Stony Brook University

Scalable End-to-End Data I/O over Enterprise and Data-Center Networks

A Dissertation presented

by

Yufei Ren

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Engineering

Stony Brook University

August 2015

Stony Brook University

The Graduate School

Yufei Ren

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Dantong Yu - Dissertation Advisor
Adjunct Professor, Department of Electrical and Computer Engineering

Yuanyuan Yang - Chairperson of Defense
Professor, Department of Electrical and Computer Engineering

Shudong Jin
Adjunct Professor, Department of Electrical and Computer Engineering

Michael A. Bender
Associate Professor, Department of Computer Science

Fan Ye
Assistant Professor, Department of Electrical and Computer Engineering

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Scalable End-to-End Data I/O over Enterprise and Data-Center Networks

by

Yufei Ren

Doctor of Philosophy

in

Computer Engineering

Stony Brook University

2015

Data-intensive applications in commercial cloud and scientific computing demand ultra-high-speed end-to-end data access capability between data storage and computing locations. Meanwhile, advancements in hardware systems continuously change the landscape of data centers core capabilities, i.e., computing, networking, and storage. The two trends expose new research and development challenges and opportunities to bring the bare-metal capacity and performance of state-of-the-art hardware to the rising needs for high performance by applications.

Simply deploying and tuning existing software and services on state-of-the-art platforms does not necessarily ensure expected performance due to the over-

head on data-copy in OS kernel functions and conservative network protocol. We adopt holistic approach, from the ground up, to reconsider network protocol, storage management, and software architecture, and align them with the new hardware characteristics to better orchestrate system resources. This is extremely challenging in several aspects. First, we can not rely on existing data-copy based OS libraries and network protocols. Secondly, simple synchronous sequential programming paradigm and network protocol becomes barrier to system performance. Therefore, the new design should follow more complex asynchronous parallel model and carefully investigate the tradeoff between the programming overheads and performance improvement.

Two major objectives are the focus of this work: designing user-level end-to-end protocol and software to coordinate data movements and to bypass OS kernels, and scaling storage caching system performance in multi-core environments with large scale asymmetric memory layout (NUMA). We design and build real systems to achieve these objectives. First, to fully utilize hardware capabilities, we propose an asynchronous memory-centric software framework for high throughput data-intensive applications. We implement a zero-copy data movement protocol using asynchronous Remote Direct Memory Access (RDMA) to maximize the parallelism of data transmission. The design achieves 97% network hardware bandwidth. Our software achieves more than $2\times$ speedup over the existing solutions (for example, GridFTP) in replicating data across the entire storage-to-storage path. Secondly, we design and implement a NUMA-aware caching system for storage area networks that optimizes in-memory data locality on serving raw storage blocks. We further improve its performance with our decentralized and parallel event processing framework. The data locality awareness provides up to 80% throughput improvement on large scale memory caching system; The decentralized event processing shows its linear scalability with the number of threads on multi-core systems.

The unprecedented data volume and the continuing trend of adopting cloud

computing and storage by industry and consumers give rise to the pressing need for efficient software design and network protocol to distribute and replicate data over high performance networks. Our research focuses on the need and proposes a scalable framework to manage and coordinate multi-/many-core computing, deep hierarchy storage and high speed network in a coherent way. Therefore, this framework enables in-situ data retrieval, checksum calculation, encryption, and transmission to address the growing concerns of data privacy, security, integrity and on-demand delivery in the cloud era. It paves the path to harness multi-core/many-core architecture, i.e., GPGPU and Intel Coprocessor, by accelerating data I/O, to replace out-dated software that was designed for traditional rotary magnetic disks, and to enhance the IOPS throughput of storage system to incorporate newly-emerging Solid State Drives (SSD) and Non-volatile random-access memory (NVRAM).

To Qi and Kaylee

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to express my deep gratitude to my advisor, Prof. Dantong Yu. Throughout my Ph.D. studies, Prof. Yu has given me numerous pieces of priceless advice. He patiently listened my thoughts in every meeting discussion through over five years, and provided insightful feedback to motivate my research forward. He also gave me substantial research freedom to explore and broaden my research interests. I am extremely thankful for his support and encouragement to complete this exceptional Ph.D. journey.

My thanks also go to Prof. Shudong Jin, for his guidance to my research exploration and technical writing. I have benefited from numerous discussions during these years.

I would also like to thank my dissertation committee members, Prof. Yuanyuan Yang, Prof. Michael Bender, and Prof. Fan Ye, for their insights and suggestions on my research. Prof. Yang was instrumental to several key steps in my graduate studies and exemplary in her passion and rigorous attention toward research. I would especially like to thank Prof. Bender and Prof. Ye for their invaluable advice to my future research.

Thanks to Dr. Li Zhang and Dr. Yandong Wang, who helped to start my career at IBM T.J. Watson Research Center. Thanks to Dr. Zhikui Wang, Dr. Dejan Milojicic, and Dr. Harumi Kuno, for providing me the rewarding internship opportunity at HP Labs.

As an international student at Stony Brook University, I would also like to express my gratitude to many individuals, including my international advisor,

Jasmina Gradistanac, and ECE department staff members, Susan Hayden and Rachel Ingrassia. They made my life easier and more comfortable.

I am fortunate to have the opportunities to work with stellar and hardworking colleagues: Tan Li, Cheng Chang, Shun Yao, Zhenzhou Peng, Jin Xu, Hao Huang, Li Shi, and Shuchu Han.

Finally, not the least, great thanks to my wife, my parents, and my maternal grandparents, for their love, strong encouragement, and unconditional supports. Kaylee, my dear daughter, your little feet made the biggest footprints in this Ph.D. journey.

Publications

Journal Publications

- Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, Thomas Robertazzi, "Design, Implementation, and Evaluation of a NUMA-Aware Cache for iSCSI Storage Servers", *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol.26, no. 2, pp. 413-422, Feb. 2015, doi:10.1109/TPDS.2014.2311817.

- Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, Thomas Robertazzi, "Design and Testbed Evaluation of RDMA-Based Middleware for High-Performance Data Transfer Applications", *Journal of Systems and Software*, Volume 86, Issue 7, July 2013, Pages 1850-1863, ISSN 0164-1212, 10.1016/j.jss.2013.01.070.

Conference Publications

- Tan Li, Yufei Ren, Dantong Yu, Shudong Jin, "Resources-conscious Asynchronous High-speed Data Transfer in Multicore Systems: Design, Optimizations, and Evaluation", In *Proceedings of Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International , vol., no., pp.1097,1106, 25-29 May 2015 28th International, (IPDPS '15),* May, 2015.

- Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, Thomas Robertazzi, "Design and Performance Evaluation of NUMA-Aware RDMA-Based End-to-End Data Transfer Systems", In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, (SC '13)*, Denver, Colorado, November 2013.

- Tan Li, Yufei Ren, Dantong Yu, Shudong Jin, Thomas Robertazzi, "Characterization of Input/Output Bandwidth Performance Models in NUMA Architecture for Data Intensive Applications", In *Proceedings of the International Conference on Parallel Processing, (ICPP '13)*, Lyon, France, October 2013.

- Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, Thomas Robertazzi, Brian L. Tierney, Eric Pouyoul, "Protocols for Wide-Area Data-intensive Applications: Design and Performance Issues", In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, (SC '12)* , Salt Lake City, Utah, November 2012.

Workshop Publications

- Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, Thomas Robertazzi, "Middleware Support for RDMA-based Data Transfer in Cloud Computing", In *Proceedings of the High-Performance Grid and Cloud Computing Workshop*, Shanghai, China, May 2012.

Chapter 1

Introduction

The fast growing data volume in data centers demands scalable data I/O systems to transfer data efficiently and to cache data effectively for supporting big data processing, analysis, visualization and archiving. The existing and upcoming applications, such as social networks, scientific computing, online storage and inter data center backup, have been generating high volume data traffic in tera-/peta-byte scale [1]. At the same time, data services become more centrally managed among distributed physical storage systems that aggregates data from billions of users. Cloud computing and cloud storage are the example of this paradigm [2]. On the other hand, network capacity in storage area network (SAN), virtual host area network in hypervisor and its Ethernet adaptors (VAN), local area network (LAN), wide area network (WAN), have been exponentially growing in last ten years, largely driven by rapid improvements in opto-electronics technologies. This growth trend which closely reassembles Moore's law, coupled with the fast-evolving software-defined network technologies (SDN), offers necessary bandwidth on demand to upper layer data applications and presents challenges of bridging the widening gap between the software layer and hardware in the network application stack. Therefore, the data I/O systems is one of the cornerstone in accelerating data centers' big data processing capability.

Hardware advancements have been changing the performance of I/O systems and the capacity of memory significantly. The design of existing software I/O systems, however, mostly relies on the general purpose OS abstractions (i.e., system calls) and services, which in turn hide the control plane, i.e., TCP flow control and transmission, from the upper layer application and expose data plane based on data copy, i.e., copy data between application user space and OS kernel space. On one hand, data copies among OS layers and modules consume a large portion of system resources [3] and undermines the software scalability in high performance networks. On the other hand, advanced hardware characteristics motivate researchers to re-think data placement and locality in such environment to efficiently utilize state-of-the-art hardware characteristics.

This dissertation tackles research challenges in the field of high performance data I/O by decoupling software functions, and integrate appropriate components into control plane and data plane. It defines a scalable software framework to cope with hardware advances in enterprise and data centers networks. It achieves the bare-metal performance of high-speed networks for end-to-end data movement and betters data locality in serving cached data of storage systems.

