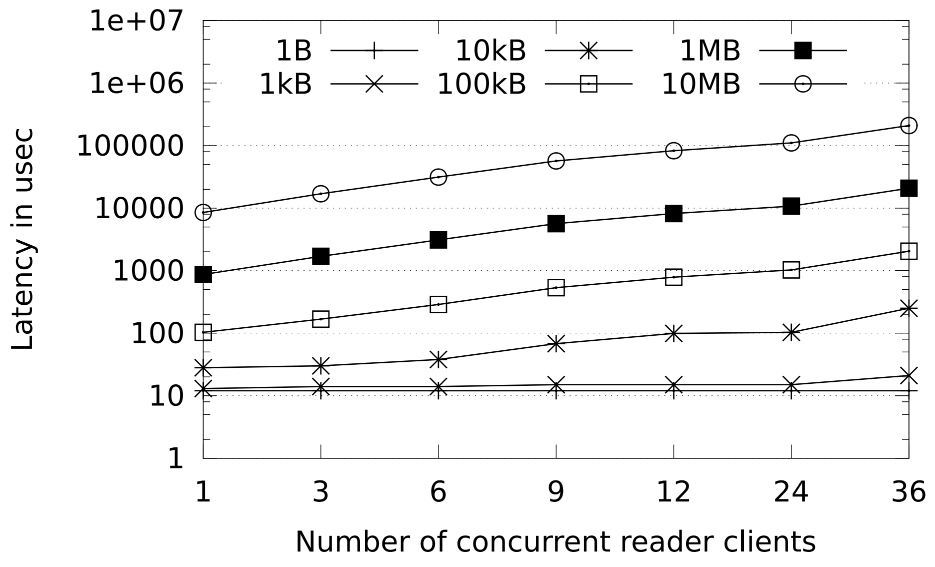


Figure 6.7: Read latencies wrt. concurrent readers.

6.4.2 Efficiency of the Resource Caching

Memory allocation is the primary contributor to the overall control setup cost. It is heavily influenced by the number of co-located clients and memory servers, the size of the allocation request, the number of free hugepages, memory fragmentation, etc. However, a part of the memory allocation cost is only incurred upon a cold start. For example, after the first `reserve()` call to a memory server, which involves allocating new memory segments via `mmap`, the RDMA credentials are used repeatedly between subsequent `release()` and `reserve()` (also between `free()` and `alloc()` calls). The master caches and maintains a configurable number of free chunks. The memory caching is most effective once a workload has hit its peak working set size and reserved sufficient DRAM. Similarly, the caching also helps on the client side with repeated `map()` and `unmap()` calls. Figure 6.5 and Figure 6.6 also show the cached performance with `reserve-cache` and `map-cache` lines. In the cached mode, RStore can reserve and map 1.15 TB of DRAM in 16.3 msec and 3.3 msec respectively — a three orders of magnitude improvement over the cold-start performance.

RDMA connection caching at the client-side is very effective in hiding the high RDMA connection setup cost (about 2.5 msec/connection). However, as the number of servers increases, so does the cost of maintaining the connection cache (with tied-up associated offloaded network resources). A typical RNIC can support up to 64K offloaded connections. In our setup, we have not yet reached these limits.

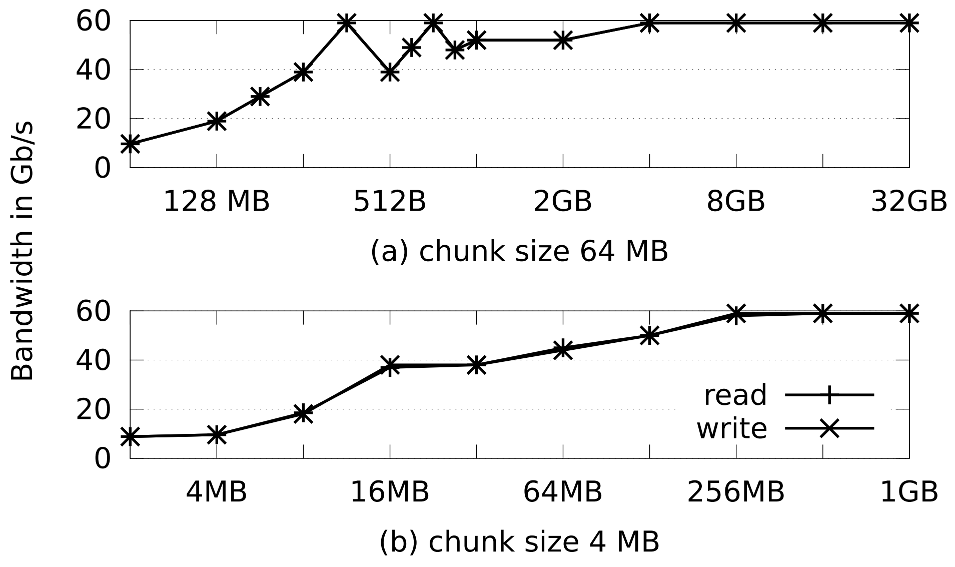


Figure 6.8: Single-client IO bandwidth for `read()` and `write()` operations.

6.4.3 Performance of Data-Path Operations

Latency: As RStore's data calls map directly to RDMA operations, their performances are very close to the network limits. On our iWARP cluster (with network round trip latency of 9.6 μsec), it takes 12 μsec to read 8 bytes of remote data (not shown). We expect this performance to improve further with ultra-low latency interconnects such as InfiniBand. Figure 6.7 shows the effect on the read latency (in μsecs on the y-axis) as we increase the number of concurrent reader clients (on the x-axis). For small requests (less than a kB), RStore is capable of delivering constant IO latencies. As the size increases, the request becomes bandwidth bound and the latency increases.

Bandwidth: The chunk size determines the level of network parallelism that clients can achieve in RStore. For large buffer sizes, a client gets the peak bandwidth of 59.1 Gb/s, less than 2% below the theoretical maximum of 60 Gb/s. Figure 6.8(a) shows the performance of a single client with respect to the various buffer sizes. The sawtooth shape of the bandwidth curve can be explained as follows: between 64 MB and 384 MB (which is 6×64 MB), the bandwidth increases linearly in multiples of a single NIC capacity (10 Gb/s) and hits the peak at 384 MB. Then the bandwidth falls because larger buffer sizes create an uneven distribution of chunks on our 6 NICs/machine setup. Hence, whenever the total transfer size is a 6-multiple of 64 MB, the client gets the full bandwidth. To confirm this hypothesis, we have also repeated the experiment with a 4 MB chunk size (see Figure 6.8(b)). We observe a similar pattern with the client getting the full bandwidth above a 256 MB buffer size. For sufficiently large sizes, the transfer time dictates the overall completion time, and this unevenness no longer matters.

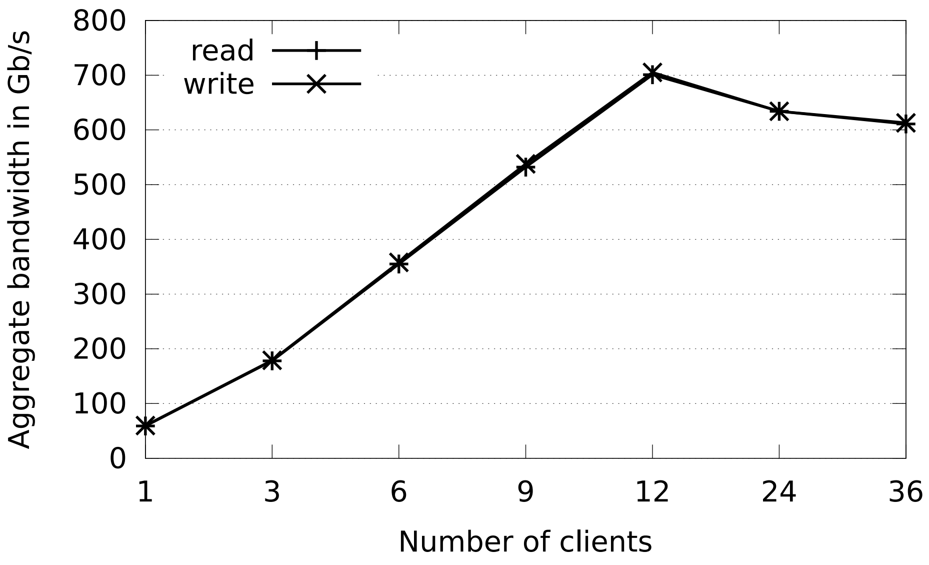


Figure 6.9: Aggregate cluster IO bandwidth of RStore.

Figure 6.9 shows the aggregate system bandwidth as the number of clients increases. The performance scales linearly and peaks at 705 Gb/s (theoretical maximum: 720 Gb/s) until we host multiple clients per machine. We are investigating the QoS management of RDMA traffic. Our initial analysis suggests that one reason for the drop is the co-location of memory servers and clients on the same physical machine in our limited 12-machine testbed.

6.5 Applications

We have developed two different applications, a distributed graph processing framework called Carafe, and a distributed Key-Value sorter called RSort. As both of these applications require coordination among workers, we have also built a simple coordination mechanism by leveraging RStore's byte-addressability property to atomically access one-byte flags in a distributed metadata structure.

6.5.1 Global Barrier Synchronization (GBS)

The coordination mechanism consists of a central coordinator and multiple workers. These entities communicate through a distributed shared data structure, laid out in RStore by the coordinator. This data region is mapped in the memory of every worker. The region contains a one byte status field together with application specific data. Byte accesses in RStore are atomic as one byte is the smallest unit of data access. The layout of the data region is shown in Figure 6.10. It contains a coordinator-specific global field and a per-worker field. The per-

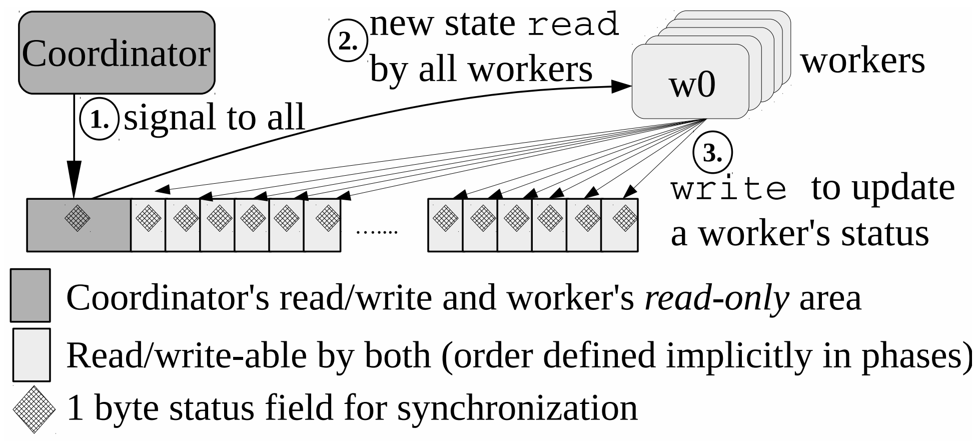

Coordinator
2. new state read by all workers
w0
workers
1. signal to all
3. write to update a worker's status
....
Coordinator's read/write and worker's *read-only* area
Read/write-able by both (order defined implicitly in phases)
1 byte status field for synchronization

Figure 6.10: Distributed barrier coordination.

worker field is written by both the coordinator and a client. At the start, the coordinator writes application-specific data in the per-worker area (depends upon the application logic) and marks their status fields as valid. It then proceeds to signal all workers by writing its status byte field as “start”. Upon reading “start” in the coordinator field, workers then read their area to get an assigned workload, execute it, and mark their status byte as “finished” or “error”. The coordinator, upon reading that every worker has finished its assigned work, can decide to start the next phase of work. Our simple coordination mechanism avoids write-write synchronization by taking turns to read or write byte status fields atomically. Carafe uses this mechanism for superstep synchronization and message delivery among workers. RSort uses it for work assignment and moving from phase one (classification) to phase two (sorting).

6.5.2 Carafe: Distributed Graph Processing

Carafe is a distributed in-memory graph processing framework, which is developed by Clemens Lutz for his M.Sc. thesis work. Here we give necessary details about how it stores, accesses, and manipulates the graph structure inside RStore. For more details about Carafe, please refer to the thesis [215]. The key performance requirement for Carafe is low-latency access to the graph structure, metadata, and messages. We have implemented an online [312] and a Pregel-like [222] graph processing engine.

Graph Storage Format: Carafe imports edges and vertices in separate RStore namespaces. The edges are stored in an adjacency list format. Edges are directional, and bi-directional edges are split and stored twice. Vertices are stored as a single contiguous array. A single vertex structure contains its internal id (array index), externally associated id given by the graph file, offset of its adjacency list in the edge namespace, and size of the list, etc. It also contains a graph property map, which contains associated contextual information and run-time data from

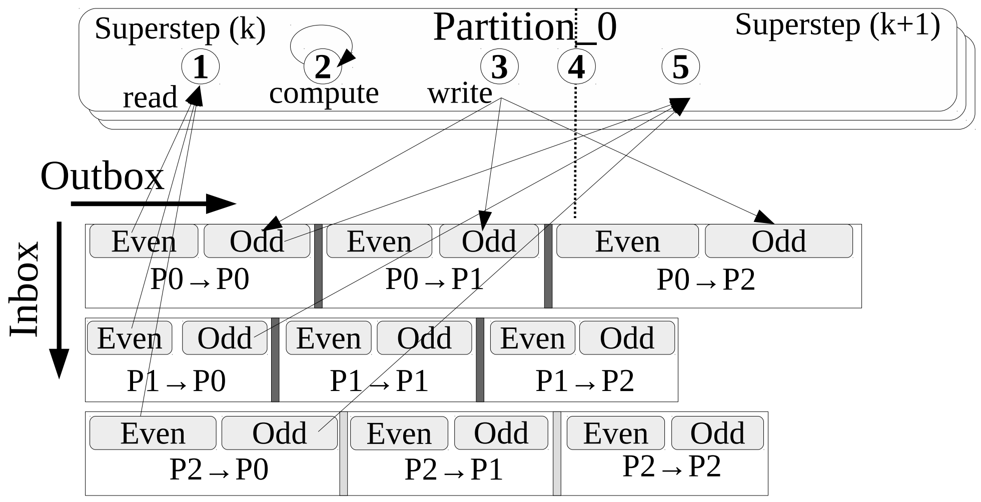


Figure 6.11: Message delivery and Pregel execution in Carafe. Assuming k is even (1) collect incoming messages from the even containers; (2) invoke `compute` on vertices with messages; (3) write messages to odd containers; (4) synchronize the superstep; (5) collect messages from even containers for (k+1)th superstep.

algorithms. The graph is represented by a `CarafeGraph` class that contains high-level pertinent information about the graph and the associated RStore metadata about vertex and edge namespaces. At this moment, Carafe does not support graph mutability.

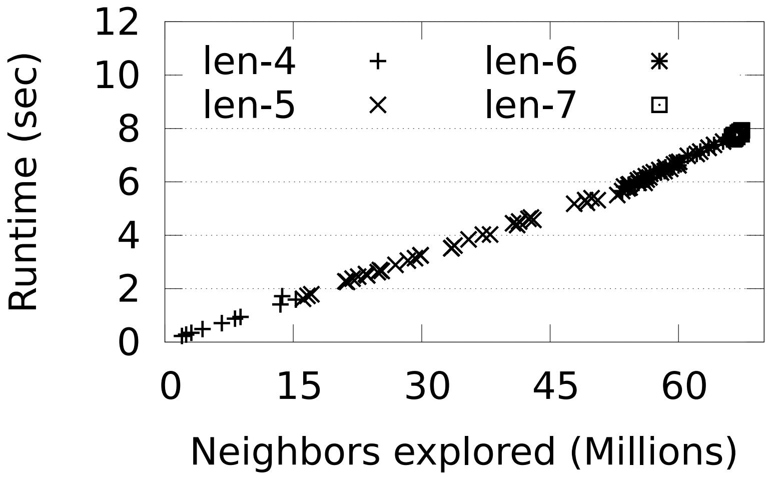
Online Graph Exploration: Carafe provides a `VertexHandle` and an `EdgeIterator` for iterative, online graph scans. A `VertexHandle` is initiated by passing a vertex id to the `CarafeGraph`. This action involves calculating the right offset into the vertex namespace, mapping and reading that vertex from RStore, and then initializing an `EdgeIterator` to its edge list in the edge namespace. An `EdgeIterator` provides a mechanism to iterate through neighbouring vertex ids. Applications can use a neighbouring vertex id to initialize a new `VertexHandle`. Once done, the application must release associated RStore resources by calling `unmap` on these graph objects. We have implemented the Dijkstra shortest path algorithm using this facility.