1.1 Research Motivation

1.1.1 Big Data Needs Efficient Movement

Various data-intensive applications require ultra high-speed data transfer capability, such as those in data centers, cloud-computing environments [2], and distributed scientific computing. They frequently need data transfer software to support true end-to-end data and file delivery, i.e., between the storage systems that are attached to the source and the destination hosts. Figure 1-1 shows an intuitive example from the Department of Energy's (DOE's) Magellan cloud

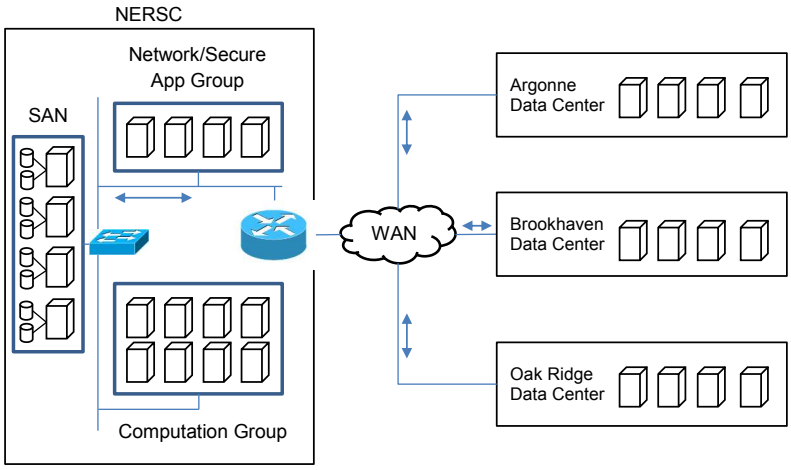


Figure 1-1: Data transfer and synchronization between data centers. This is an example from the Department of Energy's (DOE's) Magellan cloud data centers that are interconnected by multiple 100 Gbps links of the DOE's Advanced Network Initiative.

data centers [4] that are interconnected by multiple 100 Gbps links provided by the DOE's Advanced Network Initiative (ANI). The architectural layout in Figure 1-1 can often be found in the DOE's National Laboratories, for example, three leadership computing facilities hosted at Argonne National Laboratory, Oak Ridge National Laboratory, and the National Energy Research Scientific Computing Center, respectively, and the tier-1 Large Hadron Collider computing facilities at Brookhaven National Laboratory and Fermilab that play a vital role in searching and analyzing exascale experimental data for scientific insights and discoveries [5]. The science programs (climate simulation, astrophysics, high-energy physics, material science, and system biology) at these DOE Laboratories frequently rely on high-performance supercomputers and server clusters, along with back-end storage systems encompassing hundreds of petabyte disk and tape storage, to run computing and data intensive applications, and to move data from experiments and simulations between computing and storage infrastructures and frequently across high-speed wide area networks. It becomes a necessity to design and deliver efficient high-performance data I/O and data transfer systems for these computing facilities. To scale up data transfer to

100Gbps and higher, we must overcome various bottlenecks along the end-to-end paths that consist of hosts, networks, and storage systems.

1.1.2 Hardware Exposes Opportunities and Challenges

Recent hardware advances, such as network protocol offloading and large (non-volatile) memory (RAM) integration in deep memory hierarchy, have been improving bare-metal systems performance and capability significantly. It's a predictable trend that data center servers become more compact (less footprint), more energy efficient (less power consumption), and more powerful (less investment). For example, by 2020, warehouse-scale computers (WSCs) will have up to 100,000 cores and 100Petabyte Non-Volatile Memory (NVM) which are connected by high-speed radix switches and communication links spreaded across several racks [6, 7].

Remote direct memory access (RDMA), non-uniform memory access (NUMA), and multi/many-core are among the mainstream hardware techniques to achieve high efficient architecture and to provide fundamental building blocks for modern supercomputers and data centers. In particular, remote direct memory access (RDMA) [8] is one of these promising technologies that were developed as the supercomputer interconnect technology in the domain of high-performance computing (HPC). By enabling network adapters to transfer bulk application memory blocks to or from remote memory, it eliminates data copies in protocol stacks, reduces software interrupt handling, and achieves low latency and high bandwidth. InfiniBand [9, 10], a ubiquitous RDMA implementation, dominates the technology market of intra-data-center interconnects. Recently RDMA over Converged Ethernet (RoCE) [11] and the Internet wide area RDMA protocol (iWARP) [12] extended the RDMA capabilities to LAN and WAN between geographically distributed data centers. Consequently, RDMA provides an opportunity to facilitate data synchronization and replication within or between data

centers for applications to accomplish their routine tasks in a highly efficient manner. It is necessary to employ these advanced network technologies and protocols to fully utilize the bare-metal bandwidth of high-speed networks (100 Gbps and higher), and to eliminate network performance bottlenecks. One of our main goal in this dissertation is to introduce RDMA into the high throughput cloud computing and confirm its viability in new domains.

In addition, as the numbers of CPU sockets and cores per CPU processor grow in the multi-core architecture of modern computer hosts, it becomes increasingly difficult and inefficient to ensure the same memory access latency across all CPU cores by using a centralized and shared memory bus. A state-of-the-art CPU architecture integrates a memory controller as a core component within the CPU die and discards the external controller hub that would have become a bottleneck in a multi-core architecture [13]. In modern architecture, memory banks at different locations in a motherboard are directly attached to their corresponding CPUs, while multiple CPUs maintain cached data consistency by following cache coherent protocol. The accesses to a remote memory bank active inter-processor communication links and incur extra costs. Therefore, the access latencies from a specific CPU core to different memory banks are no longer same. From another aspect high speed interconnect, for example QuickPath Interconnect [13] and Hyper Transport [14],, helps a system achieve higher resource density and integration [15–17]. Although high-speed connectivity between CPUs greatly speeds up random memory access, an application that maximizes local memory accesses always outperforms than those that does not.

1.1.3 Limitations in Existing Solutions

Data-intensive applications often rely on data transfer services and place stringent requirements on the performance of the back-end storage systems and the

front-end network interfaces in modern multi-core servers. Existing data transfer software design and implementation heavily rely on operating systems abstraction and services [18]. They involve several data copies and trigger frequent interrupt services in processing networks packets. However, for transferring data at 100 Gbps and higher, existing solutions experience multiple bottlenecks along a full end-to-end path. For instance, existing TCP/IP based data transfer protocols and software contribute high CPU consumption due to excessive data copies; existing services of storage area networks are NUMA-agnostic and thus can not fully utilize the aggregate memory bandwidth of NUMA systems. As a result, existing software design and implementations often suffer scalability problems.

In storage systems, many existing applications intuitively achieve the best performance with a coarse-grained control policy by binding all related threads to a single NUMA node and allocating applications' memory to the local NUMA node via the `numactl` command tool. However, two types of popular applications require a fine-grained, NUMA-aware design for scheduling threads and organizing memory: the first one are memory intensive applications (i.e., database engines) that require a large memory footprint distributed across different NUMA nodes to speed up aggregated performance; and the second type are storage service applications that rely on kernel cache utilities, i.e., page cache. Because the standard file interface, e.g., `read` and `fread`, does not provide any NUMA-related configuration parameter, the applications using the interface cannot explicitly manipulate kernel-level cached data on a designated NUMA node. For example, the existing iSCSI target software that is widely used to manage scalable storage blocks for enterprise databases and distributed file systems, has the property of these two types of applications: it requires a great amount of memory for serving cached data and relies on OS page cache that is agnostic to NUMA. Our research exploits an explicit and fine-grained NUMA-aware design to expedite its cache access.

1.2 Research Challenges and Research Goals

In this section, we elaborate challenges and research goals on achieving scalable data transfer systems.

1.2.1 Achieve Zero-Copy In An End-to-End Data Path

The stubborn speed disparity between the CPU and memory, named "Memory Wall", which is common in the previous single-core architecture, will continue to exist and even deteriorate with multi-core architecture. As detailed in [3], latency in memory access will be a major bottleneck in computer system. Pursuing high CPU clock rate is not sustainable due to the power wall problem, i.e., severe transistor current leakage under high clock rate leads to uncontrolled power consumption and generates excessive heat that is hard to dissipate. From the system architecture perspective, memory latency might partially negate the high CPU clock rate and the associated computing capacity. As a result, chip designers turn to exploring multi-core architectures and pack more cores into a single CPU die. Consequently, the speed imbalance between fast-growing number of CPU cores and memory will become more severe in the multi-core architecture.

Meanwhile, network performance is significantly improved by zero-copy and protocol offloading technologies such as remote direct memory access (RDMA). It enables network interface to transfer memory blocks in user space to the memory in remote hosts without CPU involvement. This is different from traditional TCP based solutions in which data are copied multiple times: applications copy data to kernel sending buffer, and then OS kernel copies the data again the lower layer of network stacks before reaching network interface cards. Memory bandwidth is not efficiently utilized in this scenario.

It's challenge to build a user-level communication protocols to bypass existing kernel communication support. Different from traditional data transfer

software, an efficient software design has to relies on zero-copy mechanism and restricts data plane from involving kernel space [19, 20]. On the other hand, the control plane of data transfer needs to optimize communication, and fully utilizes network bandwidth with a user-level communication management protocol, including multiple streams, memory credit management, and flow control. We will elaborate on decoupling control plane and data plane by using asynchronous software framework for data transfer software in Chapter 3.

1.2.2 Scale Zero-Copy Based Network Protocol In WANs

In addition to scale software performance in local area networks, we study the issues related to designing a scalable high-speed network protocol and improving application performance in wide area networks. When the bandwidth delay product is large, an application inevitably needs to manages a large mount of on-fly network packets and their related memory buffers: a zero-copy based solution can not recycle memory block until the data successfully reaches their destination. We focus on RDMA primitives, and investigate the interaction between application protocols/software and network hardware capabilities. An adaptive network mechanism is critical to improve both memory and network resource utilization.

To extend the zero-copy mechanism in WAN, we design an application layer protocol for RDMA networks in Chapter 4, as part of a middleware layer that integrates network access, memory management, and multi-tasking. We address various issues in the RDMA implementation, such as buffer management, memory registration, and the parallelization of RDMA operations, all of which are vital to delivering the benefits of RDMA to applications. Using this protocol, we implemented an RDMA-based FTP software, RFTP [21–24]. This developmental work is funded by a larger project to exploit the full capacity of a 100Gbps network in the U.S. Department of Energy's Energy Sciences Network (ESnet).

1.2.3 Caching Locality in Large-Scale NUMA Systems

Storage system is another important aspect for data movement, and caching is widely used to improve the data accessing performance. Asymmetric memory architecture such as NUMA dominates enterprise and data center systems. It scales hardware memory capacity by connecting CPU and memory nodes with multiple lanes of high-speed processor interconnect. The asymmetrical performance on accessing different memory locations becomes notable as a single server integrates more CPU nodes with complicated interconnects [25]. Therefore, it's critical to improve data access locality in such highly integrated systems.

In Chapter 5, we will construct a storage area network (SAN) with the iSCSI and iSER protocols (iSCSI Extensions for RDMA) over InfiniBand based storage fabrics among which we utilize the Remote Direct Memory Access (RDMA) to move data blocks. The existing iSCSI software relies on the OS page cache which is NUMA-agnostic. We design a NUMA-aware cache at the userspace of iSCSI software to achieve better data locality, to eliminate the bottleneck effect of remote memory access and thereby to lead to high efficiency on serving cached data. The software product is tested rigorously and is demonstrated applicable to supporting various data-intensive applications and exascale supercomputers that constantly use powerful NUMA computers to serve iSCSI/iSER based storage.

1.2.4 Towards Scalable End-to-End Data I/O Systems

Our research goal is to design the end-to-end data I/O systems that can fully utilize state-of-the-art hardware advancements, and is scalable to the upcoming hardware evolutions. As shown in the previous section, existing solutions are not optimized to harness the new hardware features. Simply deploying existing solution in state-of-the-art hardware environment leads to waste resources and

incur unsatisfied user experience. This dissertation tackles the scalability problems in the network control path, the data transmission pipeline, and the storage caching layer. Particularly, Our research targets overcoming the aforementioned research challenge and implementing a real systems that can efficiently satisfy the end-to-end data I/O requirements in 100Gbps networks and beyond.

1.3 Dissertation Contributions

The main contributions of this dissertation are listed as follows:

- **Proposing asynchronous concurrent event-driven systems.** We present a memory-centric software framework for high speed data movement. Based on this framework, we implement an integrated, multi-level, asynchronous processing model to unify the computing, storage, and networking operations and to maximize their performance. By using this processing model, we develop a data-driven, memory-centric, software architecture for data-intensive applications.

- **Designing scalable RDMA-based data transfer protocol for wide area networks [21, 22, 24].** We design an application layer protocol [22] for RDMA networks, as a component of a middleware layer [21] that integrates network access, memory management, and multitasking. We address various issues that are related to the efficient implementation and implement a series of optimization methods, such as buffer management and memory registration, and the parallelization of RDMA operations, all of which are vital to delivering the benefits of RDMA to applications. Using this protocol, we implement an RDMA-based FTP software, RFTP. Our developmental work is a part of a large project in Department of Energy (DOE) to exploit the full capacity of a 100Gbps network in DOE's Energy Sciences Network (ESnet) [26]. Third, we also integrate TCP

zero-copy mechanism, sendfile and splice, to accommodate network environment without RDMA capability [24]. In addition, we show the importance of resource awareness, i.e., NUMA multi-core and parallel I/O capacity, in TCP based data transfer systems [27]. Fourth, we present our extensive experiments to evaluate the performance of our protocol, particularly over wide-area networks. We show that our tool has much higher performance compared with those existing widely used data transfer tools, such as GridFTP [28].

- **Providing NUMA-aware caching for storage area networks [29].** We characterize the bottleneck of processing I/O requests [30] in the standard iSCSI/iSER software, and show the benefit of a NUMA-aware caching strategy. Second, we design and implement a NUMA-aware cache at the user level for the Linux SCSI target software and its iSCSI/iSER drivers. This cache module reduces remote memory access penalty, and improves throughput and latency of the target process. Its core design principles, including cache partitioning, node-affinity ranking, and request forwarding, are applicable to other memory-intensive applications. Third, to tackle the scalability problem on small I/O request processing, we propose a decentralized I/O event processing model for multi-core and NUMA systems. Fourth, we evaluate our solution on a 4-node NUMA testbed. Compared with the default Linux page cache, our user level NUMA-aware solution shows impressive performance gains with both synthetic and real-life workloads. Finally, this solution, initially developed as a enhancement to the backend block storage devices, ultimately provide performance improvements to front-end bulk data transfer applications (i.e., RFTP) and small I/O intensive applications (i.e., key-value stores).

- **Optimizing the end-to-end data I/O performance [23].** We elaborate the design, tuning, and performance evaluation of a novel high-speed data

transfer system for delivering data at 100 Gbps in an end-to-end fashion. The system utilizes a pair of multi-core front-end hosts (sender and receiver). First, our back-end storage systems use the standard iSER protocol that is configured for high-speed data access. The protocol enables InfiniBand based data delivery from the back-end storage systems to the front-end hosts. This design allows us to eliminate the bottleneck in back-end storage with scalable InfiniBand fabrics. Second, we integrate our RDMA-based file transfer protocol, RFTP [21,22,24], into the end-to-end data transfer system, to maximize bandwidth throughput and to minimize the host processing overhead. Third, we perform a thorough NUMA tuning to optimize the performance for all hosts along an end-to-end path, and to minimize the impact of host processing overhead. We note in the current implementation of iSER or RFTP, the NUMA factor is not considered. We present the observed performance benefit of simple NUMA tuning in this chapter. Our design is the first one to achieve 100 Gbps end-to-end real data file transfers between one pair of commodity hosts. We evaluate our system comprehensively by designing the testbed to closely resemble the production environments that are common in large national laboratories and commercial cloud computing providers. Furthermore, we perform inter-data center data transfer tests along the long-haul high bandwidth paths of over 4000 miles long and confirm our protocol and tool clearly outperform the alternative design and implementations.

1.4 Dissertation Overview

Figure 1-2 shows the problems that each chapter tries to tackle and their scopes and locations with respect to a real-world cloud computing environment. The rest of the dissertation is organized as follows. Chapter 2 elaborates the concepts that are related to this dissertation. Chapter 3 proposes asynchronous

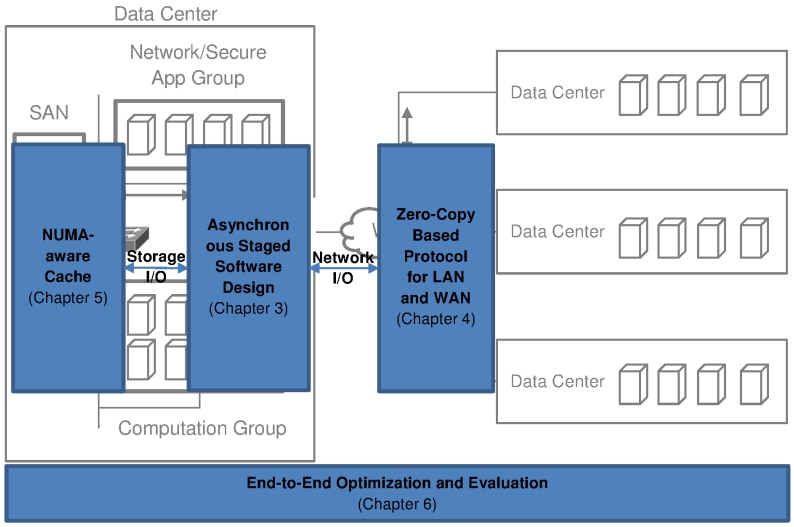


Figure 1-2: Dissertation overview.

staged software design, followed by RDMA-based zero-copy network protocol for WAN in Chapter 4. Chapter 5 presents design and implementation of NUMA-aware cache for improving iSCSI servers performance. Chapter 6 elaborates system optimization methods along an end-to-end data movement path. We summarize related research in Chapter 7 and offer our conclusion in Chapter 8.

Chapter 2

Background

This chapter provides a background of various aspects that are integral components of this dissertation. First, we provide a background on RDMA and high performance networks, e.g. RDMA semantics and memory registration in programming with RDMA networks. We then describe the asynchronous programming models and libraries for network I/O and storage I/O. We also discuss the NUMA architecture and the libraries and utilities of constructing NUMA-aware applications. Finally, we discuss the caching layer in iSCSI system and the necessity to design a NUMA-aware caching system for iSCSI protocol.

2.1 RDMA and Zero-Copy Techniques

Traditional TCP/IP applications involve multiple data copies. As shown in Fig. 2-1, sender copies data from user space memory to kernel space. It consumes a large amount of CPU resource and involves multiple context switches. Then, the sender's network card driver fetches data from kernel space by DMA operations and sends data packets into communication links. On the receiver end, NIC driver interrupts kernel service and places the packet into kernel TCP socket buffer. Thereafter, the kernel copies data into user space and interrupts the receiving application for data arrival.