Graph Partitioning and Pregel Model: Carafe does weighted vertex partitioning to divide vertices into equal weight partitions. Our initial attempt to partition the graph just using edges resulted in uneven computation. With this partitioning scheme, some partitions ended up containing too many sparsely connected vertices whose access time dominated the computation. In the weighted vertex partitioning, Carafe assigns weight to a vertex (represents access overhead) as well as an edge (represents access and messaging overheads). The graph is then divided into equal weight partitions containing vertices. These partitions are then assigned to worker machines. Worker machines access vertices using a specialized `SequentialEdgeIterator` that loads and reads graph data on a partition size granularity. Each partition contains a partition worker thread to execute the Pregel logic that invokes an equivalent `compute` function on vertices with messages from neighbours. After each superstep, new vertex data is written back to RStore and every partition worker coordinates with the Pregel master for the next superstep. We have implemented PageRank using the Pregel model.

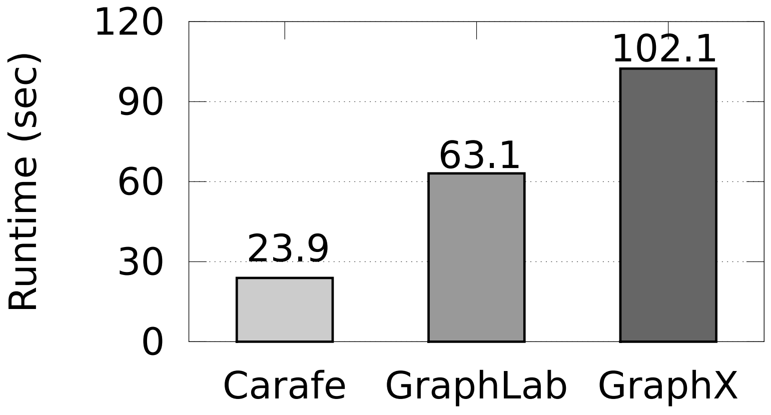
Message Passing: Message passing represents a major overhead in the Pregel computation model. RStore's efficient network IO enabled us to implement a shared mailbox schema to send and receive messages in Carafe. In the shared mailbox model, each partition worker reads and writes messages into RStore namespaces. Messages in Carafe are delivered individually. The Carafe mailbox is divided into as equal a number of rows and columns as there are workers. These `segments` (i^{th} row and j^{th} column) contain messages sent from individual partitions (from Partition$_i$ to Partition$_j$). Individual segments are further divided into two areas called `even` and `odd` containers. The size of each container is determined by the number of edges between partitions. For the i^{th} partition, i^{th} row and i^{th} column represent its Outbox and Inbox, respectively. During the k^{th} (assuming k is even) superstep, workers read from `even` Inbox containers and write new messages to `odd` Outbox containers. The `even` and `odd` containers switch their roles in the next supersteps. This setup achieves read-write coordination at the expense of more storage. Figure 6.11 shows an example of the mailbox setup for 3 partition workers.

Evaluation: We evaluate performance of Carafe on the LiveJournal social network graph from the Stanford Network Analysis Project [8]. The graph contains 4.8 million vertices and 68.9 million edges. The graph is distributed and stored completely in DRAM of servers. Our first benchmark calculates the shortest path between 100 randomly chosen vertex pairs using the Dijkstra shortest path algorithm. Figure 6.12a shows the runtime of the algorithm in comparison to the number of neighbours explored and the discovered path lengths. As shown, the cost of neighbourhood exploration increases linearly with the number of vertices accessed. For example, Carafe explored 67 million edges in 7.9 seconds for a path length of 7.

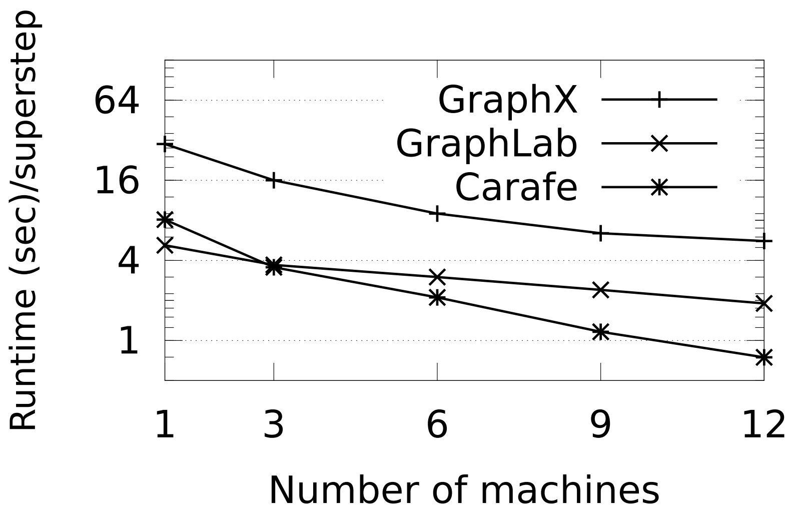
Our second benchmark consists of executing the PageRank algorithm on the LiveJournal graph in Carafe's Pregel engine. We compare the performance of Carafe against GraphLab [144] and GraphX [145], which are two of the fastest distributed graph processing systems in the literature, running in a similar setting. These systems use traditional TCP/socket based infrastructure for network IO operations. Figure 6.12b shows the runtime of PageRank for the first 32 supersteps in all three systems. Carafe outperforms GraphLab and GraphX by a margin of $2.6\times$ and $4.2\times$, respectively. Carafe also scales linearly on our 12-machine cluster as shown in Figure 6.12c.



(a) Dijkstra shortest path.



(b) PageRank runtime for 32 supersteps.



(c) PageRank scaling.

Figure 6.12: Performance of Carafe on the LiveJournal social network graph [8]

.

6.5.3 RSort: Distributed Sorting on RStore

We have implemented a distributed key-value (KV) record sorter called RSort. We use `http://sortbenchmark.org/` scripts to generate input KV records in the Indy mode. In the Indy mode, the key values are uniformly distributed over the whole key space.

RSort solves two key challenges while building on top of RStore. First, how do multiple clients coordinate among themselves? RSort leverages RStore's byte-addressability property to access one-byte flags in a distributed metadata structure. Byte accesses in RStore are atomic as one byte is the smallest unit of data access. Second, how do clients discover and access intermediate data store in discrete `raddress` segments? RSort solves this problem by using multiple namespaces with meaningful names and laying out distributed link lists of `raddress` segments. We explain these concepts in more detail in the following paragraphs. As RSort does not perform any locality aware optimizations, its performance is governed by the data access bandwidth. RSort implements a two phase bucket sorter. In the first phase, input data is classified into range buckets, and in the second phase, individual buckets are sorted locally.

Phase One (Classification): In phase one, all workers first read and then classify the input data into multiple *bucket* namespaces. A bucket, private to a worker, represents a specific key range for which its namespace should contain records. Having private bucket namespaces eliminates the need for write-write synchronization when multiple workers try to append a record in a same key range bucket namespace. The names of bucket namespaces follow a syntax of $worker\_i\_bucket\_j$, which uniquely identifies the j^{th} bucket namespace for the i^{th} worker. The key range for the j^{th} bucket for every worker is defined globally, and each worker has the same number of buckets. Consequently, the total number of buckets in RStore is $buckets\_per\_worker \times total\_workers$. At the end of phase one, each worker writes its metadata field to mark *finish*. Upon detecting that all workers have finished with the classification, the coordinator starts phase two by writing its status flag in the GBS service.

Phase Two (Assembly and Sorting): In phase two, a key range is exclusively assigned to a particular worker by the coordinator. Recall that a bucket with name $worker\_i\_bucket\_j$ contains all records that a worker i has seen for the key range assigned to bucket j. Upon getting the assigned key range, which might belong to bucket index j, a worker joins all bucket namespaces of the name syntax

$$worker\_i\_bucket\_j, where\ i \in [0, total\_workers - 1].$$

The worker then reads and assembles all records in a final output namespace. The final namespace has name syntax $final\_j$, where j represents the bucket and associated key range. After this step, each worker has all records belonging to a particular key range copied in the final

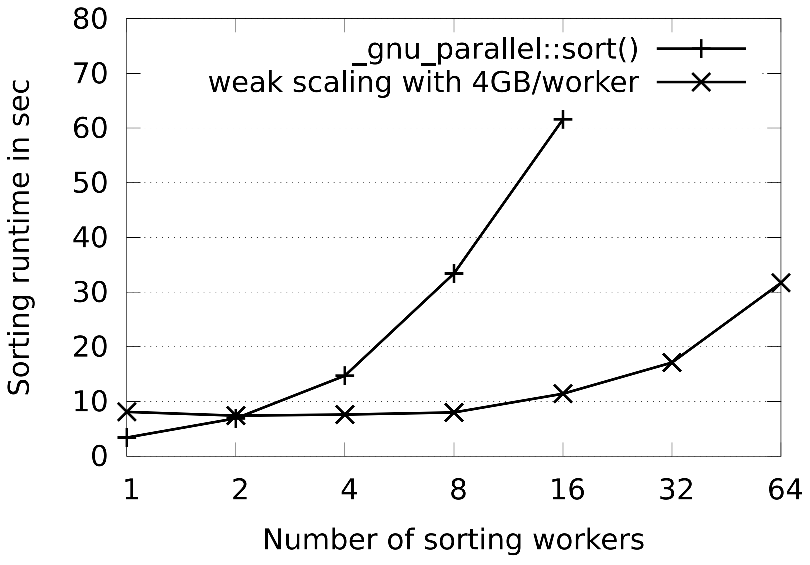


Figure 6.13: Weak scaling performance of RSort.

namespace. A worker then performs a local sorting operation (GNU parallel quick sort) on this assembled record data. After sorting, the data is written out to the final namespace and the worker's status is updated. When all workers have finished writing data in the final namespaces, the sorting is over.

Evaluation: The sorting time reported is the time between the signalling of phase one until phase two is signaled completed by all workers. The numbers reported are the average of three runs.

We evaluate weak-scaling performance of RSort by fixing the per-worker input size to 40 million records of 100 bytes each. The total data size increases with the number of workers. For 64 workers, the total data size is 256 GB[2]. Figure 6.13 shows the scaling performance of RSort. For comparison, we also show the performance of GNU's parallel sorting implementation marked as `_gnu_parallel::sort()`. The local sorting does not have any network IO or resource allocation overheads. On a single server, 64 GB is the largest data size that we can sort in memory. The number of sorting threads are placed first on CPU0 and then on CPU1 in our dual socket machine. Hence, the first 8 threads all reside on CPU0 and with 16 workers, all CPU cores are utilized. As shown in the figure, RSort exhibits very good scaling performance. It scales linearly between 1-8 workers until we host multiple workers per machine. For data sets larger than 8 GB, RSort is 1.9-$5.4\times$ faster than the parallel local sorting.

RSort vs. Hadoop TeraSort: We now compare the performance of RSort with that of Hadoop TeraSort (version 2.2) under similar circumstances running on RAMDisks. We configured

[2]For this sorting section only, we use the benchmark's definition of 10^9 Bytes = 1 GBytes.

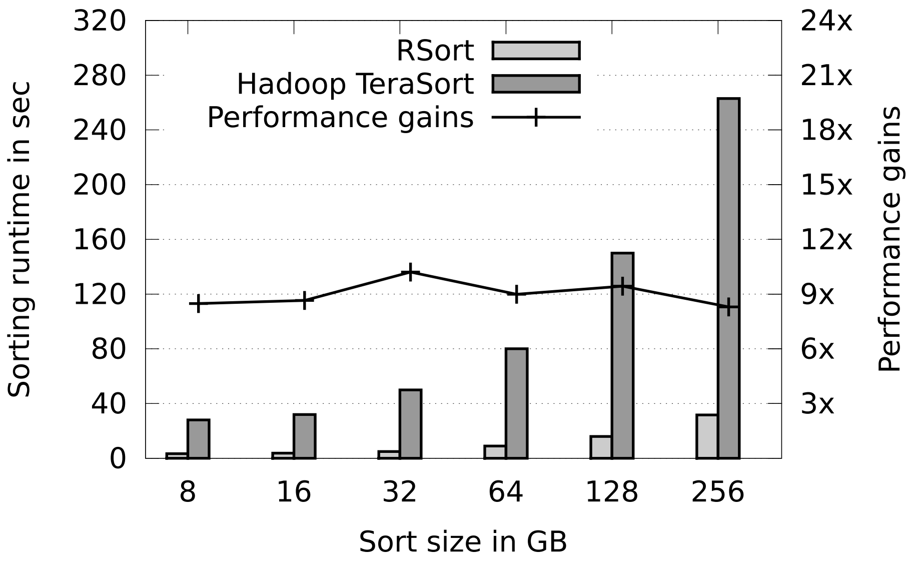


Figure 6.14: Performance of RSort and Hadoop TeraSort.

Property	**Comments/Value**
dfs.block.size, dfs.replication	64 MB, 1
io.file.buffer.size	1 MB
hadoop.temp.dir	on RAMDisk, 128GB
mapreduce.tasktracker.map/reduce.tasks.maximum	32 (no of CPU cores)
mapreduce.job.maps/reduces	256
mapreduce.input.fileinputformat.split.minsize	512 MB
mapreduce.reduce.shuffle.parallelcopies	32
yarn.nodemanager.resource.memory-mb	128 GB
yarn.scheduler.minimum/maximum-allocation-mb	8 GB, 128 GB

Table 6.3: Hadoop TeraSort configuration. The values are configured to give maximum cluster resources to Hadoop.

YARN to give maximum cluster resources (cores and DRAM) to Hadoop. Table 6.3 shows the configuration of our Hadoop cluster. Figure 6.14 compares their performances for variable-size data sorting. As shown, RSort consistently outperforms Hadoop TeraSort by margins of $8 - 10\times$.

6.6 Experiences with RStore

What made RStore and its applications faster? While developing Carafe and RSort, we kept the focus on preserving the separation philosophy of RDMA. This was made possible only by RStore's API that exposes this separation to our applications. Using the API, our applications successfully identified and prepared resources (but not all) upfront in a distributed setting. Hence, we managed to eliminate overheads stemming from unnecessary buffer allocations, memory registration, data touch operations, scheduling, context switches, RDMA connection openings, metadata fetching from the master, etc., from the fast data processing iterations. For data access, Carafe benefited from RStore's highly optimized small IO performance for graph data access; while RSort leveraged RStore's high bandwidth for accessing GBs of KV data.

Was decoupling of allocation from binding effective? Following the principle of decoupling helped us to build an efficient caching layer. The caching layer, which caches raw RDMA resources, not RStore's objects, helped us to amortize the control setup cost that cannot be avoided in the fast data access path. For example, the cache hit rate in phase one of sorting was only 10%. In phase two it was 95%, as it was able to reuse cached RDMA-ready memory from phase one.

Is global synchronization needed on the fast data path? The thin and fast IO path of RStore does not provide any global synchronization between IO requests. We illustrated that applications with their own explicit coordination mechanism can benefit vastly from very fast data access. However, we realize that such a clean and implicit access ordering and path separation is not always possible to achieve. We envision higher-level synchronization primitives, such as transactions as illustrated by FaRM [107], can be built using IO operations from RStore.

6.7 Related Work

Modern RDMA networks are built upon a large body of work from the 1990s [56, 66, 111, 352, 358]. RStore leverages and extends these ideas to deliver high end-to-end performance in a distributed setting. RStore's simple master-servers architecture is inspired by the Google filesystem [141]. RStore's use of network striping to achieve high bandwidth is in a similar spirit to Flat Datacenter Storage [257].