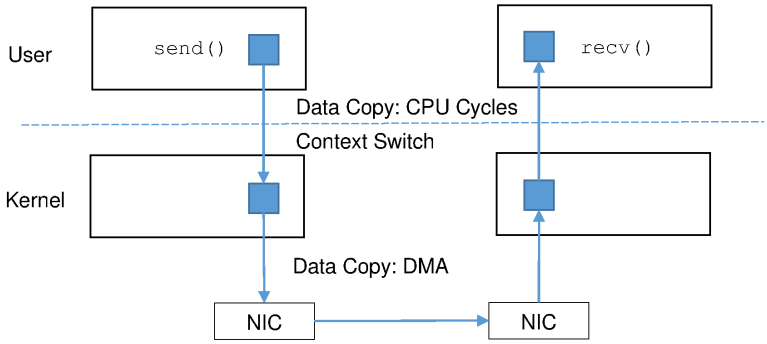


Figure 2-1: Data path of TCP/IP applications.

RDMA, remote direct memory access, is introduced to eliminate data copy overhead in high performance networks. It moves data from source host memory to a remote host memory with kernel-bypass and zero-copy operations. Before a communication party starts RDMA operations, both sides perform memory registration, as shown in Fig. 2-2. Here to register a *memory region* means to pin a designated chunk of memory into physical RAM and prevent it from being swapped out by OS. The application uses `ibv_reg_mr()` to specify the memory space to be pinned, which in turn uses the system API and the corresponding function `mlock()` to register the memory region in the physical main memory space. In addition, applications can use the interface to setup desired memory protection attributes. The registered memory information is also stored in a page table of the RDMA network interface that manages and controls direct data accesses into/from the registered memory.

RDMA write is one of the zero-copy operation that bypasses kernel space. Once getting an RDMA write *work request* from an application, the RDMA NIC (RNIC), also known as host channel adapter (HCA), uses DMA operations to fetch data from user space memory directly without kernel involvement. The work request contains the information of the local source memory and the remote destination memory. At the remote side, its RNIC contains the page table for the target region. Upon data arrival, the RNIC of the remote side adds the

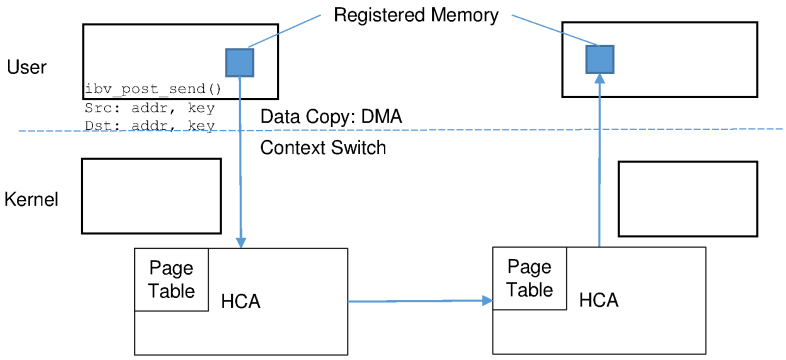
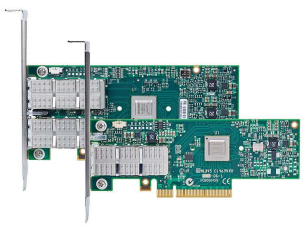
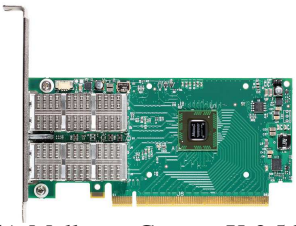


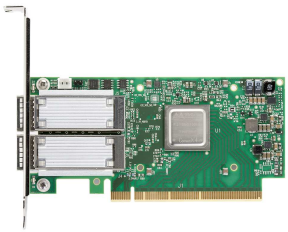
Figure 2-2: Data path of RDMA applications.



(a) Mellanox ConnectX-3 40 Gbps network interface.



(b) Mellanox ConnectX-3 56 Gbps network interface.



(c) Mellanox ConnectX-4 100 Gbps network interface.

Figure 2-3: Mellanox ConnectX supports both RDMA over InfiniBand and RDMA over Converged Ethernet (RoCE) devices.

target offset, obtains the corresponding physical pages, write data to the user space memory directly without kernel involvement. RDMA read operation is used for retrieve remote memory information bypassing kernel involvement on both local and remote sides' OS kernel, and its data movement path is opposite to the RDMA write's.

RDMA is designed to be scalable to bundle multiple links (channels) to increase the communication bandwidth. The last three generations of RDMA device support either InfiniBand, Internet, or RDMA over converged Ethernet (RoCE), and achieve 40 Gbps (QDR: quad data rate), 56 Gbps (FDR: fourteen data rate), and 100 Gbps (EDR: enhanced data rate). Figure 2-3 shows a family of InfiniBand and RoCE devices from Mellanox Technology.

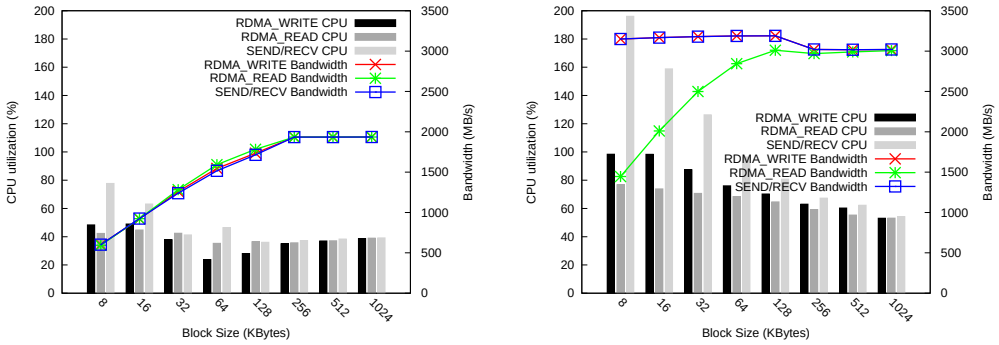
In summary, RDMA coordinates hardware and applications to eliminate data copies and to accelerate network processing with hardware support. It achieves

low latency and high throughput comparing to the traditional TCP/IP network stack. We explore the RDMA semantics, remote memory operations and programming interfaces in the following section.

2.1.1 RDMA Semantics and Performance Analysis

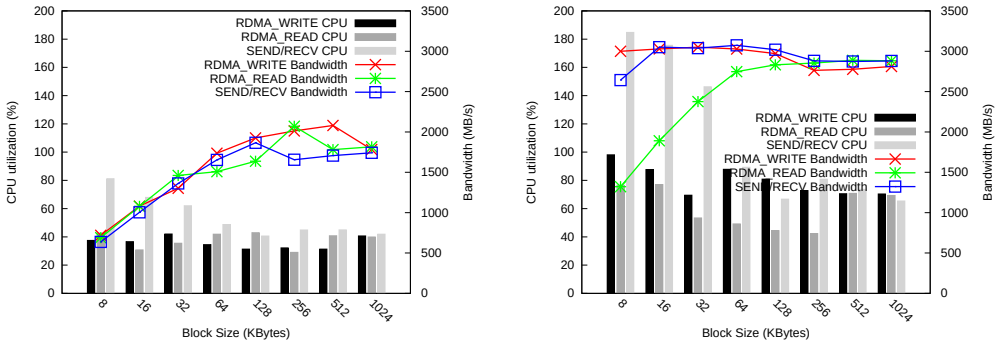
RDMA offers two types of message transfer semantics: Channel and Memory. The former, "send/receive", that is also referred as two-sided operation, involves the OS network kernel for the completion event notification at both the source and sink hosts of data transfer once a network connection is established [31]. The communication channel between source and sink is modeled as queue pair (QP). Each QP consists of a sender queue and a receiver queue, whilst each queue represents one end of the channel. Before an application uses RDMA to transfer data, the receiver posts a work request to the receiver queue. Thereafter the sender posts a work request to the sender queue. Both the sender and receiver will get a completion event after the data transfer completes. On the other hand, the memory semantics of RDMA, "read/write mode", is often termed "one-sided operation". With this type of RDMA, the receiver advertises its available registered memory to the sender, including the address and other information about the memory region, so that the sender can directly use RDMA write to write a chunk of data into the specified memory location at the receiver host.

To decide which type of RDMA semantics to choose to move bulk data, we consider various performance factors. We design an RDMA I/O engine for the fio benchmark and stress test tool [32] that offers flexible parameter settings and excellent capabilities to report the performance statistics of both synchronous and asynchronous I/Os. Compared with the standard OFED (OpenFabrics Enterprise Distribution) benchmark tools, our approach is easier to collect comprehensive performance statistics from the I/O module, including CPU usage, I/O latency, bandwidth, and I/O performance distribution. The RDMA engine



(a) I/O depth is 1 (b) I/O depth is 32

Figure 2-4: RDMA semantics performance evaluation in the RoCE Environment.



(a) I/O depth is 1 (b) I/O depth is 32

Figure 2-5: RDMA semantics performance evaluation in the InfiniBand Environment.

uses asynchronous I/O and allows our test program to simultaneously post multiple I/O requests. We conducted a comprehensive set of test cases with varying block sizes and maximum number of concurrent blocks in flight (also called I/O depths) for the two types of RDMA semantics.