Previous attempts to integrate RDMA into distributed systems have looked into using its offloaded technology and high link speed to compensate for the low performance of the socket/TCP stack. These efforts include transparently using RDMA for socket send/recv [35], integration in traditional file systems [67, 220], and use in the MPI applications [157]. RStore's IO operations are similar to get/put operations in the Separate Memory Model of Partitioned Global Address Space (PGAS) in MPI-3 [157]. RStore's applications can separate communication and storage concerns, and use RStore to store data. Data lives in a separate storage domain than the applications, and can outlive them. The computation and coordination between paral-

Table 6.4: Comparison of related work to integrate RDMA in distributed data storage and processing.

	Application Programming Interface	**Pre-fetching, Caching of Dist. Objs.**	**Unified Bufs. for Zero-copy**	**One-Sided RDMA Ops. Usage**	**Operational Latency**	**Multi-NIC Utilization**	**Failure and Fault Recovery**
RDMA/HDFS [176]	HDFS File IO	API Limitation, Implicit	No	None	N/A, SSD bounded	No	Yes, with SSDs
Key-Value Stores [183, 241, 326]	get/set objects	API Limitation, Implicit	No	Yes, data transfers	multiple variants, 2×RDMA READ RTT [241]	No	Async. SSD Logging
RAMCloud [262, 264]	Tables, Key-Value Store	API Limitation, Implicit	No	None	Send/Recv. RTT	No	Yes, fast recovery
FaRM[107]	TX on Distributed Shared Memory	Possibly, not demonstrated	No	Yes, message passing	RDMA RTT + layout adj.	No	Yes, SSD Logging
RStore	IO on Distributed Shared Memory	Explicit, App-driven API calls	Yes	Yes, data transfers	RDMA RTT	Yes, IO striping	COW mode (not evaluated)

lel jobs, which is not tied to data storage, are handled separately by application-specific logic where MPI-3 abstractions can be used. The work from Islam et al. is one of the first to propose the integration of InfiniBand network into the HDFS design [176]. However, their proposal does not use one-sided RDMA operations, performs data copies, and suffers from inefficiencies found in HDFS/Java stack for high-performance networks [325, 327]. Recent interest in RDMA has led to many Key-Value store implementations [180, 241, 326]. However, due to the limitations of the API, these systems do not expose or give control of resource allocation to applications. Furthermore, RStore, in comparison, is a more general-purpose RDMA memory management platform on top of which these applications can be built. soNUMA provides a transparent, distributed coherent shared memory abstraction using a restricted form of RDMA-based protocol [259]. However, it required modifications to the host memory controller.

The recently proposed FaRM system is closest to our approach [107]. It provides a general computing platform that leverages RDMA to provide high-performance transactions and lock-free reads to applications. However, FaRM's opaque object API does not let applications pre-allocate or pre-set expensive RDMA resources to avoid their overheads in the object access. Furthermore, FaRM's internal representation of objects (laced with metadata at every cache line) is different from an application's view. Consequently, FaRM must either copy or adjust the layout before giving access to an application. For small object requests (a few KBs), for which FaRM delivers good performance, overheads from DRAM management, together with layout adjustments (data touch operations), are small. For large data accesses (MBs or GBs), however, these overheads can easily dominate any performance gains. FaRM's transactional object API is arguably better for storing critical system metadata.

Other general-purpose, distributed, in-memory abstractions include RAMCloud [264], Sinfonia [23], and Resilient Distributed Datasets [367]. RAMCloud aims to provide very-low data-access latencies by storing entire data sets in DRAM, and aggregating main memories across hundreds of servers. It uses InfiniBand, a high-performance interconnect for fast inter-machine communication, which is limited to fast message passing only [262]. Sinfonia and RDD, despite storing data in memory, do not leverage RDMA for data access, although they provide more facilities such as fault-tolerance, durability, and straggler handling, etc., that RStore lacks. Table 6.4 compares RStore to other high-performance network integration efforts over properties that are discussed throughout this chapter.

6.8 Conclusion

In this chapter, we have presented RStore, a distributed, in-memory data store that delivers performance (12 μsec latency, 705 Gb/s aggregate bandwidth on an iWARP cluster) very close to the network limits. RStore adheres to the separation philosophy of RDMA networks and is built using two design principles: (a) decouple resource allocation from its abstraction binding; (b) keep the data access path thin and fast. RStore's API exposes and extends these ideas to applications which benefit from pre-allocating and pre-fetching resources in a distributed setting. We demonstrated RStore's capabilities by developing two types of applications on it. Our first application, Carafe, which is a low-latency graph processing system, outperformed state-of-the-art systems by margins of 2.6-$4.2\times$. Our second application, RSort, is an order of magnitude faster than Hadoop TeraSort for sorting distributed key-value tuples. The key reasons for the high application performance are the design and abstraction choices that RStore is based upon.

7 RStore on FlashNet

As a natural extension of the work presented so far in the thesis, in this chapter, we combine efforts on storage and distributed system fronts by running RStore on FlashNet. RStore is a good fit as an application for FlashNet for to multiple reasons. First, it is a fully RDMA-enabled distributed data store that leverages the RDMA separation philosophy in every facet of its design. Memory servers and client machines, which are involved in an IO operation, set up IO resources in advance on the distributed control path. The data transfers happen on a separate data transfer path using one-sided RDMA operations. By having a clean separation, RStore's control and data paths naturally extend to FlashNet. This extension establishes *end-to-end* distributed control and data paths between application buffers and remote flash devices. Such setup allows RStore to prepare flash buffers from multiple flash devices and access data from them transparently using its API's control and data operations.

Second, many other distributed storage systems such as RAMCloud or FaRM, which also use RDMA for data transfers between DRAM buffers, distinguish between DRAM buffer locations and persistent storage data locations. This distinction and the use of the two-tier memory hierarchy necessitate an application intervention to stage data between in-memory buffers and storage locations for servicing a network-storage IO request. In contrast, RStore assumes a single-tier, unified memory hierarchy. This unified view allows RStore's clients to access data completely transparently from remote DRAM buffers or flash devices without involving the server application.

And lastly, RStore is designed and implemented in the same Linux OFED RDMA framework as the FlashNet stack, thus facilitating a rapid integration. Other publicly available RDMA-enabled systems, such as RAMCloud [11] and HERD Key-Value store [7], etc., do not use the same framework and require additional efforts for porting. Other potential systems such as RDMA-enabled file systems e.g., GlusterFS [6] and NFS [67], etc., only use RDMA for network IO between their transport buffers. Extending the RDMA path of these file systems to flash does not deliver the full benefit as they still require the explicit logic to flush transport buffers to storage devices.

7.1 Porting RStore to FlashNet

We have modified and ported RStore to run on FlashNet. The modified RStore now supports storing data in remote flash devices together with the currently supported DRAM buffers. The porting effort was straight forward and required less than 1% lines of code changes. On the memory server side, the majority of changes focus on acquiring RDMA-ready flash buffers by `mmap`ing ContigFS files. A server now supports two types of storage buffers. The DRAM storage buffers are allocated using anonymous `mmap` with huge pages as described in Section 6.3. The flash storage buffers are allocated using a file `mmap` on files stored on ContigFS. Changes on the RStore master side aim at supporting a new type of *flash* namespace. Resource allocation calls (`reserve()` and `alloc()`) in a flash namespace only use flash buffers. Like DRAM buffers pools, flash buffers are managed internally in a multiple of chunks. Client-side memory obtained from the `map()` call on an `raddress` object is still DRAM backed.

Accordingly, the RStore API is updated to reflect these changes. Namespace creation (`create_ns(string name, enum ns_scope=DRAM)`) now takes a scope flag that requests the master to create a namespace either in the DRAM or the flash scope. The default scope is DRAM. There are no further client-visible changes in the RStore API when extending it to include flash namespaces. Subsequent resource allocations in a newly created namespace use chunks from the associated flash or DRAM chunk pool. Data from these locations are always accessed transparently using one-sided RDMA operations. The client-side library is not aware if data is accessed from a remote DRAM or flash buffer. In the subsequent subsection, we refer to ported RStore as RStore/FlashNet.

7.2 The Cost of the Distributed Control Path

We now quantify the cost of the extended and distributed control operations, namely `reserve()` and `alloc()` of the RStore/FlashNet system. As described in Table 6.1, the `reserve()` operation allocates and prepares DRAM buffers on multiple servers for RDMA accesses. For RStore, as presented in Chapter 6, the preparation phase includes allocation, registration, and pinning of DRAM buffers with the local RDMA provider. For RStore/FlashNet, the `reserve()` call includes creating a file on ContigFS, truncating it to give it the requested reserve size, `mmap`ing the file, and then registering it with the FlashNet RDMA controller. However, the `alloc()` operation, which stitches flash or DRAM chunks within an `raddress` region, is mostly unchanged. The key change includes the resolution of the right pool type from where chunk allocations happen to the DRAM or the flash pool.

For all experiments, we use 4 machines from the 9-machine testbed (Table 5.3) used in

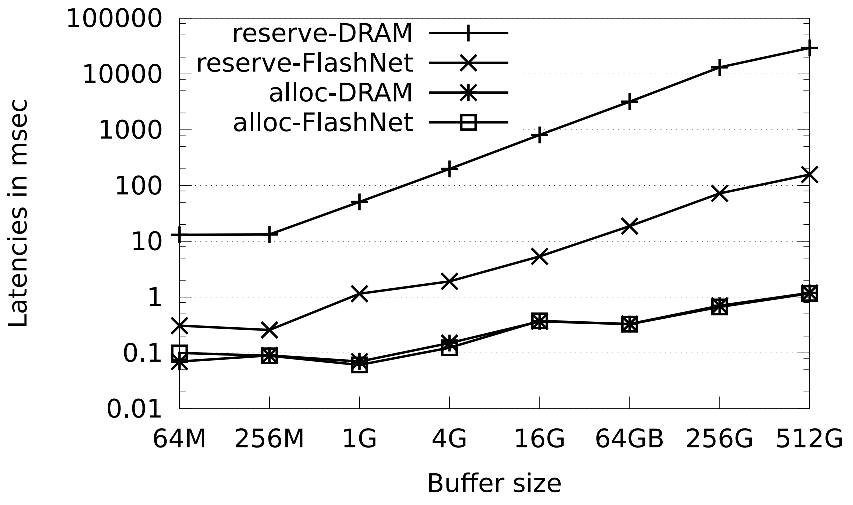


Figure 7.1: Control setup cost for a single RStore/FlashNet client.

Chapter 5. 3 out of 4 machines contain Intel NVMe devices with the base-line performance of 2.2 GB/s and 1.1 GB/s for reads and writes, respectively. The one remaining machine contains the unnamed enterprise-grade PCIe-attached flash device used for experiments in Chapter 5.

Figure 7.1 shows the performance of extended and distributed control path operations for a single client. The x-axis shows the buffer size and the y-axis shows the operation time in milliseconds. For reference purposes, the figure also shows the operation time for DRAM-based buffers. There are two observations which can be made from the figure. First, the `reserve()` operation in a flash-scope namespace is 1–2 orders of magnitude faster than a DRAM-scope namespace operation. This is due to the fact that unlike a conventional RDMA provider, the FlashNet RDMA controller does not pin pages for ContigFS-backed files during the memory registration call (Section 5.4.1). Other operations such as the file creation and truncation calls on ContigFS, are also fast and only involve a minimum amount of meta-data changes. In contrast, an `mmap` call into Linux costs significantly more due to shared locking on the huge page pool and zero-ing out page cost due to security reasons (Section 6.4.1). Second, the `alloc()` operation cost remains unchanged between a flash-scoped and a DRAM-scoped namespace. This behaviour is largely expected as the `alloc()` operation executes completely on the master and creates new abstraction bindings for chunks (either DRAM or flash) within an `raddress` region. The key performance numbers from this single client experiment are that on our 4-machine cluster, a single RStore/FlashNet client can `reserve()` and `alloc()` 512 GB of flash capacity in 157.3 msecs and 1.1 msecs, respectively.

For our next set of experiments, we focus on the scaling of the extended and distributed `reserve()` and `alloc()` operations in presence of multiple clients. We quantify both weak

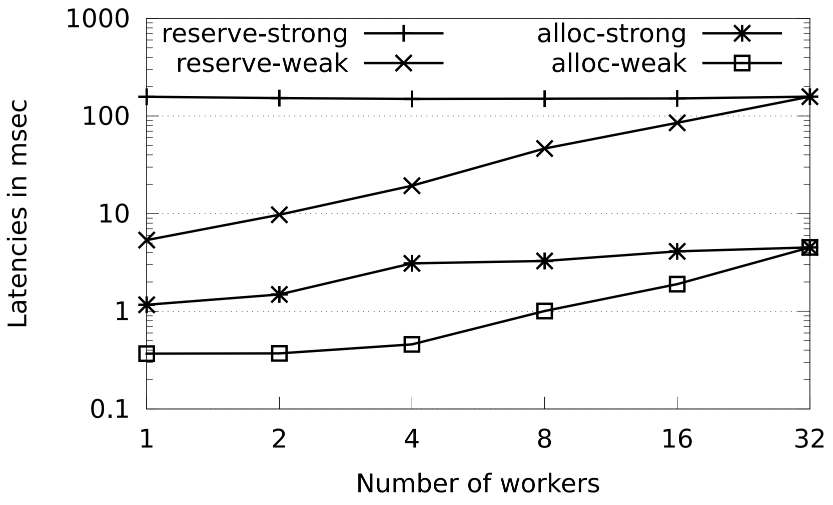


Figure 7.2: Scaling of distributed and extended `reserve()` operation.

and strong scaling performance of RStore/FlashNet. For strong scaling, we take the total capacity of 512 GB from the last single client experiment and increase the number of concurrent clients by dividing the work (allocation of 512 GB of flash capacity) among them. For weak scaling, we fix the reservation size per client to 16 GB. With 32 clients, the weak-scaling experiment also prepares 512 GB of flash memory. Figure 7.2 shows our findings. There are two main takeaway messages here. First, for the weak scaling, the cost of the flash buffer `reserve()` call increases (so does the amount of work done) linearly with the number of clients. The strong scaling cost remains the same because RStore/FlashNet efficiently distributes a `reserve()` call among all available servers independently from the number of concurrent clients. Hence, the reported performance numbers stay close to the 157.1 msecs mark. Second, the `alloc()` cost increases gradually with the number of concurrent clients as the `alloc()` operations involve shared locks on the master.

To summarize, the ported RStore/FlashNet system retains the efficiency of the distributed control setup paths of the original system. The numbers reported in this section for the *extended* and *distributed* control path operations represent a small (100s of msecs) but one time cost that needs to be paid when setting up the control resources for the first time in a distributed setting. For repeated executions, the performance of these operations could be further improved by the use of RStore's resource caching layer.

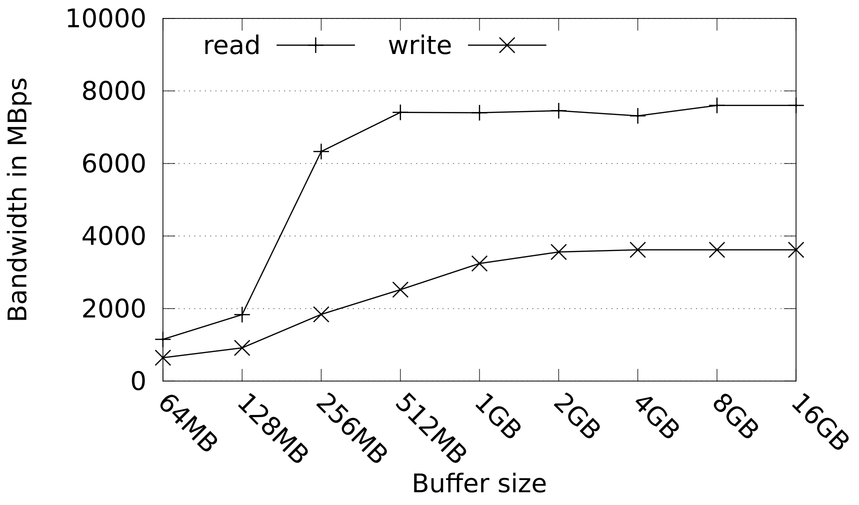


Figure 7.3: Single-client `read()` and `write()` bandwidth for RStore/FlashNet operations.

7.3 Performance of Data-Path Operations

We now measure the performance of the extended and distributed data path operations, namely `read()` and `write()`. These operations are split internally into chunk-sized, one-sided RDMA operations by the RStore's client-side library. The default allocation policy of RStore is round-robin allocation among servers or devices. Hence, the chunks are evenly distributed among the 4 flash devices that we have in our 4 machines. For example, for 512 GB of flash space, each device serves 2048 64MB-sized chunks. Figure 7.3 shows the performance of IO operations for `read()` and `write()` operations. As expected, the delivered read and write bandwidths improve as the operation size increases due to the increased opportunity for parallelism. For sizes greater than 512MB, all 4 devices operate at 95% device capacity. The aggregate cluster throughput in presence of multiple clients also exhibits a similar performance behaviour.

7.4 Running RSort on FlashNet

We have also modified RSort, the distributed key-value sorting application of RStore, to run on RStore/FlashNet. The modified RSort reads the input data from a flash namespace, processes it in buckets stored in DRAM namespaces, and then writes out the sorted data back to a flash namespace. We compare the performance of RSort to the native in-memory version of it and Spark/TeraSort in similar configurations. The Spark configuration is set to get all memory and CPU cores from the YARN cluster resource manager. As RSort and Spark/TeraSort are implemented in different manners, we used a fixed-work per worker model for a fair comparison.

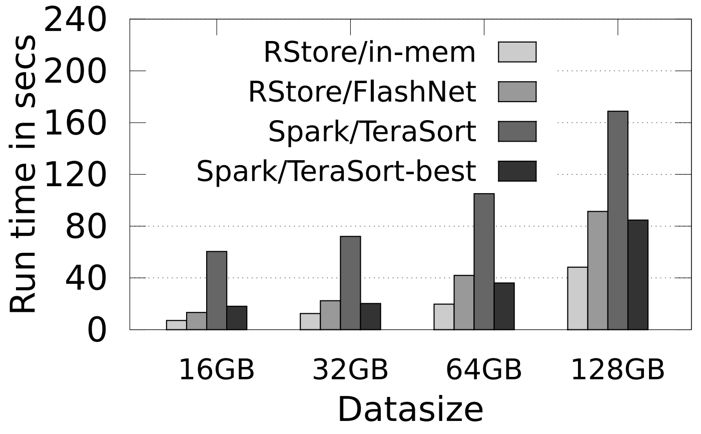


Figure 7.4: Distributed sorting performance.

In this model, every worker in the system (Spark or RStore) processes 4 GB raw data or 40 million key-value pairs. The number of workers varies and depends upon the data size. For 16GB, there are a total of 4 workers on 4 machines (1 worker/machine). For 128 GB, there are a total of 32 workers, 8 workers per machine. We run Spark/TeraSort in the fixed-work model by setting the input `splitsize` to 4 GB. We also show the best configuration performance for Spark/TeraSort that we can obtain on our machines. In all its settings, Spark/TeraSort uses Java's NIO API for networking IO that is a socket-equivalent communication abstraction within the Java virtual machine (JVM).

Figure 7.4 shows our performance numbers. There are three noteworthy results that can be derived from the numbers showed in the graph. First, in comparison to the identical in-memory execution of RSort, the RStore/FlashNet mostly adds the time of IO from the flash devices and does not incur any additional overheads. Second, in the fix-work-per-worker mode, RSort performs 44-77% better than Spark/TeraSort. Third, in the best configuration Spark/TeraSort outperforms RSort by 7-13%.

We exercise caution in interpreting these performance gains (or losses) as they are results of various design decisions made in both systems. The Spark compute engine is highly optimized for waves of single-threaded small tasks where IO is overlapped with a very efficient sorting implementation. In contrast, RSort implements a strict distributed two-phase external bucket sort, where the next wave of computation does not start until IO from the last phase is finished. Hence, gains of RSort in the fixed-work model come from the limited compute/IO overlap opportunities experienced by Spark/TeraSort. These opportunities are increased (hence the improved performance) in the best configuration. Moreover, Spark/TeraSort computation is locality-based and does negligible network IO. In contrast, RSort stripes data across multiple machines to leverage network and device parallelism while stressing network and storage devices. Furthermore, Spark/TeraSort does asynchronous IO where writes are gradually flushed by HDFS over a long period of time. In contrast, FlashNet writes are flushed immediately which affects IO completion times. Hence, for 7-13% performance overheads, RStore/FlashNet offers fully synchronous IO operations for reading and writing data. These times could be further improved by implementing a caching and buffering layer on top of FlashNet. Currently, all IO requests on FlashNet are direct IO requests which are served directly by a flash device.

Consequently, a direct comparison between RSort's static and Spark/TeraSort's dynamic compute frameworks is beyond the scope of this thesis. Instead, the key takeaway message that we push for is that RDMA-ready systems such as RStore can be ported and executed on FlashNet with minimum efforts. This porting extends the scope of their benefits to data residing in flash devices.

8 Conclusion

Systems builders traditionally assume a *fast* CPU with a few *slow* IO devices, such as network and storage devices. This assumption has led to designs of mechanisms and abstractions in systems software where the CPU is kept busy by executing various OS services and IO routines from multiple processes while slow IO operations are in progress. However, in the past decade, the raw IO performance of network and storage devices has improved significantly, while CPU speed improvements have stalled. This role reversal concerning the performance assumption of these components is a fundamental shift that affects the way we should think and build systems software. In this thesis we identify excessive CPU and systems software involvement as a performance bottleneck in the IO processing path on high-performance network and storage devices. We advocate to curtail this involvement by using the separation principle from the area of high-performance networking. We apply the principle and its philosophy, originally developed for parallel computing, to general-purpose commodity computing systems and evaluate its effectiveness.

In the first part of the thesis, we make a case for extending the separation philosophy and associated interfaces and abstractions (e.g. IO queues, asynchronous IO posting, notification channels, etc.) to access high-performance storage devices. We then develop a proof-of-concept prototype called FlashNet that unifies high-performance network properties with flash storage access and management, thus eliminating excessive application and OS involvement from IO processing. The unified FlashNet device contains a software Remote Direct Memory Access (RDMA) controller and a software flash controller. Furthermore, by designing the network and flash controllers together, FlashNet enhances RDMA network operations and makes them more amenable to the flash storage. FlashNet delivers up to $50 - 200\%$ better IO operations/sec (IOPS) performance than the traditional solutions for transferring data between the storage and networking stacks. However, being a software-only solution it focuses solely on restricting an operating system's involvement in a cross-stack IO processing by smartly connecting abstractions from networking to storage devices.

In the second part of the thesis, we leverage hardware capabilities to eliminate the CPU involvement in network IO processing and build RStore, a distributed in-memory data store. The key challenge in building RStore was to design a system that supports end-to-end separation of data and control paths in presence of multiple clients and servers. In such a distributed environment a single IO request can affect resources scattered across multiple servers and hence, any system design needs to consider resource-setup and data-transfer paths in an end-to-end manner. This challenge was addressed by developing a unique memory-like API and storage abstractions that let an application pre-allocate and set up distributed resources prior to an access to them. RStore further classifies resources into system- and abstraction-related resources and is built using two design principles: (i) decouple system resource allocation from its abstraction binding, and (ii) keep the data access path thin and fast without any implicit heavy-weight global synchronization operations. The decoupling principle helped to build a smart resource management layer with caching to amortize the control setup cost that cannot be avoided in the fast data access path. The thin data access path helps to deliver full network performance to applications. We envision that applications can be provided full global ordering and synchronization through out-of-band services built using RDMA operations, e.g., DARE SMR system [278]. We also built a primitive version of such a service, aiming to provide global coordination for bulk synchronization operations. Our two applications, a distributed Key-Value sorter and a distributed graph processing engine, leverage RStore's API to pre-set up storage and network resources and then access data that is striped, distributed and stored in DRAMs of participating memory servers. RStore delivered a high aggregate bandwidth (705 Gb/s) and close-to-hardware latency on our 12-machine testbed. The graph-processing framework, which relies on RStore for low-latency graph access, outperforms state-of-the-art systems by margins of $2.6 - 4.2\times$ when calculating PageRank. The Key-Value sorter can sort 256 GB of data in 31.7 sec, which is $8\times$ better than Hadoop TeraSort in a similar setting. The porting of RStore on FlashNet requires minimum efforts and FlashNet adds negligible overheads to RDMA operations used by RStore for data accesses.

8.1 Experience and Outlook

From a system organization point of view, performance and flexibility typically occupy opposite positions on the design spectrum. On the one side, traditional interfaces, such as UNIX file descriptors, are flexible and simple enough to facilitate rapid application development. They can obtain and release resources on demand and perform heavy-lifting during IO operations by organizing IO buffers, hide the complexity of device management, adaptively choose the best IO strategy, improve utilization of the system, etc. Since their inception in 1974 with the UNIX system, researchers have amassed a large amount of expertise and experience with them. On the other extreme, approaches such as FlashNet, RStore and their applications trade flexibility for performance. They push to identify and pre-allocate as many resources as possible, thus requiring them to manage all resources and complexity in the application code. Pre-allocation also commits resources to a particular workload, which might under-utilize them. This dichotomy that is deeply embedded in application programming interfaces (APIs) and abstractions, forces developers to choose one over the other at the application design time. Ideally, applications should be able to trade off flexibility for performance gains at run-time, and a system should make necessary adjustments by pre-allocating some resources while leaving others. However, this would require support from devices and also across the entire application stack, including more expressive APIs and abstractions. Recent steps towards *software-defined environments* feature some necessary tools and hooks to facilitate separation even in a distributed setting, also including switches, routers, and storage controllers as well [336]. For a more holistic environment, we need to integrate these components into modern distributed resource management frameworks [346, 348] to set up an end-to-end fast data path.

As discussed, the application programming interface (API) of Remote Direct Memory Access (RDMA)-capable networks differ vastly from the BSD socket/TCP. Instead of having a simple file descriptor where all IO operations and notifications happen, the RDMA API provides abstractions and mechanisms for memory buffers, IO posting and completion queues, connection management logic, notification strategies (polling vs. blocking), etc. These abstractions achieve separation by pre-allocating resources and eliminating overheads from IO accesses. However, using this new IO API also necessitates re-writing much of the storage and computing infrastructure to demonstrate its effectiveness. For example, the development of RStore entailed designing not only the distributed store but also the relevant distributed computing applications. We could not leverage already written applications because (i) they do not use the same API, and (ii) were not designed to identify and pre-allocate resources separately from the computation. Consequently, in the limited time frame of this thesis work, we develop a system from scratch and focus on performance-specific optimizations while omitting other more pertinent features that would be of relevance in a production environment. Some of these features, which are absent from RStore and its applications, are fault tolerance, failure recovery, generic workload management with partitioning and load-balancing, multi-tenancy, quality of service, virtualization support, isolation, security, and access control.

Over the course of this thesis, we realized that even for a relatively small experimental system, hardware failures and bugs are real possibilities. We have witnessed bugs that included memory failures, RNIC bugs, PCI-compliance issues, CPU faults, and a misbehaving switch. The RNIC bugs were the hardest to resolve because of the lack of proper documentation, a complex driver, and the offloaded nature of IO operations. Additional challenges came from the fact that these bugs manifest themselves as a completely unrelated problem in the distributed storage. For example, in one RSort run, the data verification step failed. We started with debugging our implementation of distributed sorting, then the RStore IO path implementation, and then subsequently focused to its data transfer logic. However, after days of debugging, we identified the RNIC as the culprit: in certain circumstances, the RNIC transmitted bogus get-location() RPC data because of a wrong offset calculation in a RPC buffer which used huge pages instead of 4kB pages. RStore clients, oblivious of this glitch, wrote to the wrong data locations, hence corrupting previously written data. In these circumstances, having a software RNIC device (SoftiWARP) was immensely helpful to confidently locate these bugs in the RNIC. Thankfully, we got tremendous support from our RNIC vendor, who readily located and fixed these issues in the RNIC firmware. Nonetheless, the incident shows the need for a clean and standard support to resolve such issues. Over the years, vendors have added support for performance optimization and debugging on the CPU. We envision a similar support for emerging capable and powerful network and storage devices.

8.2 Recommendations for Future Hardware and Stacks

In this section, we give some recommendations to future hardware and stack developers. Our recommendations come from our own experience from developing RDMA applications for the network and extending RDMA semantics for storage. The general theme goes along the lines of separating policies from mechanisms and not enforcing arbitrary limits [246]. Part of these recommendations aim at RNIC vendors; others at the industry consortium of the Open Fabric Alliance (OFA), which maintains the popular OFED RDMA stack for GNU/Linux and Microsoft Windows platforms.

8.2.1 Resource Management

Memory management is one of the key operations in a distributed storage system. As we expand RDMA operations to Non-Volatile Memory (NVM) storage, we need more efficient ways to manage physical resources for higher-level file operations. For example, truncating or expanding files are common file operations supported by many file systems including ContigFS (see Section 5.2.2). Consequently, truncation and expansion of file-backed RDMA memory regions should also be supported without unreasonable overheads. Although one can de-register the old region and re-register an area reflecting its new size, this action revokes the old RDMA STag, which may still be in use at other clients. We expect a generic mechanism that allows the growing or shrinking of an already registered area while preserving its RDMA credentials. Furthermore, current RDMA stacks and implementations lack a way to expose RDMA resources beyond a single-process scope to multiple processes that might share NVM memory regions.

All IO buffers in an RDMA environment are pre-allocated and identified by a 32-bit identifier called STag. These identifiers are generated by an RNIC during a memory registration operation without taking any inputs/hints from an application. This is analogous to a key-value system where keys cannot be controlled by the client who uses them to index into its data structures. Hence, to simplify the development of such applications, we strongly recommend to let applications choose their own identifier keys. An RNIC can still reject the registration if the key already exists. Moreover, a 32-bit identifier, which is further divided in two sections of 8 bits and 24 bits, is not sufficient to address all possible memory buffers in a system. We suggest to increase this buffer identifier to 64 or 128 bits and possibly to include a user-specific field that can be used by applications for message multiplexing or autonomous procedure execution in a similar spirit as Active Messages [353].

8.2.2 IO Operations

RDMA semantics offer a direct zero-copy data-transfer mechanism from the data source to the sink. With the recent interest in the RDMA technology, it has been applied in a variety of systems including networked servers for applications such as Key-Value stores. To maintain data consistency, RNIC operations must coordinate accesses with the CPU accesses for data updates. Although it is still possible to build a working system as demonstrated by systems such as Pilaf [241] and HERD [183], some additional support from the RNIC can immensely simplify the design of such systems.

In this regard, we expect support for RNIC in the form of suspension and resumption of IO operations on a particular IO region, precise and deterministic cancellation of IO operations to rollback and maintain consistency, and in the case of concurrency, a possibility to specify execution and notification ordering with barrier and fencing operations. Having large numbers of unordered IOs has been shown to deliver high performance with NVM storage devices [69]. Supporting these operations will necessitate integrating RNICs more closely with the CPU architecture, and perhaps also integrating and leveraging the same synchronization mechanisms that are used between the many cores of a CPU.

8.3 Summary

Data-crunching application frameworks expect IO performance improvements when running on modern high-performance network and storage devices. However, the delivered performance is limited by the cumbersome host CPU and operating system involvement in the IO processing path. This thesis presents the design of an end-host storage and a distributed system stack to bring applications closer to the IO hardware by leveraging the separation principle from user-space networking stacks. We demonstrate that the proposed designs deliver good IO performance and provide a solid foundation for building future network-centric, data-intensive distributed systems. We envision that some of our findings and recommendations find their way into popular application frameworks, such as MapReduce or Spark, and benefit their users without requiring a complete stack rewrite.

Bibliography

[1] Apache Hadoop. `https://hadoop.apache.org/`.

[2] Apache Spark - Lightning-Fast Cluster Computing. `http://spark.apache.org/`.

[3] Facebook processes more than 500 TB of data daily, at `http://www.cnet.com/news/facebook-processes-more-than-500-tb-of-data-daily/`.

[4] FAQ: MongoDB Storage, section MMAPv1 Storage Engine, at `http://docs.mongodb.org/v3.0/faq/storage/`.

[5] GASNet Communication System, at `http://gasnet.lbl.gov/`.

[6] GlusterFS, at `http://www.gluster.org/`.

[7] HERD, a highly efficient key-value system for RDMA, at `https://github.com/efficient/HERD`.