Figure 2-4(a) and Figure 2-5(a) show that with a small I/O depth, RDMA WRITE, RDMA READ and SEND/RECEIVE exhibit similar performance, while the CPU consumption of SEND/RECEIVE is much higher than that of the others. The high CPU consumption reflects the fact that SEND/RECEIVE involves both the data source and sink during transfer, and the sink must process the same number of RDMA events as the source. However, RDMA READ/WRITE only

handles RDMA events at one end. A high I/O depth improves bandwidth performance as depicted in Figure 2-4(b) and Figure 2-5(b). To improve RDMA performance, an application must post multiple I/O tasks in flight to fully take advantage of OFED's asynchronous programming interface. We made several observations from these experiments: 1) RDMA WRITE and SEND/RECEIVE perform better than RDMA READ; 2) all test cases set block size in the range from 16KB to 128KB to achieve the best bandwidth; 3) performance saturates when the block size is bigger than 128KB; 4) CPU usage decreases when the block size increases because of fewer interrupts; and 5) during their peak performance, the CPU usage of SEND/RECEIVE is higher than that of RDMA WRITE.

Since the arrival rate of incoming data is unpredictable, the data sink must post sufficient registered buffers at the receive queue in advance before the data source sends data. Otherwise, the data source may encounter the error of Receiver Not Ready (RNR). The error will pause the source, and lead to low performance and under-utilized network resource. Tian et. al. proposed to introduce a control message mechanism to avoid the RNR error in their protocol implementation [33], where the source must wait for credits to be piggybacked with the message from the data sink before the source further posts more tasks into the send queue. In summary, RDMA WRITE performs the best with the least CPU consumption in all test cases, and I/O depth should be set to a relatively large number, as identified in the previous testing results. Therefore, we designed a hybrid data transfer protocol that exchanges control messages via SEND/RECEIVE, and transfers bulk data via RDMA WRITE.

2.1.2 OS Kernel Zero-Copy Techniques

In addition to the RDMA hardware-based zero-copy acceleration, software based optimization is another approach to reduce data copies and the associated pro-

cessing overheads, for example the system calls `sendfile()` and `splice()`, newly supported by modern operating system kernels. In contrasting to TCP/IP stack that involves multiple data copies and incurs high CPU load in the high speed data transfer environment, the `sendfile()` primitive in Linux performs pass-by-reference to copy data between one file descriptor and another, e.g., between a disk file descriptor and a socket descriptor. Because this type of copying occurs within the kernel and merely exchanges data reference pointers, `sendfile()` is more efficient than the combination of the standard POSIX read and write functions which require physical data copies to and from the user space.

`splice` further extends the functionality of `sendfile()`. It is a Linux-specific system call that moves data between a file descriptor and a pipe without a detour to user space. `splice()` requires to setup a pipe buffer prior to the function call. A pipe buffer is an in-kernel memory buffer and transparent to user space processes. A user process can splice the content of a source file, e.g., a socket, into this pipe buffer, then splice the pipe buffer into the destination file descriptor, e.g., a file in the disk systems.

2.2 Asynchronous High Throughput Computing and Thread-based Concurrency

General asynchronous I/O model separates I/O issuing and I/O status checking to avoid blocking operations. It can be utilized to overlap computing and I/O resources efficiently. Inspired by the RDMA based asynchronous communication model, we extend it to be a generalized asynchronous computing model and apply it to many other aspects of an end-to-end high throughput data path. A typical OS interacts with heterogeneous devices and handles various system calls via hardware/software interrupts (events) asynchronously, while providing

a simple uniform POSIX interface to applications. Thus, the manner the software handles asynchrony is fundamental to the performance of applications and their hosting server in high throughput computing.

Newly developed multi-core processor technology is capable of packing dozens of CPU cores in a single server, and therefore it is cost-effective to fully utilize these resources first within the server before reaching out to other servers for high-throughput computing. A simple approach is to maximize the concurrency of a server via a multi-threaded parallel programming model, wherein a master program is responsible for partitioning a large task into smaller ones and assigning each one of them to a different, dynamically created thread to parallelize and expedite processing. Here, the master program often relies on complex inter-thread synchronization schemes to synchronize thread status and ensure task completion. For example, OpenMP [34] uses the lock and barrier primitives. The slowest thread always dominates the overall execution pace and performance of an application that is implemented with this model. Given the uncertainty of server system status, non-uniform memory access, and processing loads, it is possible that several threads may hold CPU cores and wait idle for the slowest thread to complete. Thereby, this parallel model does not guarantee the optimal server performance.

On the other hand, asynchronous computation proceeds via triggering events and acting upon them instead of being directly orchestrated by a central master program. Within the asynchronous computing paradigm, a set of threads that entail a sequence of data processing instructions are created apriori and bound to all available cores to reduce context switches and to minimize overheads. A thread, along with its associated instructions and one allocated CPU core, forms a unit of computing resource, and is made available to the main application via event exchanges.

Even when an application involves multiple threads, some of them engage in the synchronous parallel paradigm (for example, wait for and respond to user

requests) and some of them perform asynchronous computing, an asynchronous computing thread executes at its own rate without affecting the aggregate performance, and remains in a background mode with regard to the master program running in the foreground. No explicit join step is required once an asynchronous thread completes its current data processing assignment. Instead, it will proactively check for the next event to get more task assignments.

In the context of our work, a high throughput computing program often loads initial input data from internal and/or external storage and high speed network adapters, applies a series of transformations to data, and writes output back to storage media and/or network stacks. There is less inter-dependency between two threads assigned to different datasets or segments of memory. To avoid excessive data copies and relocations, we adopt a data-and-memory-centric computing paradigm: an end-to-end high throughput program is regarded as a series of operations that are applied to memory-resident data: storage I/O to populate and evacuate memory space, data transformations (for example, compression/decompression, encryption/decryption, Fourier Transformation and its inverse), and network I/O to import and export memory space. These operations interact with a main program asynchronously and apply changes to shared memory space even though their nature, duration, and internal implementation are heterogeneous.

2.2.1 RDMA Asynchronous Programming Model

The RDMA architecture permits an application to access hardware directly in the user mode. RDMA provides a set of verbs (function calls) that customizes and optimizes the RDMA-aware network interface cards. All I/O operations involved in RDMA programming are asynchronous: an application posts an I/O request into hardware, and then polls a corresponding completion event to confirm the outcome of the I/O, either success or failure with a particular cause.

To perform RDMA operations, an application must establish connections between local and remote host. The Queue Pair (QP) in the RDMA programming plays an equivalent role as the socket interface to TCP/IP stack. The QP needs to be setup and initialized on both sides of connection. An application uses Verbs to interact with Communication Manager (CM) which subsequently exchanges the information about the QP and sets up the QP. Once a QP is established, application calls the verbs API to perform one-sided RDMA reads, RDMA writes, and other atomic operations. Applications can also perform serialized two-sided send/receive operations which are similar to the socket `send()/recv()`.

2.2.2 Asynchronous Storage I/O

Asynchronous I/O functions enables applications to saturate the storage I/O performance with a single thread by submitting a batch of concurrent I/O requests to the disk queues. There are two types of asynchronous I/O interfaces: *POSIX AIO* and *Linux native AIO - libaio* [35]. These two implementations are fundamentally different: the POSIX AIO creates multiple threads within OS, each of which performs normal synchronous and blocking I/O. Essentially it wraps synchronous I/O functions with multiple threads to emulate the behavior of asynchronous I/Os. On the other side, libaio is truly asynchronous, and directly supported by the OS kernel which internally queues I/O requests and submits to devices in an optimized manner. We choose libaio for our implementation method in the following chapters.

2.3 NUMA Architecture

In a state-of-the-art NUMA system, A CPU die, along with its memory controller and local memory banks, comprises a *NUMA node*. A NUMA node references its directly attached memory banks as *local memory* and those at-

tached to other CPU dies as *remote memory*. Multiple NUMA nodes in a host are connected by high-bandwidth and low-latency interconnects, such as Intel QuickPath [13] and AMD HyperTransport [14]. Due to the traffic contention on memory controllers and interconnects, remote memory accesses cause longer access latency and lower throughput compared to local accesses [36].

2.3.1 SMP and NUMA Architecture

In multi-core era, symmetric multiprocessing (SMP) was first introduces to increase a single server's processing capacity. It connects two or more identical processors to a single, shared main memory by using a shared system bus. However, connecting processors with a shared system bus suffers the scalability problem when the number of processors increases because of the contention for the shared bus resource. Modern servers typically adopt the NUMA architecture and use dedicated CPU interconnects instead to scale memory system performance.

2.3.2 Asymmetric Memory Layout

Because interconnect topologies in NUMA system can be 2-D torus or even 3-D torus, the distance or the number of "hops" in NUMA system between the CPU which runs an application and the memory location where the application's data resides varies and leads to different performance behavior in terms of latency and bandwidth. On one hand, this asymmetric memory layout further enlarges the performance disparity among the memory accesses by different CPU cores. On the other hand, because of the shared memory architecture, all processors enforce the cache coherence policy to simplify application-level development. NUMA-agnostic applications incur remote memory access penalty and unstable performance. Therefore, applications has to be tuned accordingly to gain better performance in NUMA systems.

2.4 iSCSI/iSER and Storage Caching System

This section presents the research problems and efforts on cache performance in iSCSI/iSER systems.

2.4.1 iSCSI and iSER

The Internet Small Computer System Interface (iSCSI) is an IP-based approach to build storage area networks. It supports storage virtualization over local area networks (LANs) and even wide area networks (WANs) by encapsulating SCSI commands and payload within IP packets, which are exchanged between clients (*initiators*) and SCSI storage devices (*targets*). Ko *et al.* proposed a standard extension of iSCSI for RDMA networks, named iSER [37]. The Linux SCSI target framework [38, 39] is the de facto iSCSI target software supporting various target drivers, such as the iSCSI driver for TCP/IP, the iSER driver for RDMA [40], and several vendor specific drivers (e.g. IBM, QLogic, LSI). We design and implement a cache module in user space for this framework, and provide the same interface for all transport drivers to interoperate with the cache module.

2.4.2 Caching Layer in iSCSI/iSER

Several caching mechanisms have been investigated and developed to improve the overall performance of the iSCSI system. He *et al.* proposed a strategy of combining non-volatile RAM and log disk to cache iSCSI traffic in initiators [41]. For a shared target server, however, synchronizing cache status among multiple initiators incurs a significant cost. Wang *et al.* proposed an on-board cache in a network interface card. Given a NIC cache hit, cached data does not need to traverse the host's PCI bus and memory controllers. A "Helper" cache in an iSCSI target server has advantages over the on-board one as it eliminates the limitation to cache size [42].

From a system perspective, storage area networks consists of a three-tier memory cache hierarchy, including application client cache, file or database server cache, and storage server cache, and cache layer often duplicates a fraction of the content already cached by another. To convert the *inclusive* cache hierarchy with duplications to the *exclusive* one, researchers have proposed *hierarchy-aware caching*, which maintains the existing I/O interface and is transparent to the application client, and *aggressively-collaborative caching*, which requires centralized cache management by the application client. For hierarchy-aware caching, an eviction-based placement policy [43] and Multi-Queue (MQ) algorithm [44] reduce network traffic and improve the cache hit rate at storage servers. For aggressively-collaborative caching, Wong and Wilkes proposed a "DEMOTE" operation to evict the content from the frontend and re-cache it into the backend storage server to ensure *exclusive* caching [45]. Chen *et al.* found that the aggressively-collaborative caching provides a mere 1.0% improvement over hierarchy-aware caching for most workloads and cache configurations in their real system experiments [46].

2.5 Summary

In this section, we introduce RDMA networks and the NUMA architecture. We also provide necessary background knowledge on the state-of-the-art research works on the RDMA programming model, storage area networks, and the caching layer in the iSCSI and iSER systems.

Chapter 3

ACES Software Architecture

Asynchronous storage I/O and communication exhibit advantages over their synchronous counterparts and can achieve bare-metal performance. Various types of asynchronous processing (computing, storage, and networking), however, have never been considered holistically in today's high throughput computing, and therefore their advantages and flexibility have never been shown to cope with big data processing. In this chapter, we present a memory-centric software framework for high-throughput data transfer applications, called asynchronous concurrent event-driven system (ACES).

3.1 Introduction

Scientific and industry applications are generating increasingly large amounts of data, often referred to as "big data". Effective analysis and accurate inference from these data towards knowledge discovery require high-throughput data processing, transforming, transporting, and sharing among data management/analytics specialists and application-domain scientists. A deluge of parallel and distributed programming paradigms have been proposed to address this "big data issues": Google MapReduce [47] and its open source Hadoop [48], Grid Computing [49], and cluster computing [50]. All these paradigms parti-

Hardware	Hardware Advances	Software Policy
Network	RDMA, TCP offloading	asynchronous I/O event-driven
Disk	SSD, TCQ, NCQ	asynchronous I/O event-driven
Memory	NUMA, memory wall	zero-copy
CPU	Multicore Encryption acceleration	thread-based concurrency

Table 3.1: Software policies for each part of hardware advances

tion data and dispatch data processing tasks into a large number of computer servers for an aggregated high-throughput computing (HTC) performance.

The steady progress of VLSI technology continues adding more cores, higher network bandwidth, and faster storage into a single commodity server than before. These technology breakthroughs afford rethinking the existing computing models for a single Linux server to meet the performance, energy, and efficiency requirements of various HTC applications. Meanwhile, technological trends indicated that hardware throughput/bandwidth continues to be the biggest winner, and outpaces other performance components (for example, latency) with individual servers and computing facilities [51]. In the network and storage fields, disruptive innovations, such as CPU bypass Remote Direct Memory Access (RDMA) and semiconductor flash drives, have improved the throughput of I/O facilities by two to three orders of magnitude. Single host computation density and capacity are continuously improved by multi-/many-core technologies. In contrast, the speed of main memory is relatively slow and memory controller is generally the bottleneck in symmetric multiprocessing (SMP). This led to the popularity of Non-Uniform Memory Architecture (NUMA), which obtains a good aggregate memory performance, but complicates application tuning of memory accesses cross the asymmetric layout. Given the expensive memory copy operations within the NUMA servers, zero-copy primitives (such as direct I/O in storage and RDMA in network) becomes more attractive in application

design and implementation. Furthermore, high performance hardware becomes competitive in term of price and can be economically added into a commodity servers [52]. Given these trends, it is important that we understand how to leverage the advantages of these hardware advances to build high throughput applications.

Unfortunately, existing software design in high-throughput computing cannot survive rapid changes in computer technology. In terms of data transfer which has stringent requirements on throughput, existing solutions emphasize and satisfy a fraction of user demands by a unified design philosophy. For example, OpenSSH scp/sftp [53] concentrates user requirements on security and believes in clean and simple code. It heavily relies on traditional operating system facilities such as pipe, `fork()`, synchronous I/O to maintain its simple design choice. GridFTP, a popular data transfer tool in grid computing, focuses on TCP throughput optimization and makes use of an event-driven software architecture to manage numerous concurrent TCP connections [28]. However, with the increase of hardware bare-metal throughput, no single design philosophy and methodology is currently able to take full advantage of the range of hardware advances. As shown in table 3.1, a hybrid software design policy is required to extract the best performance from each part of a computing system.

In this chapter, we present a memory-centric software framework for high-throughput applications: *asynchronous concurrent event-driven system* (ACES). Our framework divides an end-to-end data path into a series of stages, each of which implements either an (storage/network) I/O task or a computation task to check, search, or transform data. We use explicit task queues to connect the stages and utilize thread synchronization primitives to efficiently handle tasks. Within each stage, an application selects a combination of asynchronous, concurrent, and event-driven operations to maximize the I/O and computing performance.

We implemented a prototype high-performance secure data transfer applica-

tion, called ACES-FTP, on top of the proposed ACES framework. In particular, ACES-FTP leverages asynchronous I/O and direct I/O operations to expedite disk access, maximizes RDMA network performance through a hybrid design of event-driven and concurrent threads, makes use of zero-copy primitives to offset memory copy overhead, and offloads computation intensive encryption to dozens of CPU cores. In end-to-end data transfer tests over high performance networks, ACES-FTP demonstrates three times higher throughput than scp while maintaining the same level of data security. During these performance test, we have achieved secure data transfer at line speeds of 80 Gb/s using off-the-shelf hardware. To further explore the benefit of asynchronous computing, we present an asynchronous implementation of a data integrity function for iSCSI servers and, and show it achieves 140% higher IOPS over its synchronous counterpart in small size dominant workloads.

Our contributions are as follows: firstly, we implement an integrated, multi-level, asynchronous processing model to unify the computing, storage, and networking operations and maximize their performance; secondly, based on this processing model, we develop a data-driven, memory-centric, software architecture for high-throughput data-intensive applications; finally, using this software architecture, we provide a reference implementation on how to transfer data securely at speeds of 40G/100Gbps or higher.

3.2 Existing High Throughput Solutions

Most existing works apply the asynchronous paradigm in only one or two aspects of processing (event-driven programming, asynchronous storage I/O, and one-sided RDMA transport), and asynchrony is implemented at a single level. In this chapter, we propose a two-level asynchronous computing model to optimize application performance from the high level programming logic to the low level devices. An asynchronous computing thread that transforms data can

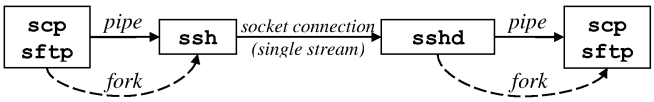


Figure 3-1: Data transfer workflow with OpenSSH `scp` and `sftp`.

asynchronously off-load computation-intensive operations to a many-core system such as the Intel Phi coprocessor or GPGPU hardware [54, 55]. For example, SSLShader [56] implemented SSL acceleration with modern graphics processor units (GP-GPU), where expensive public/private key encryption and the associated exponentiations and modular operations are off-loaded to hundreds of simple GPU cores.

However, two-level asynchronous hierarchy gains more and more attention in high performance exascale computing [57], and provides a promising model to mitigate variations and uncertainties that arise from heterogeneous problem domains, dynamic system load, hardware devices and their optimal access mechanisms (sequential and/or random) and execution costs and duration.

Existing end-to-end high throughput solutions often create a data moving path by reusing OS functions and kernel facilities, such as forks and pipes, and by integrating user function libraries in a synchronization logic. This simplified software design and implementation. Yet most of those solutions are not able to meet the performance requirements in HTC, due to the overhead for data copies, process management, etc. For example, a secure data transfer involves six steps: reading (from storage), encrypting, sending, receiving, decrypting, and writing (into storage systems). The secure data transfer software - OpenSSH `scp` relies on `ssh`, which compresses and encrypts data by utilizing Zlib's compression and digest functions [53], and OpenSSL's security modules [58]. The `scp` and `sftp` data transfer procedures involve four single-threaded processes: `scp` client for data reading, `ssh` for encryption and sending, `sshd` for receiving and decryption, and `scp` server for data writing. As depicted in Figure 3-1, the two processes

at each side use a kernel pipe for their inter-process communication. Each data transfer task uses a single TCP connection. Our preliminary experiments have shown that even with multiple scp instances, this software is not able to transfer data at the line speed of state-of-the-art hardware connections, such as 40Gbps RoCE (RDMA over Converged Ethernet) and 56 Gbps InfiniBand.