[8] LiveJournal social network at Stanford Network Analysis Project, `https://snap.stanford.edu/data/soc-LiveJournal1.html`.

[9] Netperf: A network performance benchmark. `http://www.netperf.org`.

[10] NVMExpress, The Optimized PCI-Express SSD Interface, `http://www.nvmexpress.org/`.

[11] Official RAMCloud repo, at `https://github.com/PlatformLab/RAMCloud`.

[12] perf: Linux profiling with performance counters. `http://perf.wiki.kernel.org/`.

[13] Redis KV store, `http://redis.io/`.

[14] SCSI Express, `http://www.scsita.org/library/scsi-express/`.

[15] Soft RDMA over Ethernet (RoCE) Driver. `https://github.com/SoftRoCE`.

[16] The Open Fabric Alliance (OFA), `https://www.openfabrics.org/`.

[17] Violin and Microsoft's High-Performance, All-Flash Enterprise Storage, `http://www.violin-memory.com/blog/violin-and-microsoft-windows-flash-array/2/`.

[18] Yaffs (Yet Another Flash File System), `http://www.yaffs.net/`.

[19] Abbott, M. B. and Peterson, L. L. (1993). A Language-based Approach to Protocol Implementation. *IEEE/ACM Trans. Netw.*, 1(1):4–19.

[20] Accetta, M. J., Baron, R. V., Bolosky, W. J., Golub, D. B., Rashid, R. F., Tevanian, A., and Young, M. (1986). Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer Conference, Altanta, GA, USA, June 1986*, pages 93–113.

[21] Agarwal, A., Bianchiniand, R., Chaikenand, D., Johnson, K. L., Kranz, D., Kubiatowicz, J., Lim, B.-H., Mackenzie, K., and Yeung, D. (1995). The MIT Alewife machine: architecture and performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA'95, pages 2–13.

[22] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R. (2008). Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX 2008 Annual Technical Conference*, ATC'08, pages 57–70, Boston, MA, USA.

[23] Aguilera, M. K., Merchant, A., Shah, M., Veitch, A., and Karamanolis, C. (2007). Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 159–174, Stevenson, WA, USA.

[24] Ahmad, I., Gulati, A., and Mashtizadeh, A. (2011). vIC: Interrupt Coalescing for Virtual Machine Storage Device IO. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ATC '11, pages 45–58, Portland, OR, USA.

[25] Akel, A., Caulfield, A. M., Mollov, T. I., Gupta, R. K., and Swanson, S. (2011). Onyx: A Protoype Phase Change Memory Storage Array. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'11, Portland, OR, USA.

[26] Anderson, E., Brooks, J., Grassl, C., and Scott, S. (1997). Performance of the CRAY T3E Multiprocessor. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, SC '97, pages 1–17, San Jose, CA, USA.

[27] Anderson, E. and Tucek, J. (2010). Efficiency matters! *SIGOPS Oper. Syst. Rev.*, 44(1):40–45.

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

[29] Arnould, E., Kung, H. T., Bitz, F., Sansom, R. D., and Cooperm, E. C. (1989). The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, pages 205–216, Boston, MA, USA.

[30] Aron, M. and Druschel, P. (1999). Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 232–246, Charleston, SC, USA.

[31] Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., Burnett, N. C., Denehy, T. E., Engle, T. J., Gunawi, H. S., Nugent, J. A., and Popovici, F. I. (2003). Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 90–105, Bolton Landing, NY, USA.

[32] Bachand, D., Bilgin, S., Greiner, R., Hammarlund, P., Hill, D. L., Huff, T., Kulick, S., and Safranek, R. (2010). The Uncore: A Modular Approach to Feeding The High-Performance Cores. In *Intel Technology Journal*, Volume 14, Issue 3, pages 30–49.

[33] Badam, A. and Pai, V. S. (2011). SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 211–224, Boston, MA, USA.

[34] Bailey, K., Ceze, L., Gribble, S. D., and Levy, H. M. (2011). Operating System Implications of Fast, Cheap, Non-volatile Memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 2–2, Napa, CA, USA.

[35] Balaji, P., Narravula, S., Vaidyanathan, K., Krishnamoorthy, S., Wu, J., and Panda, D. K. (2004a). Sockets Direct Protocol over InfiniBand in Clusters: Is It Beneficial? In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '04, pages 28–35.

[36] Balaji, P., Shah, H. V., and Panda, D. K. (2004b). Sockets vs RDMA Interface over 10-Gigabit Networks: An In-depth analysis of the Memory Traffic Bottleneck. In *Proceedings of the IEEE Cluster RAIT (RDMA Applications, Implementations, and Technologies) Workshop*.

[37] Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobber, T., Wei, M., and Davis, J. D. (2012). CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th USENIX Conference on Networked Systems*

Design and Implementation, NSDI'12, pages 1–14, San Jose, CA, USA.

[38] Banikazemi, M., Moorthy, V., Herger, L., Panda, D., and Abali, B. (2000). Efficient virtual interface architecture (VIA) support for the IBM SP switch-connected NT clusters. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, IPDPS 2000, pages 33–42.

[39] Banks, D. and Prudence, M. (1993). A high-performance network architecture for a PA-RISC workstation. *IEEE Journal on Selected Areas in Communications*, 11(2):191–202.

[40] Bates, S. Donard: NVM Express for Peer-2-Peer between SSDs and other PCIe Devices, `http://www.snia.org/sites/default/files/SDC15_presentations/nvme_fab/StephenBates_Donard_NVM_Express_Peer-2_Peer.pdf`.

[41] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schupbach, A., and Singhania, A. (2009). The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, Big Sky, MT, USA.

[42] Beckmann, N. Z., Gruenwald III, C., Johnson, C. R., Kasture, H., Sironi, F., Agarwal, A., Kaashoek, M. F., and Zeldovich, N. (2014). PIKA: A Network Service for Multikernel Operating Systems, Technical report MIT-CSAIL-TR-2014-002 at Parallel and Distributed Operating Systems group, MIT, `http://dspace.mit.edu/handle/1721.1/84608`.

[43] Beecroft, J., Homewood, M., and McLaren, M. (1994). Meiko CS-2 Interconnect Elan-Elite Design. *Parallel Computuers*, 20(10-11):1627–1638.

[44] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. (2014). IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Broomfield, CO, USA.

[45] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. (1995). Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, Copper Mountain, Colorado, USA.

[46] Bestler, C. and Stewart, R. (2007). Stream Control Transmission Protocol (SCTP) Direct Data Placement (DDP) Adaptation. RFC 5043, RFC Editor.

[47] Bhandari, Kumud; Chakrabarti, Dhruva R.; Boehm, Hans- J. Implications of CPU Caching on Byte-addressable Non-Volatile Memory Programming, HP Laboratories technical report number HPL-2012-236, at `http://www.hpl.hp.com/techreports/2012/HPL-2012-236.html`.

[48] Bhaskaran, M. S., Xu, J., and Swanson, S. (2014). Bankshot: Caching Slow Storage in Fast Non-volatile Memory. *SIGOPS Oper. Syst. Rev.*, 48(1):73–81.

[49] Biagioni, E. (1994). A Structured TCP in Standard ML. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 36–45, London, United Kingdom.

[50] Bilas, A. and Felten, E. W. (1997). Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. *J. Parallel Distrib. Comput.*, 40(1):138–146.

[51] Binkert, N. L., Hsu, L. R., Saidi, A. G., Dreslinski, R. G., Schultz, A. L., and Reinhardt, S. K. (2005). Performance analysis of system overheads in TCP/IP workloads. In *Proceeding of the 14th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 218–228.

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

[53] Bjørling, M., Axboe, J., Nellans, D., and Bonnet, P. (2013). Linux Block IO: Introducing Multi-queue

SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 22:1–22:10, Haifa, Israel.

[54] Blackwell, T. (1996). Speeding Up Protocols for Small Messages. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '96, pages 85–95, Palo Alto, CA, USA.

[55] Blumrich, M. A., Dubnicki, C., Felten, E. W., and Li, K. (1996). Protected, User-level DMA for the SHRIMP Network Interface. In *Proceedings of the 2Nd IEEE Symposium on High-Performance Computer Architecture*, HPCA '96, pages 154–165.

[56] Blumrich, M. A., Li, K., Alpert, R., Dubnicki, C., Felten, E. W., and Sandberg, J. (1994). Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pages 142–153, Chicago, IL, USA.

[57] Boden, N. J., Cohen, D., Felderman, R. E., Kulawik, A. E., Seitz, C. L., Seizovic, J. N., and Su, W.-K. (1995). Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36.

[58] Bonachea, D., Hargrove, P. H., Welcome, M., , and Yelick, K. (2009). Porting GASNet to Portals: Partitioned Global Address Space (PGAS) Language Support for the Cray XT. In *Proceedings of the Cray User Group (CUG)*, CUG'09.

[59] Borkar, S., Cohn, R., Cox, G., Gross, T., Kung, H. T., Lam, M., Levine, M., Moore, B., Moore, W., Peterson, C., Susman, J., Sutton, J., Urbanski, J., and Webb, J. (1990). Supporting Systolic and Memory Communication in iWarp. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 70–81, Seattle, WA, USA.

[60] Brewer, E. A., Chong, F. T., Liu, L. T., Sharma, S. D., and Kubiatowicz, J. D. (1995). Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 42–53, Santa Barbara, CA, USA.

[61] Brightwell, R. and Maccabe, A. (2000). Scalability Limitations of VIA-based Technologies in supporting MPI. In *In Proceedings of the Fourth MPI Developers and Users Conference*.

[62] Brown, J. D., Woodward, S., Bass, B. M., and Johnson, C. L. (2011). IBM Power Edge of Network Processor: A Wire-Speed System on a Chip. *IEEE Micro*, 31(2):76–85.

[63] Brustoloni, J. C. and Steenkiste, P. (1996). Effects of Buffering Semantics on I/O Performance. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 277–291, Seattle, WA, USA.

[64] Buonadonna, P. and Culler, D. (2002). Queue Pair IP: A Hybrid Architecture for System Area Networks. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 247–256.

[65] Buonadonna, P., Geweke, A., and Culler, D. (1998). An Implementation and Analysis of the Virtual Interface Architecture. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–15, San Jose, CA, USA.

[66] Buzzard, G., Jacobson, D., Mackey, M., Marovich, S., and Wilkes, J. (1996). An Implementation of the Hamlyn Sender-managed Interface Architecture. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 245–259, Seattle, WA, USA.

[67] Callaghan, B., Lingutla-Raj, T., Chiu, A., Staubach, P., and Asad, O. (2003). NFS over RDMA. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, NICELI '03, pages 196–208, Karlsruhe, Germany.

[68] Caulfield, A. M., Coburn, J., Mollov, T., De, A., Akel, A., He, J., Jagatheesan, A., Gupta, R. K., Snavely, A., and Swanson, S. (2010a). Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11.

[69] Caulfield, A. M., De, A., Coburn, J., Mollow, T. I., Gupta, R. K., and Swanson, S. (2010b). Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395.

[70] Caulfield, A. M., Grupp, L. M., and Swanson, S. (2009). Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 217–228, Washington, DC, USA.

[71] Caulfield, A. M., Mollov, T. I., Eisner, L. A., De, A., Coburn, J., and Swanson, S. (2012). Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, London, England, UK.

[72] Caulfield, A. M. and Swanson, S. (2013). QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 464–474, Tel-Aviv, Israel.

[73] Chadalapaka, M., Hufferd, J., Satran, J., and Shah, H. (2007). Da: Datamover architecture for the internet small computer system interface (iscsi). RFC 5047, RFC Editor.

[74] Chan, M. and Cheriton, D. R. (2013). Improving server application performance via pure tcp ack receive optimization. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 359–364, San Jose, CA, USA.

[75] Chase, J. S., Gallatin, A. J., and Yocum, K. G. (2001). End System Optimizations for High-speed TCP. *IEEE Communications Magazine*, 39(4):68–74.

[76] Chelsio Communications. Terminator 4 (T4) 1G/10Gbe ASIC solutions. `http://www.chelsio.com/terminator-4-asic/`.

[77] Chelsio Communications. Terminator 5 (T5), 40 Gigabit Ethernet (40GbE) iWARP RDMA, iSCSI, TOE, FCoE, NIC Engine. `http://www.chelsio.com/nic/terminator-5-asic/`.

[78] Chen, S. (2009). FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 73–86, Providence, RI, USA.

[79] Chen, S. and Jin, Q. (2015). Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow.*, 8(7):786–797.

[80] Cheriton, D. (1988a). The V Distributed System. *Commun. ACM*, 31(3):314–333.

[81] Cheriton, D. R. (1988b). VMTP: Versatile Message Transaction Protocol.

[82] Cheriton, D. R., Slavenburg, G. A., and Boyle, P. D. (1986). Software-controlled Caches in the VMP Multiprocessor. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ISCA '86, pages 366–374, Tokyo, Japan.

[83] Chesson, G. (1988). Protocol Engine Design. In Partridge, C., editor, *Innovations in Internetworking*. Artech House, Inc, Norwood, MA, USA.

[84] Chu, H.-k. J. (1996). Zero-copy TCP in Solaris. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATC '96, pages 21–21, San Diego, CA, USA.

[85] Clark, D. D. (1985). The Structuring of Systems Using Upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 171–180, Orcas Island, WA, USA.

[86] Clark, D. D., Romkey, J., and Salwen, H. (1989). An Analysis of TCP Processing Overhead. *Comm. Mag.*, 27(6):23–29.

[87] Clark, D. D. and Tennenhouse, D. L. (1990). Architectural Considerations for a New Generation of Protocols. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, SIGCOMM '90, pages 200–208, Philadelphia, PA, USA.

[88] Coburn, J., Bunker, T., Schwarz, M., Gupta, R., and Swanson, S. (2013). From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, Farminton, PA, USA.

[89] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. (2011). NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, Newport Beach, CA, USA.

[90] Coburn, J. D. Providing Fast and Safe Access to Next-Generation of Non-Volatile Memories, Ph.D. thesis, University of California, San Diego, 2012. Available `http://cseweb.ucsd.edu/~swanson/papers/NVHeapsARIESThesis.pdf`.

[91] Compaq Corporation, Intel Corporation, and Microsoft Corporation (1997). Virtual Interface Architecture Specification, version 1.0, available at `http://www.cs.uml.edu/~bill/cs560/VI_spec.pdf`.

[92] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Coetzee, D. (2009). Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, Big Sky, MT, USA.

[93] Cooper, E., Menzilcioglu, O., and abd Frangois Bitzl, R. S. (1991). Host interface design for ATM LANs. In *Proceedings. of the 16th Conference on Local Computer Networks*, pages 247–258.

[94] Cooper, E. C., Steenkiste, P. A., Sansom, R. D., and Zill, B. D. (1990). Protocol Implementation on the Nectar Communication Processor. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols*, SIGCOMM '90, pages 135–144, Philadelphia, PA, USA.

[95] Culley, P., Elzur, U., Recio, R., Bailey, S., and Carrier, J. (2007). Marker PDU Aligned Framing for TCP Specification. RFC 5044, RFC Editor.

[96] Cully, B., Wires, J., Meyer, D., Jamieson, K., Fraser, K., Deegan, T., Stodden, D., Lefebvre, G., Ferstay, D., and Warfield, A. (2014). Strata: Scalable High-performance Storage on Virtualized Non-volatile Memory. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 17–31, Santa Clara, CA, USA.

[97] Daglis, A., Novaković, S., Bugnion, E., Falsafi, B., and Grot, B. (2015). Manycore Network Interfaces for In-memory Rack-scale Computing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 567–579, Portland, OR, USA.

[98] Dalton, C., Watson, G., Banks, D., Calamvokis, C., Edwards, A., and Lumley, J. (1993). Afterburner (network-independent card for protocols). *IEEE Network*, 7(4):36–43.

[99] Davie, B. S. (1991). A Host-network Interface Architecture for ATM. In *Proceedings of the Conference on Communications Architecture & Protocols*, SIGCOMM '91, pages 307–315, Zurich, Switzerland.

[100] Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 137–150, San Francisco, CA, USA.

[101] DeBergalis, M., Corbett, P., Kleiman, S., Lent, A., Noveck, D., Talpey, T., and Wittle, M. (2003). The direct access file system. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST'03, pages 175–188.

[102] Desnoyers, P. (2014). Analytic Models of SSD Write Performance. *Trans. Storage*, 10(2):8:1–8:25.

[103] Dirik, C. and Jacob, B. (2009). The Performance of PC Solid-state Disks (SSDs) As a Function of Band-

width, Concurrency, Device Architecture, and System Organization. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA ’09, pages 279–289, Austin, TX, USA.

[104] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. (2009). RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP ’09, pages 15–28, Big Sky, MT, USA.

[105] Doeringer, W. A., Dykeman, D., Kaiserswerth, M., Meister, B. W., Rudin, H., and Williamson, R. (1994). A Survey of Light-Weight Protocols for High-Speed Networks. In Tantawy, A. N., editor, *High Performance Networks*, volume 237 of *The Springer International Series in Engineering and Computer Science*, pages 3–28. Springer US.

[106] Douglis, F., Cáceres, R., Kaashoek, F., Li, K., Marsh, B., and Tauber, J. A. (1994). Storage Alternatives for Mobile Computers. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI ’94, pages 25–37, Monterey, CA, USA.

[107] Dragojević, A., Narayanan, D., Hodson, O., and Castro, M. (2014). FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI’14, pages 401–414, Seattle, WA, USA.

[108] Druschel, P., Abbott, M. B., Pagels, M. A., and Peterson, L. L. (1993). Network Subsystem Design. *IEEE Network*, 7(4):8–17.

[109] Druschel, P. and Banga, G. (1996). Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI ’96, pages 261–275, Seattle, WA, USA.

[110] Druschel, P. and Peterson, L. L. (1993). Fbufs: A High-bandwidth Cross-domain Transfer Facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP ’93, pages 189–202, Asheville, NC, USA.

[111] Druschel, P., Peterson, L. L., and Davie, B. S. (1994). Experiences with a High-speed Network Adaptor: A Software Perspective. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM ’94, pages 2–13, London, United Kingdom.

[112] Dubnicki, C., Iftode, L., Felten, E. W., and Li, K. (1996). Software support for virtual memory-mapped communication. In *Proceedings of the 10th International Parallel Processing Symposium*, IPPS ’96, pages 372–381.

[113] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. (2014). System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys ’14, pages 15:1–15:15, Amsterdam, The Netherlands.

[114] Edwards, A. and Muir, S. (1995). Experiences Implementing a High Performance TCP in User-space. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM ’95, pages 196–205, Cambridge, MA, USA.

[115] Edwards, A., Watson, G., Lumley, J., Banks, D., Calamvokis, C., and Dalton, C. (1994). User-space Protocols Deliver High Performance to Applications on a Low-cost Gb/s LAN. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM ’94, pages 14–23, London, United Kingdom.

[116] Eleftheriou, E., Haas, R., Jelitto, J., Lantz, M., and Pozidis, H. (2010). Trends in Storage Technologies. *IEEE Data Engineering Bulletin, 33(4), 4-13, IEEE.*

[117] Engler, D. R., Kaashoek, M. F., , and Jr., J. O. (1995). Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP ’95, pages 251–266, Copper Mountain, CO, USA.

[118] Feldmeier, C. C. (1990). Multiplexing Issues in Communication System Design. In *Proceedings of the ACM*

Symposium on Communications Architectures & Protocols, SIGCOMM '90, pages 209–219, Philadelphia, PA, USA.

[119] Fitch, Blake G. and Rayshubskiy, Alex and Ward, T.J. Chris and Pitman, Mike and Metzler, Bernard and Schick, Heiko J. and Krill, Benjamin and Morjan, Peter and Germain, Robert S. Blue Gene Active Storage, `http://institute.lanl.gov/hec-fsio/workshops/2010/presentations/day1/Fitch-HECFSIO-2010-BlueGeneActiveStorage.pdf`.

[120] Fitzgerald, R. and Rashid, R. F. (1986). The Integration of Virtual Memory Management and Interprocess Communication in Accent. *ACM Trans. Comput. Syst.*, 4(2):147–177.

[121] Fitzpatrick, B. (2004). Distributed Caching with Memcached. *Linux J.*, 2004(124):5–.

[122] Fiuczynski, M. E. and Bershad, B. N. (1996). An Extensible Protocol Architecture for Application-specific Networking. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 55–64, San Diego, CA, USA.

[123] Flajslik, M. and Rosenblum, M. (2013). Network Interface Design for Low Latency Request-response Protocols. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 333–346, San Jose, CA, USA.

[124] Foong, A. P., Huff, T. R., Hum, H. H., Patwardhan, J. P., and Regnier, G. J. (2003). TCP Performance Re-visited. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '03, pages 70–79.

[125] Forin, A., Golub, D., and Bershad, B. (1991). An I/O system for Mach 3.0. In *the USENIX Mach Symposium*, pages 163–176.

[126] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. (1997). Cluster-based Scalable Network Services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 78–91, Saint Malo, France.

[127] Fraser, A. (1989). The Universal Receiver Protocol. In *Proceedings of the 1st International Workshop on High-Speed Networks*, pages 19–25.

[128] Freimuth, D., Hu, E., LaVoie, J., Mraz, R., Nahum, E., Pradhan, P., and Tracey, J. (2005). Server Network Scalability and TCP Offload. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATC '05, pages 209–222, Anaheim, CA, USA.

[129] Frey, P. W. Zero-copy network communication. an applicability study of iWARP beyond micro benchmarks, PhD thesis, ETH Zurich, 2010. Available online `http://dx.doi.org/10.3929/ethz-a-006133695`.

[130] Frey, P. W. and Alonso, G. (2009). Minimizing the Hidden Cost of RDMA. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 553–560.

[131] Frey, P. W., Goncalves, R., Kersten, M., and Teubner, J. (2010). A spinning join that does not get dizzy. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, ICDCS '10, pages 283–292, Washington, DC, USA. IEEE Computer Society.

[132] Frey, P. W., Hasler, A., Metzler, B., and Alonso, G. (2009). Server-efficient High-definition Media Dissemination. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '09, pages 49–54, Williamsburg, VA, USA.

[133] Frey, P. W., Metzler, B., and Neeser, F. D. (2014). Zero copy data transmission in a software based RDMA network stack. US Patent 8,655,974.

[134] Fusion-IO. ioDrive Octal Data Sheet, at `http://www.fusionio.com/data-sheets/iodrive-octal-data-sheet/`.

[135] Fusion-IO. Software Development Kit (sdk) enables native flash mem-

ory access. `http : / / www . fusionio . com / press-releases / fusion-io-software-development-kit-enables-native-flash-memory-access`.

[136] Fusion-IO. Under the Hood of the ioMemory SDK, at `http://www.fusionio.com/blog/under-the-hood-of-the-iomemory-sdk/`.

[137] Fusion-IO. `http://www.fusionio.com/products/`.

[138] Fusion-IO. ioDrive2 and ioDrive2 Duo Multi Level Cell, Product Datasheet. `http://www.fusionio.com/load/-media-/2rezss/docsLibrary/FIO_DS_ioDrive2.pdf`.

[139] Gallatin, A., Chase, J., and Yocum, K. (1999). Trapeze/IP: TCP/IP at Near-gigabit Speeds. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATC '99, pages 109–120, Monterey, CA, USA.

[140] Ganger, G. R., Engler, D. R., Kaashoek, M. F., Briceño, H. M., Hunt, R., and Pinckney, T. (2002). Fast and Flexible Application-level Networking on Exokernel Systems. *ACM Trans. Comput. Syst.*, 20(1):49–83.

[141] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, Bolton Landing, NY, USA.

[142] Gibson, G. A., Nagle, D. F., Amiri, K., Butler, J., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. (1998). A Cost-effective, High-bandwidth Storage Architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 92–103, San Jose, CA, USA.

[143] Gibson, G. A., Nagle, D. F., Amiri, K., Chang, F. W., Feinberg, E. M., Gobioff, H., Lee, C., Ozceri, B., Riedel, E., Rochberg, D., and Zelenka, J. (1997). File Server Scaling with Network-attached Secure Disks. *SIGMETRICS Perform. Eval. Rev.*, 25(1):272–284.

[144] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012). PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Hollywood, CA, USA.

[145] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. (2014). GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Broomfield, CO, USA.

[146] Greaves, D. J., McAuley, D., French, L. J., and Hyden, E. (1994). Protocol and Interface for ATM LANs. *Journal of High Speed Networks*, 3(2):147–163.

[147] Gropp, W. (2001). Learning from the Success of MPI. In *Proceedings of the 8th International Conference on High Performance Computing*, HiPC '01, pages 81–94, London, UK, UK. Springer-Verlag.

[148] Grossman, L. (2005). Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In *Ottawa Linux Symposium*.

[149] Grun, P., Hefty, S., Sur, S., Goodell, D., Russell, R. D., Pritchard, H., and Squyres, J. M. (2015). A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency. In *23rd IEEE Annual Symposium on High-Performance Interconnects (HOTI)*, pages 34–39.

[150] Gunawi, H. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2004). Deploying Safe User-level Network Services with icTCP. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 317–332.

[151] Gurd, J. R., Kirkham, C. C., and Watson, I. (1985). The Manchester Prototype Dataflow Computer. *Commun. ACM*, 28(1):34–52.

[152] Han, S., Jang, K., Panda, A., Palkar, S., Han, D., and Ratnasamy, S. (2015). Softnic: A software nic to

augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of CA, Berkeley.

[153] Han, S., Marshall, S., Chun, B.-G., and Ratnasamy, S. (2012). MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 135–148, Hollywood, CA, USA.

[154] Hassani, A., Skjellum, A., Brightwell, R., and Barrett, B. W. (2013). Design, Implementation, and Performance Evaluation of MPI 3.0 on Portals 4.0. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 55–60, Madrid, Spain.

[155] Henry, D. S. and Joerg, C. F. (1992). A tightly-coupled processor-network interface. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 111–122.

[156] Hilland, J., Culley, P., Pinkerton, J., and Recio, R. (April, 2003). RDMA Protocol Verbs Specification (Version 1.0). `http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf`.

[157] Hoefler, T., Dinan, J., Thakur, R., Barrett, B., Balaji, P., Gropp, W., and Underwood, K. (2015a). Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.*, 2(2):9:1–9:26.

[158] Hoefler, T., Ross, R. B., and Roscoe, T. (2015b). Distributing the data plane for remote storage access. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland.

[159] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. (1988). Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81.

[160] Hu, X.-Y., Haas, R., and Eleftheriou, E. (2011). Container Marking: Combining Data Placement, Garbage Collection and Wear Levelling for Flash. In *Proceedings of the 19th IEEE Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 237–247.

[161] Huggahalli, R., Iyer, R., and Tetrick, S. (2005). Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, ISCA '05, pages 50–59.

[162] Hutchinson, N. C. and Peterson, L. L. (1988). Design of the X-kernel. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 65–75, Stanford, CA, USA.

[163] Hutchinson, N. C. and Peterson, L. L. (1991). The X-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76.

[164] Infiniband Trade Association. `http://www.infinibandta.org`.

[165] Infiniband Trade Association (2012). InfiniBand Architecture Volume 1 and Volume 2 `http://www.infinibandta.org/content/pages.php?pg=technology_public_specification`.

[166] Infiniband Trade Association. (2015). `http://www.infinibandta.org/content/pages.php?pg=about_us_RoCE`.

[167] Intel. 50 Years of Moore's Law, `http://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html`.

[168] Intel. Data Direct I/O Technology (Intel DDIO): A Primer, at `http://www.intel.co.uk/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf`.

[169] Intel. Intel Data Plane Development Kit (Intel DPDK), `http://www.intel.com/go/dpdk`.

[170] Intel. Intel Xeon Processor 7500 Series Uncore Programming Guide `http://www.intel.com/Assets/en_US/PDF/designguide/323535.pdf`.

[171] Intel. SSD 750 Series, `http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-750-series.html`.

[172] Intel. SSD Data Center S3610 Series, `http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-dc-s3610-series.html`.

[173] Intel weaves strategy to put interconnect fabrics on chip. `http://www.hpcwire.com/hpcwire/2012-09-10/intel_weaves_strategy_to_put_interconnect_fabrics_on_chip.html`, 2012.

[174] Ioannou, N., Koltsidas, I., Pletka, R., Tomic, S., Stoica, R., Weigold, T., and Eleftheriou, E. SALSA: treating the weaknesses of low-cost Flash in software. In *as a poster in 6th Annual Non-Volatile Memories Workshop 2015*.

[175] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. (2007). Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, Lisbon, Portugal.

[176] Islam, N. S., Rahman, M. W., Jose, J., Rajachandrasekar, R., Wang, H., Subramoni, H., Murthy, C., and Panda, D. K. (2012). High Performance RDMA-based Design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 35:1–35:35, Salt Lake City, UT, USA.

[177] Jacobson, V. (1990). Efficient Protocol Implementation. In *ACM SIGCOMM Tutorial*.

[178] Jeong, E. Y., Woo, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., and Park, K. (2014). mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 489–502, Seattle, WA, USA.

[179] Johnson, D. B. and Zwaenepoel, W. (1993). The Peregrine High-performance RPC System. *Software: Practice and Experience*, 23(2):201–221.

[180] Jose, J., Subramoni, H., Luo, M., Zhang, M., Huang, J., ur Rahman, M. W., Islam, N. S., Ouyang, X., Wang, H., Sur, S., and Panda, D. K. (2011). Memcached Design on High Performance RDMA Capable Interconnects. In *2011 International Conference on Parallel Processing (ICPP)*, pages 743–752.

[181] Josephson, W. K., Bongo, L. A., Flynn, D., and Li, K. (2010). DFS: A File System for Virtualized Flash Storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 85–100, San Jose, CA, USA.

[182] Kaashoek, M. F., Engler, D. R., Ganger, G. R., Briceño, H. M., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Jannotti, J., and Mackenzie, K. (1997). Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 52–65, Saint Malo, France.

[183] Kalia, A., Kaminsky, M., and Andersen, D. G. (2014). Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, Chicago, IL, USA.

[184] Kanakia, H. and Cheriton, D. (1988). The VMP Network Adapter Board (NAB): High-performance Network Communication for Multiprocessors. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 175–187, Stanford, CA, USA.

[185] Kanevsky, A., Bestler, C., Sharp, R., and Wise, S. (2012). Enhanced Remote Direct Memory Access (RDMA) Connection Establishment. RFC 6581, RFC Editor.

[186] Kapoor, R., Porter, G., Tewari, M., Voelker, G. M., and Vahdat, A. (2012). Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 9:1–9:14, San Jose, CA, USA.

[187] Kaufmann, A., Peter, S., Anderson, T., and Krishnamurthy, A. (2015). Flexnic: Rethinking network dma.

In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland.

[188] Kawaguchi, A., Nishioka, S., and Motoda, H. (1995). A Flash-memory Based File System. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 13–13, New Orleans, LA, USA.

[189] Kay, J. and Pasquale, J. (1993). The Importance of Non-data Touching Processing Overheads in TCP/IP. In *Conference Proceedings on Communications Architectures, Protocols and Applications*, SIGCOMM '93, pages 259–268, San Francisco, CA, USA.

[190] Kay, J. and Pasquale, J. (1996). Profiling and reducing processing overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, 4:817–828.

[191] Kgil, T. and Mudge, T. (2006). FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 103–112, Seoul, Korea.

[192] Kim, S., Huh, S., Hu, Y., Zhang, X., Witchel, E., Wated, A., and Silberstein, M. (2014). GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 201–216, Broomfield, CO, USA.

[193] Kleinpaste, K., Steenkiste, P., and Zill, B. (1995). Software Support for Outboard Buffering and Checksumming. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '95, pages 87–98, Cambridge, MA, USA.

[194] Knowlton, K. C. (1965). A fast storage allocator. *Commun. ACM*, 8(10):623–624.

[195] Ko, M. and Black, D. (2012). IANA Registries for the Remote Direct Data Placement (RDDP) Protocols. RFC 6580, RFC Editor.

[196] Kohler, E., Kaashoek, M. F., and Montgomery, D. R. (1999). A Readable TCP in the Prolac Protocol Language. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 3–13, Cambridge, Massachusetts, USA.

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

[198] Labrinidis, A. and Jagadish, H. V. (2012). Challenges and Opportunities with Big Data. *Proc. VLDB Endow.*, 5(12):2032–2033.