3.3 Memory-Centric Asynchronous Design

In this section, we present a two-level asynchronous design, ACES, which focuses on the memory-centric principle for high throughput computing. ACES treats user requirements as divisible into stages, such as storage I/O stage, network I/O stage, data integrity stage, etc, connected by explicit task transit queues. From an application's perspective, memory blocks are processed by multiple stages asynchronously. Within each stage ACES selects a suitable implementation method to maximize the corresponding hardware throughput and uses asynchronous I/O interfaces whenever possible. In addition, ACES manipulates, maintains, and transits memory blocks to avoid memory copy operations. Based on these design methods, we implemented ACES-FTP, an asynchronous file transfer application for high throughput systems.

3.3.1 Design Goals

The ACES framework and ACES-FTP software target various big data applications that require intensive network and storage I/O operations, as well as memory-resident data processing and transformation, such as data integrity and encryption/decryption. Our design goals include the following:

- **Efficiently utilize hardware advances**: To attain bare-metal throughput capacity of hardware, ACES-FTP makes use of the advanced features provided by hardware, such as zero-copy and parallel I/O execution. Specif-

ically, ACES-FTP leverages RDMA technology and its one-sided *RDMA Write* operation to achieve high network data transfer speeds, and makes use of asynchronous and direct I/O storage file operations to gain abundant storage throughput.

- **Mitigate I/O latency accumulation**: In general, there are two types of latency: I/O latency associated with peripherals and computation latency in yield from CPU intensive functions. A single-threaded synchronous application accumulates all computation and I/O latencies in a sequence of processing steps. This may lead to some peripheral devices waiting in idle state when an application interacts with a different peripheral device. Therefore, an application may suffer low throughput as a result of inefficient resource utilization. Although a multi-threaded application can avoid wasting I/O resources, it incurs the overheads for thread management and context switching if the application has to manage numerous resources, such as TCP connections. The two-level asynchronous design in ACES aims to overlap latencies among different I/O-intensive operations and computation-intensive ones without incurring thread context switches.

- **Flexible, reusable, and configurable stage design**: A stage in the ACES model is a reusable module that combines threads design and function integration. This is different from traditional software design which either delivers a user function binary or a packaged library of function calls. In ACES, we concentrate more on the throughput of each module, and try to use the best practice in each hardware facility. In addition, to satisfy user demands, the ACES functions should be configurable per task. For example, some local data transfer may not need security stage involvement.

- **Automatic balancing of stage threads**: This framework balances the number of threads in a stage automatically by monitoring the queue and

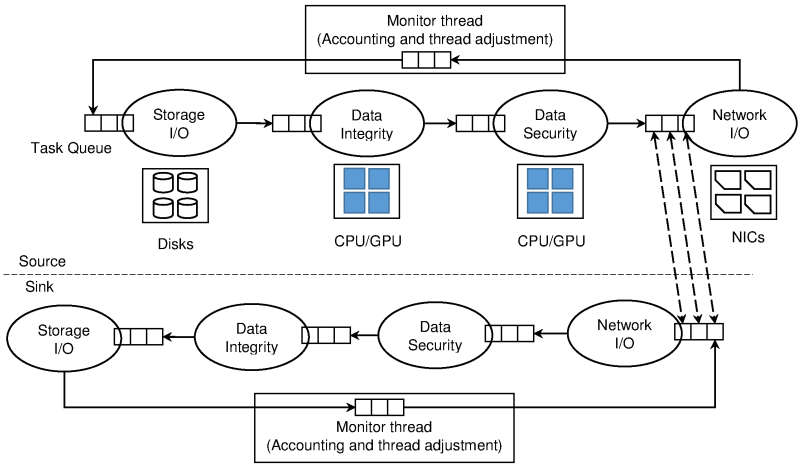


Figure 3-2: End-to-end asynchronous software architecture.

thread status. We propose a global monitor thread for status accounting and threads adjustment through queuing delay, thread waiting time, and task processing time. For this purpose, timestamps are marked by the framework when tasks are going through processing stages. This centralized monitor is different from SEDA distributed resource controllers deployed in each stage [59]. A new introduced stage could easily leverage on the existing timing based feedback and adjustment mechanism.

3.3.2 End-to-end Staged Asynchronous Software Architecture

An end-to-end data transfer path involves several stages, including storage stage, data integrity stage, data security stage, and network stage, in both the data source and data sink, as shown in Figure 3-2. ACES-FTP defines these stages and connects them using explicit task queues. Once a storage I/O stage pushes a task into the task queue of a data integrity stage, it uses a thread signal as a notification to trigger another thread in the data integrity stage to process the incoming task.

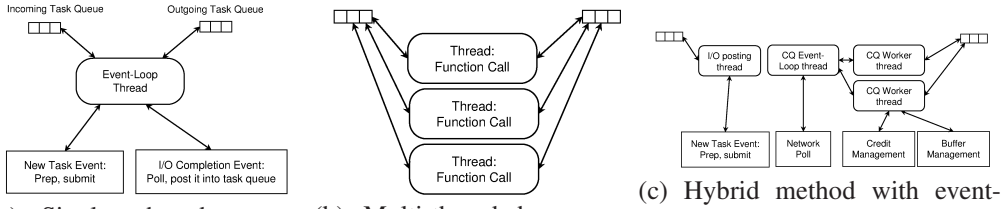
With the event-based design, the queue states are critical data accessed and updated by multiple threads. Their integrity and consistency are protected by

thread synchronization primitives: mutexes and condition variables. Multiple threads will simultaneously access large memory segments that host data to be processed and transported. However, we only need to protect event queues with exclusive access. Such a design minimizes the granularity of critical sections within a thread to access the protected data and avoids potential thread wait, thereby maximizing parallel operations.

Between the two sides (source and sink) of an event queue, we develop a flow control mechanism. The starting or ending point in each side is a queue of available memory blocks. A memory block has to be ready before each side starts a new task. The queue is also used for flow control, and it limits the throughput automatically if any stage incurs performance fluctuation. For example, when the sink side is experiencing a period of performance drop due to storage competition, it will stack memory in the task queue of the storage I/O stage. Consequently, the source side is not be able to get more credits from the sink side and will wait in the network stage.

In RDMA network stage, we use RDMA one-sided operation and its zero-copy capability to attain the best performance of state-of-the-art high performance network capability. At the same time, we use two-sided Send/Recv to exchange control messages, such as memory credits and completion notifications [22, 23]. Because RDMA one-sided operation is only supported by Reliable Connected (RC) communication, we selected RC for RDMA connection management.

To fully utilize the hardware advances and concurrent capacity of a system, multiple tasks associated with memory regions are processed in parallel by several facilities in different stages, such as disks, network interfaces, and CPUs (we regard CPU cores for function computation as a type of resources). We pass between stages tasks with memory data pointers to eliminate data copies between stages. However, different stages can manipulate the content of memory blocks according to the control header information.



(a) Single thread event-driven.

(b) Multi-threaded concurrency.

(c) Hybrid method with event-driven and multi-threaded.

Figure 3-3: Stage implementation methods.

3.3.3 Stage Implementation

To capture the characteristics of different hardwares (CPU, storage disk, and network adapter), we use three types of stage implementation in ACES-FTP. As shown in Figure 3-3, each stage could be implemented as one of three scenarios: (1) a single thread with an event-driven design (for disk I/O), (2) multiple parallel threads (for CPU processing), and, (3) multiple threads with different functions (for network I/O).

Asynchronous event-driven single-threaded stage. The disk I/O stage makes use of asynchronous I/O functions, and a single thread is able to saturate the entire disk I/O performance by submitting a batch of I/O requests to hardware disk, as shown in Figure 3-3(a). There are two types of asynchronous I/O interfaces in Linux: *POSIX AIO* and *Linux native AIO - libaio* [35]. These two AIO implementations are fundamentally different. The POSIX AIO implementation creates multiple threads within the OS, each of which performs normal blocking I/O. It wraps synchronous I/O functions to emulate the behavior of asynchronous I/Os. On the other side, libaio is truly asynchronous, and directly supported by the OS kernel which internally queues I/O requests and submits to devices in an optimized manner. We choose libaio in our implementation.

Parallel multi-threaded stage. In data integrity and encryption/decryption stages, we use multiple parallel threads to maximally utilize multiple cores, as shown in Figure 3-3(b). In the staged software design, we uniformly treat CPU cores as one regular type of hardware resource for data processing functions

(such as encryption, checksum, and compression), which altogether can process a batch of tasks asynchronously. The number of parallel threads is adjusted dynamically by a global monitor thread and is determined by system running status.

Asynchronous multi-threaded stage. Network I/O stage in ACES-FTP is more complex given the need for coordination between sender and receiver. ACES-FTP tries to obtain the line speed of state-of-the-art hardware by using one-sided RDMA data transport via verbs interfaces `libibverbs` and RDMA communication with the communication manager `librdmacm`. The RDMA-based network stage involves explicit user-level protocols, and ACES-FTP manages several resources, such as RDMA connections, memory credits, and asynchronous RDMA events. Therefore, we use a hybrid software module that combines event-driven and multi-threaded paradigm, as shown in Figure 3-3(c). First, ACES-FTP at the source side proactively requests memory credits prior to the incoming data transfer events. This asynchronous credit preparation overlaps the network latency on data transfer with that on credit exchanging. Second, ACES-FTP posts multiple RDMA one-sided operations into each of the reliable connections. It places a master-worker thread pool to handle completion events, including Send, Recv, and RDMA Write completions, polled from *completion queue* (CQ). This asynchronous, multi-threaded stage is different from the single threaded event-driven stage in which one thread handles the entire completed events: here, one top level data transfer request (application event) will incur a series of low level network RDMA events, and a CQ thread dispatches network events to a pool of worker threads to increase the concurrency in processing a large number of low level events.