[199] Landsman, D. and Walker, D. AHCI and NVMe as interfaces for SATA Express Devices), `https://www.sata-io.org/sites/default/files/images/NVMe_and_AHCI_as_SATA_Express_Interface_Options_final.pdf`.

[200] Lee, B. C., Ipek, E., Mutlu, O., and Burger, D. (2009). Architecting Phase Change Memory As a Scalable Dram Alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, Austin, TX, USA.

[201] Lee, C., Sim, D., Hwang, J.-Y., and Cho, S. (2015). F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 273–286, Santa Clara, CA, USA.

[202] Lee, S.-W., Moon, B., Park, C., Kim, J.-M., and Kim, S.-W. (2008). A Case for Flash Memory Ssd in Enterprise Database Applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1075–1086, Vancouver, Canada.

[203] Leiserson, C. E., Abuhamdeh, Z. S., Douglas, D. C., Feynman, C. R., Ganmukhi, M. N., Hill, J. V., Hillis, D., Kuszmaul, B. C., St. Pierre, M. A., Wells, D. S., Wong, M. C., Yang, S.-W., and Zak, R. (1992). The Network Architecture of the Connection Machine CM-5 (Extended Abstract). In *Proceedings of the Fourth*

Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '92, pages 272–285, San Diego, California, USA.

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

[205] Li, S., Lim, H., Lee, V. W., Ahn, J. H., Kalia, A., Kaminsky, M., Andersen, D. G., Seongil, O., Lee, S., and Dubey, P. (2015). Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 476–488, Portland, OR, USA.

[206] Liao, G., Znu, X., and Bnuyan, L. (2011). A New Server I/O Architecture for High Speed Networks. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 255–265.

[207] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. (2014). MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Seattle, WA.

[208] Lim, K., Meisner, D., Saidi, A. G., Ranganathan, P., and Wenisch, T. F. (2013). Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 36–47, Tel-Aviv, Israel.

[209] Liss, L. (2013). On Demand Paging for User-level Networking, at `http://downloads.openfabrics.org/downloads/Media/Monterey_2013/2013_Workshop_Tues_0930_liss_odp.pdf`.

[210] Liu, J., Chandrasekaran, B., Wu, J., Jiang, W., Kini, S., Yu, W., Buntinas, D., Wyckoff, P., and Panda, D. K. (2003). Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, Phoenix, AZ, USA.

[211] Liu, N., Cope, J., Carns, P., Carothers, C., Ross, R., Grider, G., Crume, A., and Maltzahn, C. (2012). On the role of burst buffers in leadership-class storage systems. In *In Proceedings of the 2012 IEEE Conference on Massive Data Storage*.

[212] Liu, R.-S., Shen, D.-Y., Yang, C.-L., Yu, S.-C., and Wang, C.-Y. M. (2014). NVM Duet: Unified Working Memory and Persistent Store Architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 455–470, Salt Lake City, UT, USA.

[213] Lowell, D. E. and Chen, P. M. (1997). Free Transactions with Rio Vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 92–101, Saint Malo, France.

[214] Luo, M., Seager, K., Murthy, K. S., Archer, C. J., Sur, S., and Hefty, S. (2014). Early evaluation of scalable fabric interface for pgas programming models. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 1:1–1:13, New York, NY, USA. ACM.

[215] Lutz, C. High-Performance, In-Memory Graph Processing with RDMA, M.Sc. thesis, ETH Zurich, October 2014.

[216] Maeda, C. and Bershad, B. N. (1992). Networking performance for microkernels. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 154–159.

[217] Maeda, C. and Bershad, B. N. (1993). Protocol Service Decomposition for High-performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 244–255, Asheville, NC, USA.

[218] Magoutis, K. (2004). The Case Against User-level Networking. In *In Third Workshop on Novel Uses of System Area Networks (SAN-3) (Held in conjunction with HPCA-10.*

[219] Magoutis, K., Addetia, S., Fedorova, A., and Seltzer, M. I. (2003). Making the Most Out of Direct-Access Network Attached Storage. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 189–202, San Francisco, CA, USA.

[220] Magoutis, K., Addetia, S., Fedorova, A., Seltzer, M. I., Chase, J. S., Gallatin, A. J., Kisley, R., Wickremesinghe, R., and Gabber, E. (2002). Structure and Performance of the Direct Access File System. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATC '02, pages 1–14.

[221] Makineni, S. and Iyer, R. (2004). Architectural Characterization of TCP/IP Packet Processing on the Pentium M Microprocessor. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pages 152–161.

[222] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, Indianapolis, IN, USA.

[223] Mangstor. NX-Series NVMe over Fabric Flash Storage Arrays, at `https://www.mangstor.com/page/nx-series-flash-storage-arrays`.

[224] Marinos, I., Watson, R. N., and Handley, M. (2014). Network Stack Specialization for Performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 175–186, Chicago, IL, USA.

[225] Markatos, E. P. (2002). Speeding up TCP/IP: faster processors are not enough. In *Proceedings of the 21st IEEE International on Performance, Computing, and Communications Conference*, pages 341–345.

[226] Markus Levy. Interfacing Microsoft's Flash File System In Memory Products, Intel Corp., 1993.

[227] Martin, R. P., Vahdat, A. M., Culler, D. E., and Anderson, T. E. (1997). Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 85–97, Denver, CO, USA.

[228] McKenney, P. E. and Dove, K. F. (1992). Efficient Demultiplexing of Incoming TCP Packets. In *Conference Proceedings on Communications Architectures &Amp; Protocols*, SIGCOMM '92, pages 269–279.

[229] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, pages 69–74.

[230] Mehlan, T., Rehm, W., Engler, R., and Wenzel, T. (2004). Providing a High-Performance VIA-Module for LAM/MPI. In *Proceedings of the 2004 International Conference on Parallel Computing in Electrical Engineering*, PARELEC 2004, pages 277–282.

[231] Mellanox Technologies. Accelio-based network block device (NBDX), at `https://github.com/accelio/NBDX`.

[232] Mellanox Technologies. Introduction to InfiniBand, White Paper at at `http://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf`.

[233] Mellanox Technologies. RDMA Aware Networks Programming User Manual, version 1.7. `http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf`.

[234] Mellanox Technologies. RoCE vs. iWARP Competitive Analysis. `http://www.mellanox.com/pdf/whitepapers/WP_RoCE_vs_iWARP.pdf`.

[235] Menon, A. and Zwaenepoel, W. (2008). Optimizing TCP Receive Performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 85–98, Boston, MA, USA.

[236] Metzler, B. Software iWARP kernel driver and user library for Linux. `https://github.com/zrlio/softiwarp`.

[237] Microsoft Corporation. Microsoft SMB Protocol and CIFS Protocol Overview, at `https://msdn.microsoft.com/en-us/library/windows/desktop/aa365233(v=vs.85).aspx`, 2015.

[238] Microsoft Corporation. SMB2 Remote Direct Memory Access (RDMA) Transport Protocol, at `https://msdn.microsoft.com/en-us/library/hh536346.aspx`, 2015.

[239] Minturn, D., Regnier, G., Krueger, J., Iyer, R., and Makineni, S. (2003). Addressing TCP/IP processing challenges using the IA and IXP processors. *Intel Technology Journal, Volume 7, Issue 4*.

[240] Minturn, Dave. NVM Express Over Fabrics, at `http://downloads.openfabrics.org/downloads/Media/Monterey_2015/Monday/monday_10.pdf` in OpenFabrics Developers Workshop, 2015.

[241] Mitchell, C., Geng, Y., and Li, J. (2013). Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, San Jose, CA, USA.

[242] Mogul, J., Rashid, R., and Accetta, M. (1987). The Packer Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 39–51, Austin, Texas, USA.

[243] Mogul, J. C. (1991). Network Locality at the Scale of Processes. In *Proceedings of the Conference on Communications Architecture &Amp; Protocols*, SIGCOMM '91, pages 273–284.

[244] Mogul, J. C. (2003). TCP Offload is a Dumb Idea Whose Time Has Come. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 5–5, Lihue, Hawaii.

[245] Mogul, J. C., Argollo, E., Shah, M., and Faraboschi, P. (2009). Operating System Support for NVM+DRAM Hybrid Main Memory. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, pages 14–14, Monte Verita, Switzerland.

[246] Mogul, J. C., Baumann, A., Roscoe, T., and Soares, L. (2011). Mind the Gap: Reconnecting Architecture and OS Research. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 1–1, Napa, CA, USA.

[247] Mogul, J. C. and Borg, A. (1991). The Effect of Context Switches on Cache Performance. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 75–84, Santa Clara, CA, USA.

[248] Mogul, J. C. and Ramakrishnan, K. K. (1996). Eliminating Receive Livelock in an Interrupt-driven Kernel. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATC '96, pages 99–112, San Diego, CA, USA.

[249] Mosberger, D. and Peterson, L. L. (1996). Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 153–167, Seattle, WA, USA.

[250] MPI Forum. Message Passing Interface (MPI) Forum Home Page, `http://www.mpi-forum.org/`.

[251] Mukherjee, S. S., Falsafi, B., Hill, M. D., and Wood, D. A. (1996). Coherent Network Interfaces for Fine-grain Communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, pages 247–258.

[252] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. (2013). Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, Farminton, PA, USA.

[253] Nahum, E., Yates, D., Kurose, J., and Towsley, D. (1997). Cache Behavior of Network Protocols. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '97, pages 169–180, Seattle, WA, USA.

[254] Narayanan, D. and Hodson, O. (2012). Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 401–410, London, England, UK.

[255] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S., and Oskin, M. (2015). Latency-tolerant Software Distributed Shared Memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 291–305, Santa Clara, CA, USA.

[256] NetApp. Network Attached Storage (NAS), at `http://www.netapp.com/us/products/protocols/nas/`.

[257] Nightingale, E. B., Elson, J., Fan, J., Hofmann, O., Howell, J., and Suzue, Y. (2012). Flat Datacenter Storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 1–15, Hollywood, CA, USA.

[258] Noakes, M. D., Wallach, D. A., and Dally, W. J. (1993). The J-machine Multicomputer: An Architectural Evaluation. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 224–235, San Diego, CA, USA.

[259] Novakovic, S., Daglis, A., Bugnion, E., Falsafi, B., and Grot, B. (2014). Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 3–18, Salt Lake City, UT, USA.

[260] Nowicki, B. (1989). Nfs: Network file system protocol specification. RFC 1094, RFC Editor.

[261] Nvidia. RDMA for GPUDirect, CUDA Toolkit Documentation. `http://docs.nvidia.com/cuda/gpudirect-rdma/index.html`.

[262] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., and Rosenblum, M. (2011). Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, Cascais, Portugal.

[263] Ousterhout, J. (1990). Why arent operating systems getting faster as fast as hardware. In *In Summer USENIX 90*, pages 247–256.

[264] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R. (2010). The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105.

[265] Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., Rumble, S., Stutsman, R., and Yang, S. (2015a). The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55.

[266] Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., and Chun, B.-G. (2015b). Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Oakland, CA, USA.

[267] Ouyang, X., Nellans, D., Wipfel, R., Flynn, D., and Panda, D. K. (2011). Beyond Block I/O: Rethinking Traditional Storage Primitives. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 301–311.

[268] Pagels, M., Druschel, P., and Peterson, L. L. (1993). Cache and TLB Effectiveness in the Processing of Network Data. Technical Report 9408, Department of Computer Science, The University of Arizona, Tuscon, AZ, USA.

[269] Pai, V. S., Druschel, P., and Zwaenepoel, W. (1999). IO-lite: A Unified I/O Buffering and Caching System. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 15–28, New Orleans, LA, USA.

[270] Park, S., Kelly, T., and Shen, K. (2013). Failure-atomic Msync(): A Simple and Efficient Mechanism for

Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 225–238, Prague, Czech Republic.

[271] Pearson, R. S. (2015). Burst Buffers, `http://downloads.openfabrics.org/downloads/Media/Monterey_2015/Monday/monday_14.pdf`, at OFA Developer Workshop.

[272] Pesterev, A., Strauss, J., Zeldovich, N., and Morris, R. T. (2012). Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 337–350, Bern, Switzerland.

[273] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Anderson, T., Krishnamurthy, A., Zbikowski, M., and Woos, D. (2014a). Towards High-Performance Application-Level Storage Management. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, USA.

[274] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., and Roscoe, T. (2014b). Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, Broomfield, CO.

[275] Peterson, L., Hutchinson, N., O'Malley, S., and Abbott, M. (1989). RPC in the x-Kernel: Evaluating New Design Techniques. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 91–101.

[276] Pierce, P. and Regnier, G. (1994). The Paragon implementation of the NX message passing interface. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 184–190.

[277] Pinkerton, J. and Deleganes, E. (2007). Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMAP) Security. RFC 5042, RFC Editor.

[278] Poke, M. and Hoefler, T. (2015). DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 107–118, Portland, Oregon, USA.

[279] Prabhakaran, V., Rodeheffer, T. L., and Zhou, L. (2008). Transactional Flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 147–160, San Diego, CA, USA.

[280] Pratt, I. and Fraser, K. (2001). Arsenic: a user-accessible gigabit ethernet interface. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1 of *INFOCOMM '01*, pages 67–76.

[281] Qureshi, M. K., Srinivasan, V., and Rivers, J. A. (2009). Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, Austin, TX, USA.

[282] Ramakrishnan, K. (1993). Performance considerations in designing network interfaces. *IEEE Journal on Selected Areas in Communications*, 11(2):203–219.

[283] Recio, R., Metzler, B., Culley, P., Hilland, J., and Garcia, D. (2007). A Remote Direct Memory Access Protocol Specification. RFC 5040, RFC Editor.

[284] Recio, R. J. (2003). Server I/O Networks Past, Present, and Future. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, NICELI '03, pages 163–178, Karlsruhe, Germany.

[285] Regnier, G., Makineni, S., Illikkal, R., Iyer, R., Minturn, D., Huggahalli, R., Newell, D., Cline, L., and Foong, A. (2004). TCP Onloading for Data Center Servers. *IEEE Computer*, 37(11):48–58.

[286] Reinhardt, S. K., Larus, J. R., and Wood, D. A. (1994). Tempest and Typhoon: User-level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pages 325–336, Chicago, IL, USA.

[287] Riesen, R., Brightwell, R., Pedretti, K., Barrett, B., Underwood, K., Maccabe, A. B., and Hudson, T. (2008). The Portals 4.0 Message Passing Interface. Technical Report SAND2008-2639, Sandia National Laboratories.

[288] Ritchie, D. M. and Thompson, K. (1974). The UNIX Time-sharing System. *Commun. ACM*, 17(7):365–375.

[289] Rizzo, L. (2012). Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 101–112, Boston, MA, USA.

[290] Rodrigues, S. H., Anderson, T. E., and Culler, D. E. (1997). High-performance Local Area Communication with Fast Sockets. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATC '97, Anaheim, CA, USA.

[291] Rosenblum, M. and Ousterhout, J. K. (1992). The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52.

[292] Rumble, S. M., Ongaro, D., Stutsman, R., Rosenblum, M., and Ousterhout, J. K. (2011). It's Time for Low Latency. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 11–11, Napa, CA, USA.

[293] Samsung. The 850 PRO SSD, `http://www.samsung.com/us/computer/memory-storage/MZ-7KE2T0BW`.

[294] Samsung. XS1715 enterprise drive, `http://www.samsung.com/global/business/semiconductor/file/product/XS1715_ProdOverview_2014_1.pdf`.

[295] Sandberg, R. (1986). The Sun Network File System: Design, Implementation and Experience. Technical report, in Proceedings of the Summer 1985 USENIX Technical Conference and Exhibition.

[296] Satyanarayanan, M., Howard, J. H., Nichols, D. A., Sidebotham, R. N., Spector, A. Z., and West, M. J. (1985). The ITC Distributed File System: Principles and Design. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, SOSP '85, pages 35–50, Orcas Island, WA, USA.

[297] Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., and Kistler, J. J. (1993). Lightweight Recoverable Virtual Memory. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 146–160, Asheville, NC, USA.

[298] Saxena, M., Shah, M. A., Harizopoulos, S., Swift, M. M., and Merchant, A. (2012a). Hathi: Durable Transactions for Memory Using Flash. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 33–38, Scottsdale, AZ, USA.

[299] Saxena, M. and Swift, M. M. (2009). FlashVM: Revisiting the Virtual Memory Hierarchy. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, pages 13–13, Monte Verita, Switzerland.

[300] Saxena, M. and Swift, M. M. (2010). FlashVM: Virtual Memory Management on Flash. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ATC'10, pages 14–14, Boston, MA, USA.

[301] Saxena, M., Swift, M. M., and Zhang, Y. (2012b). FlashTier: A Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, Bern, Switzerland.

[302] Scaling the Facebook data warehouse to 300 PB. `https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/`.

[303] Schroeder, M. D. and Burrows, M. (1989). Performance of Firefly RPC. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 83–90.

[304] Schrmann, F., Delalondre, F., Kumbhar, P., Biddiscombe, J., Gila, M., Tacchella, D., Curioni, A., Metzler, B., Morjan, P., Fenkes, J., Franceschini, M., Germain, R., Schneidenbach, L., Ward, T., and Fitch, B. (2014). Rebasing i/o for scientific computing: Leveraging storage class memory in an ibm bluegene/q supercomputer. In Kunkel, J., Ludwig, T., and Meuer, H., editors, *Supercomputing*, volume 8488 of *Lecture Notes in Computer*

Science, pages 331–347. Springer International Publishing.

[305] Seagate. The 1200 SSD, `http://www.seagate.com/internal-hard-drives/solid-state-hybrid/1200-ssd/`.

[306] Seppanen, E., O'Keefe, M., and Lilja, D. (2010). High performance solid state storage under Linux. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12.

[307] Shah, H., Marti, F., Noureddine, W., Eiriksson, A., and Sharp, R. (2014). Remote Direct Memory Access (RDMA) Protocol Extensions. RFC 7306, RFC Editor.

[308] Shah, H., Pinkerton, J., Recio, R., and Culley, P. (2007). Direct Data Placement over Reliable Transports. RFC 5041, RFC Editor.

[309] Shah, H. V., Pu, C., and Madukkarumukumana, R. S. (1999). High Performance Sockets and RPC over Virtual Interface (VI) Architecture. In Sivasubramaniam, A. and Lauria, M., editors, *Network-Based Parallel Computing. Communication, Architecture, and Applications*, volume 1602 of *Lecture Notes in Computer Science*, pages 91–107. Springer Berlin Heidelberg.

[310] Shalev, L., Satran, J., Borovik, E., and Ben-Yehuda, M. (2010). IsoStack: Highly Efficient Network Processing on Dedicated Cores. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ATC'10, pages 61–74, Boston, MA, USA.

[311] Shamis, P., Venkata, M. G., Lopez, M., Baker, M. B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R. L., Liss, L., Shahar, Y., Potluri, S., Rossetti, D., Becker, D., Poole, D., Lamb, C., Kumar, S., Stunkel, C., Bosilca, G., and Bouteiller, A. (2015). UCX: An Open Source Framework for HPC Network APIs and Beyond. In *Proceedings of the 23rd IEEE Annual Symposium on High-Performance Interconnects (HOTI),*, pages 40–43.

[312] Shao, B., Wang, H., and Li, Y. (2013). Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, New York, USA.

[313] Shin, D. I., Yu, Y. J., Kim, H. S., Choi, J. W., Jung, D. Y., and Yeom, H. Y. (2013). Dynamic Interval Polling and Pipelined Post I/O Processing for Low-latency Storage Class Memory. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'13, pages 5–5, San Jose, CA, USA.

[314] Shin, W., Chen, Q., Oh, M., Eom, H., and Yeom, H. Y. (2014). OS I/O Path Optimizations for Flash Solid-state Drives. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 483–488, Philadelphia, PA, USA.

[315] Shinde, P., Kaufmann, A., Kourtis, K., and Roscoe, T. (2013a). Modeling NICs with Unicorn. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, PLOS '13, pages 3:1–3:6, Farmington, PA, USA.

[316] Shinde, P., Kaufmann, A., Roscoe, T., and Kaestle, S. (2013b). We Need to Talk About NICs. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 1–1, Santa Ana Pueblo, NM, USA.

[317] Shivam, P. and Chase, J. S. (2003a). On the Elusive Benefits of Protocol Offload. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, NICELI '03, pages 179–184, Karlsruhe, Germany.

[318] Shivam, P. and Chase, J. S. (2003b). On the Elusive Benefits of Protocol Offload. In *Proceedings of the ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, NICELI '03, pages 179–184, Karlsruhe, Germany.

[319] Smith, J. M. and Traw, C. B. S. (1993). Giving Applications Access to Gb/s Networking. *IEEE Network*, 7(4):44–52.

[320] Soares, L. and Stumm, M. (2010). FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Vancouver, BC, Canada.

[321] Soares, L. and Stumm, M. (2011). Exception-less System Calls for Event-driven Servers. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ATC'11, pages 131–144, Portland, OR, USA.

[322] Steenkiste, P., Zill, B., Kung, H. T., Schlick, S., Hughes, J., Kowalski, B., and Mullaney, J. (1993). A Host Interface Architecture for High-Speed Networks. In *Proceedings of the IFIP TC6/WG6.4 Fourth International Conference on High Performance Networking IV*, pages 31–46, Amsterdam, The Netherlands, The Netherlands. North-Holland Publishing Co.

[323] Steenkiste, P. A. (1994). A Systematic Approach to Host Interface Design for High-speed Networks. *IEEE Computer*, 27(3):47–57.

[324] Sterling, T., Becker, D. J., Savarese, D., Dorband, J. E., Ranawake, U. A., and Packer, C. V. (1995). Beowulf: A Parallel Workstation For Scientific Computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14.

[325] Stuedi, P., Metzler, B., and Trivedi, A. (2013). jVerbs: Ultra-low Latency for Data Center Applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 10:1–10:14, Santa Clara, CA, USA.

[326] Stuedi, P., Trivedi, A., and Metzler, B. (2012). Wimpy Nodes with 10GbE: Leveraging One-sided Operations in soft-RDMA to Boost Memcached. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 347–353, Boston, MA, USA.

[327] Stuedi, P., Trivedi, A., Metzler, B., and Pfefferle, J. (2014). DaRPC: Data Center RPC. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 15:1–15:13, Seattle, WA, USA.

[328] Suzuki, K. and Swanson, S. (2015). The non-volatile memory technology database (nvmdb). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego. http://nvmdb.ucsd.edu.

[329] Swanson, L. A. E. T. M. S. (2013). Quill: Exploiting fast non-volatile memory by transparently bypassing the file system. Technical Report CS2013-0991, Computer Science and Engineering University of California, San Diego.

[330] Swanson, S. and Caulfield, A. (2013). Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage. *IEEE Computer*, 46(8):52–59.

[331] Tennenhouse, D. L. (1989). Layered Multiplexing Considered Harmful. In *In First International Workshop on High Speed Networking*.

[332] The Data Center Bridging (DCB) Task Group (TG). `http://www.ieee802.org/1/pages/dcbridges.html`.

[333] Thekkath, C. A. and Levy, H. M. (1993). Limits to Low-latency Communication on High-speed Networks. *ACM Trans. Comput. Syst.*, 11(2):179–203.

[334] Thekkath, C. A., Levy, H. M., and Lazowska, E. D. (1994). Separating Data and Control Transfer in Distributed Operating Systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 2–11, San Jose, CA, USA.

[335] Thekkath, C. A., Nguyen, T. D., Moy, E., and Lazowska, E. D. (1993). Implementing Network Protocols at User Level. In *Conference Proceedings on Communications Architectures, Protocols and Applications*, SIGCOMM '93, pages 64–73, San Francisco, CA, USA.

[336] Thereska, E., Ballani, H., O'Shea, G., Karagiannis, T., Rowstron, A., Talpey, T., Black, R., and Zhu,

T. (2013). IOFlow: A Software-defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, Farminton, PA, USA.

[337] Traw, C. and Smith, J. (1993). Hardware/software organization of a high-performance ATM host interface. *IEEE Journal on Selected Areas in Communications*, 11(2):240–253.

[338] Trivedi, A. "(R)DMA in userspace" on Linux RDMA mailing list, `http://comments.gmane.org/gmane.linux.drivers.rdma/13635`, October, 2012.

[339] Trivedi, A., Metzler, B., and Stuedi, P. (2011). A Case for RDMA in Clouds: Turning Supercomputer Networking into Commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 17:1–17:5, Shanghai, China.

[340] Trivedi, A., Stuedi, P., Metzler, B., Lutz, C., Schmatz, M., and Gross, T. R. (2015). RStore: A Direct-Access DRAM-based Data Store. In *IEEE 35th International Conference on Distributed Computing Systems (ICDCS)*, pages 674–685, Columbus, OH, USA.

[341] Trivedi, A., Stuedi, P., Metzler, B., Pletka, R., Fitch, B. G., and Gross, T. R. (2013). Unified High-performance I/O: One Stack to Rule Them All. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 4–4, Santa Ana Pueblo, NM, USA.

[342] Tschudin, C. (1991). Flexible Protocol Stacks. In *Proceedings of the Conference on Communications Architecture &Amp; Protocols*, SIGCOMM '91, pages 197–205, Zurich, Switzerland.

[343] Tzou, S.-Y. and Anderson, D. P. (1991). The Performance of Message-passing Using Restricted Virtual Memory Remapping. *Software - Practice and Experience*, 21(3):251–267.

[344] Vasudevan, V., Andersen, D. G., and Kaminsky, M. (2011). The Case for VOS: The Vector Operating System. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 31–31, Napa, CA, USA.

[345] Vasudevan, V., Kaminsky, M., and Andersen, D. G. (2012). Using Vector Interfaces to Deliver Millions of IOPS from a Networked Key-value Storage Server. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 8:1–8:13, San Jose, CA, USA.

[346] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. (2013). Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, Santa Clara, CA, USA.

[347] Venkataraman, S., Tolia, N., Ranganathan, P., and Campbell, R. H. (2011). Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 61–76, San Jose, CA, USA.

[348] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 18:1–18:17, Bordeaux, France.

[349] Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P., and Swift, M. M. (2014). Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, Amsterdam, The Netherlands.

[350] Volos, H., Panneerselvam, S., Nalli, S., and Swift, M. M. (2013). Storage-class Memory Needs Flexible Interfaces. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, APSys '13, pages 11:1–11:7, Singapore, Singapore.

[351] Volos, H., Tack, A. J., and Swift, M. M. (2011). Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, Newport Beach, CA, USA.

[352] von Eicken, T., Basu, A., Buch, V., and Vogels, W. (1995). U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 40–53, Copper Mountain, CO, USA.

[353] von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E. (1992). Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, Queensland, Australia.

[354] Watson, R. W. and Mamrak, S. A. (1987). Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices. *ACM Trans. Comput. Syst.*, 5(2):97–120.

[355] Wei, M., Bjørling, M., Bonnet, P., and Swanson, S. (2014). I/O Speculation for the Microsecond Era. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 475–482, Philadelphia, PA, USA.

[356] Weiss, Z., Subramanian, S., Sundararaman, S., Talagala, N., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2015). ANViL: Advanced Virtualization for Modern Non-volatile Memory Devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 111–118, Santa Clara, CA, USA.

[357] Welsh, M., Basu, A., and von Eicken, T. (1997). Incorporating memory management into user-level network interfaces. Technical report, Ithaca, NY, USA.

[358] Wilkes, J. (1992). Hamlyn - an Interface for sender-based communications. Technical Report HPL-OSR-92-13, Hewlett-Packard Laboratories.

[359] Willmann, P., Kim, H.-y., Rixner, S., and Pai, V. S. (2005). An efficient programmable 10 gigabit ethernet network interface card. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 96–107.

[360] Willmann, P., Rixner, S., and Cox, A. L. (2006). An Evaluation of Network Stack Parallelization Strategies in Modern Operating Systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATC '06, pages 91–96, Boston, MA, USA.

[361] Woodhouse, D. (2001). JFFS: The journaling flash file system. In *Proceedings of the Ottawa Linux Symposium.*

[362] Wu, M. and Zwaenepoel, W. (1994). eNVy: A Non-volatile, Main Memory Storage System. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 86–97, San Jose, CA, USA.

[363] Wu, X. and Reddy, A. L. N. (2011). SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, Seattle, WA, USA.

[364] Yang, J., Minturn, D. B., and Hady, F. (2012). When Poll is Better Than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 25–32, San Jose, CA, USA.

[365] Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K. L., and He, B. (2015). NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181, Santa Clara, CA, USA.

[366] Yu, Y. J., Shin, D. I., Shin, W., Song, N. Y., Choi, J. W., Kim, H. S., Eom, H., and Yeom, H. Y. (2014). Optimizing the Block I/O Subsystem for Fast Storage Devices. *ACM Trans. Comput. Syst.*, 32(2):6:1–6:48.

[367] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 15–28, San Jose, CA, USA.

[368] Zhang, Y., Arulraj, L. P., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2012). De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 1–16, San Jose, CA, USA.

[369] Zhang, Y., Yang, J., Memaripour, A., and Swanson, S. (2015). Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, Istanbul, Turkey.

[370] Zhao, J., Mutlu, O., and Xie, Y. (2014). FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 153–165, Cambridge, United Kingdom.

List of Figures

List of Tables

Curriculum Vitae

Personal Data

Full Name:	Animesh Kumar Trivedi
Date of birth:	August 02 1986, Lucknow, India
Citizenship:	The Republic of India

Education

2011–2015	Ph.D studies, Department of Computer Science, ETH Zurich. Pre-Doc researcher, IBM Research Lab, Zurich.
2007–2009	Master of Science (M.Sc.) ETH in Computer Science, Department of Computer Science, ETH Zurich.
2003–2007	Bachelor of Technology (B.Tech.) in Information Technology, Indian Institute of Information Technology, Allahabad (IIITA), India.
1999-2003	Secondary Education (High-School), CBSE, Rani Laxmi Bai Memorial School, Lucknow, India.

Conference Publications

– Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Clemens Lutz, Martin Schmatz, Thomas R. Gross. RStore: A Direct-Access DRAM-based Data Store. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*, ICDCS'15, pages 674–685, Columbus, OH, USA, July 2015.

– Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ioannis Koltsidas, Thomas R. Gross. A Hybrid I/O Virtualization Framework for RDMA-capable Network Interfaces. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environment*, (VEE'15), pages 17–30, Istanbul, Turkey, March 2015.

- Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Jonas Pfefferle. DaRPC: Data Center RPC. In *Proceedings of the 5th ACM Symposium on Cloud Computing 2014* (SoCC'14), pages 15:1–15:13, Seattle, WA, USA, November 2014.
- Animesh Trivedi, Bernard Metzler, Patrick Stuedi, and Thomas R. Gross. On Limitations of Network Acceleration. In *Proceedings of the Ninth ACM Conference on Emerging Networking EXperiments and Technologies* (CoNEXT '13), pages 121–126, Santa Barbara, CA, USA, December 2013.
- Patrick Stuedi, Bernard Metzler, Animesh Trivedi. jVerbs: Ultra-low Latency for Data Center Applications. In Proceedings of the 4th ACM Symposium on Cloud Computing 2013 (SoCC'13), pages 10:1–10:14, Santa Clara, CA, USA, October 2013.
- Animesh Trivedi, Patrick Stuedi, Bernard Metzler, Roman Pletka, Blake G. Fitch, Thomas R. Gross. Unied High-Performance I/O: One Stack to Rule Them All. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems* (HotOS XIV), Santa Ana Pueblo, NM, USA, May 2013.
- Patrick Stuedi, Animesh Trivedi, Bernard Metzler. Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft RDMA to Boost Memcached. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (ATC), pages 347–353, Boston, MA, USA, June 2012.
- Animesh Trivedi, Bernard Metzler, and Patrick Stuedi. A Case for RDMA in Clouds: Turning Supercomputer Networking into Commodity. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys 11, pages 17:1–17:5, Shanghai, China, July 2011.

Patents

- US 8909727 B2 - RDMA read destination buffers mapped onto a single representation.
- US 20130326122 A1 - Distributed memory access in a network.
- US 20140214997 A1 - Method and device for data transmissions using RDMA.
- US 20140359146 A1 - Remote procedure call with call-by-reference semantics using Remote Direct Memory Access.
- US 20150113088 A1 - Persistent caching for operating a persistent caching system.