3.3.4 Uncertainty and Determinism

Traditional sequential programs consider data as sequential streams. This deterministic sequence of data stream simplifies software implementation. Because data integrity and consistency in any stage are guaranteed by the sequential processing logic, there is no need to implement additional control logic to deal with the out-of-order issue. On the other hand, this approach suffers from hardware uncertainty on performance. For example, the throughput performance varies when different disk sectors and/or disk storage media (magnetic v.s. solid state) are involved or network is under congestion for a particular task, an individual processing step will hold up the entire processing pipeline without moving forward.

The two-level asynchronous software architecture increases the probability of data sequence uncertainty while alleviating the hazard of hardware uncertainty. In the disk I/O stage, although ACES-FTP submits I/O requests in a sequential manner, the completion sequence will not necessarily be identical to the submitted one. OS I/O scheduler may reorder the execution sequence of I/O requests. Second, the completion sequence might be affected by storage hardware uncertainty. The same behavior may occur in the network I/O stage because ACES-FTP uses multiple concurrent connections to avoid the bottleneck of a single connection. The out-of-order issue causes storage performance degradation in the data sink side. We implemented a pending queue for sequence reordering in the network I/O stage at the sink side to minimize the impact of this uncertainty.

3.3.5 Memory Centric Design and Memory State Transition

We design our software with a memory-centric perspective to maximize processing throughput while minimizing the overhead of memory, disk, and network operations. State-of-the-art hardware fosters the adoption of several zero-

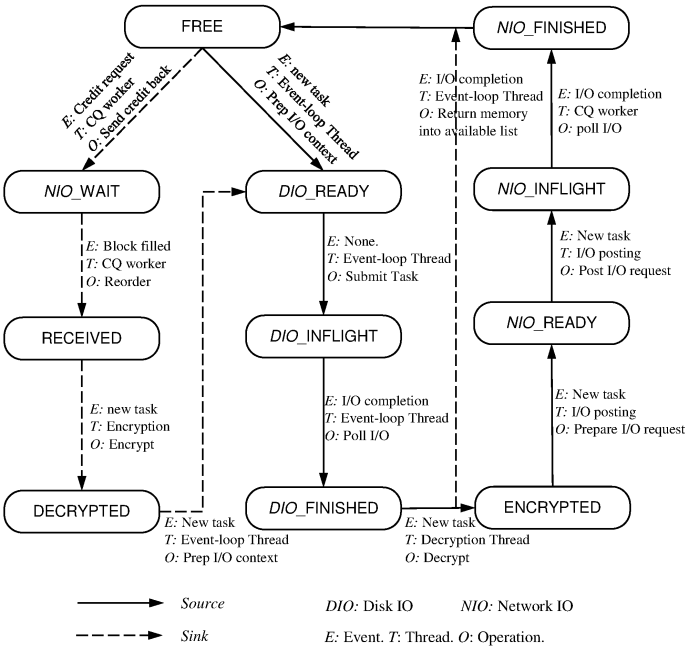


Figure 3-4: End-to-End memory status transition.

copy mechanisms. For instance, high performance network adapters bridge the performance gap between network and memory, and RDMA's advanced zero-copy feature eliminates the overhead of memory data copy between kernel and user space. For storage systems, direct I/O in file operation prevents data from going through the kernel cache, and enables applications, instead of the OS, to optimize data access, since applications have first-hand knowledge on what data are going to be accessed in the incoming operation steps. In addition, asynchronous I/O operations require software to track and associate memory states in different stages, then to perform corresponding behavior according to I/O context and triggering event.

In our design, we model memory state transitions using a finite state machine(FSM), as shown in Figure 3-4. In terms of each state transition, an *operator* (thread) performs a defined *operation* (method function) in responding to a related *event*, usually in an asynchronous fashion. For instance, the memory block at the sink side is in the "FREE" state in the beginning. Once the

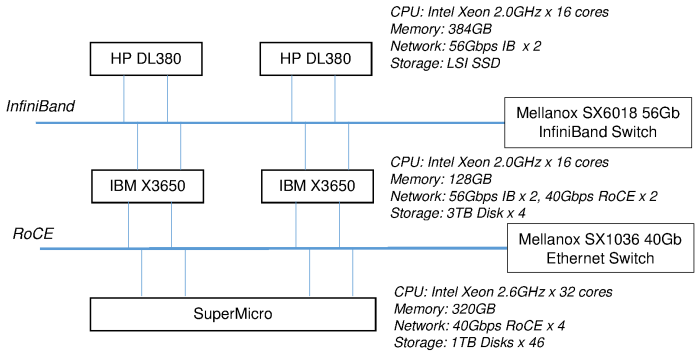


Figure 3-5: Testbed connectivity.

sink side gets a event on credit request (i.e., availability of a memory block), a *completion queue worker thread* sets a memory block into "NIO_WAIT" state, and sends back the credit of this particular memory block to the source. Each memory block has its own state fields, and this FSM guides application threads to parallelly perform on numerous memory blocks asynchronously.

3.4 Evaluation

We evaluate the integrated asynchronous computing and asynchronous I/O operations from high-level application layer. We use our developed ACES-FTP to transfer data in a high performance network environment, and compare its performance with that of the scp software which is commonly used to transfer sensitive data in an insecure network, for example, Internet.

3.4.1 Experimental Setup

Our testbed consists of five servers connected by both RoCE and InfiniBand links via a switch. Each host connected to the InfiniBand switch has two 56-Gbps Mellanox InfiniBand FDR adapters. Each IBM host has two 40-Gbps Mellanox RoCE QDR adapters, and the SuperMicro server has four network adapters of the same brand. Figure 3-5 shows the detailed host configurations.

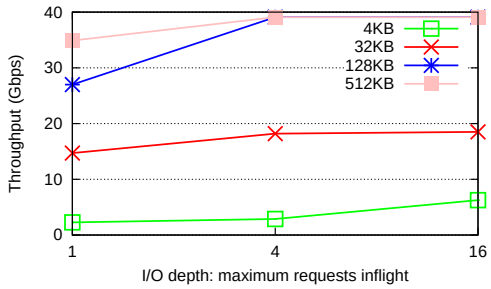
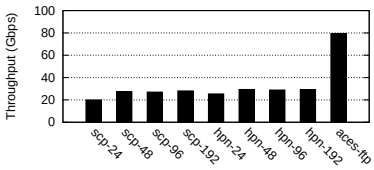


Figure 3-6: RDMA asynchronous performance in 40 Gbps Ethernet.

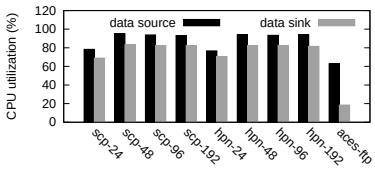
We enabled hyper-threading in all three hosts to maximize the processing capacity for encryption. We set up a Ganglia [60] cluster monitoring system in our testbed. The performance metrics, including both CPU usage and cluster throughput, are obtained by Ganglia comma-separated values (CSV) output. All disk I/O accesses use direct I/O to bypass kernel page cache.

3.4.2 RDMA Asynchronous I/O Evaluation

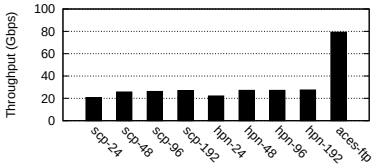
We developed an asynchronous RDMA benchmark, called iperf-rdma, based on iperf [61], a popular TCP/UDP bandwidth benchmark. The iperf-rdma supports multiple RDMA streams and multiple asynchronous outstanding requests in each stream. To show the impact of asynchronous I/O operation, we perform iperf-rdma in a mockup synchronous manner with only one I/O block in a connection and an asynchronous manner with multiple network I/O blocks pushed simultaneously into the sending queue. We deployed the iperf-rdma over a 40 Gbps RoCE link. As shown in Figure 3-6, RDMA performance is improved by posting multiple requests in the network link. Therefore, to attain bare-metal RDMA performance, an application should post multiple I/O tasks in flight to fully take advantage of RDMA hardware advances.



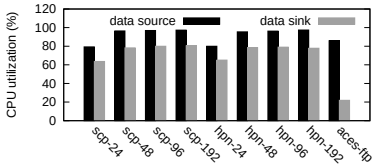
(a) Throughput comparison with AES 128bit CBC.



(b) CPU utilization comparison with AES 128bit CBC.



(c) Throughput comparison with AES 256bit CBC.



(d) CPU utilization comparison with AES 256bit CBC.

Figure 3-7: Performance comparison among OpenSSH-SCP, HPN-SSH, and ACES-FTP. The numbers in X-axis is the parallel processes launched during experiments.

3.4.3 ACES-FTP End-to-End Evaluation

We compare ACES-FTP to the popular openssh software suite and its high performance variant, High Performance SSH/SCP (HPN-SSH) [62]. HPN-SSH improves the performance of traditional ssh implementation by adding multi-threaded cipher, desynchronizing all I/O and cryptographic operations, and auto-tuning the TCP window sizes. In the storage I/O stage, we exported files from the memory based storage (RAM disk) at HP hosts to IBM hosts via InfiniBand links using iSER (iSCSI Extensions for RDMA) protocol to emulate high throughput storage devices. The detailed design of the Storage Area Network can be found at [23, 29]. Those applications transferred data via the RoCE links, and the capacity of network adapters is 80 Gbps with two 40 Gbps RoCE links, as shown in Figure 3-5. We launched multiple instances of scp to maximize the performance.

Figure 3-7 shows performance comparison among scp (OpenSSH), hpn-scp (HPN-SSH), and ACES-FTP. Both scp and hpn-scp are not able to saturate the 80 Gbps links due to high CPU consumption, including data copies between

user and kernel spaces, TCP stack processing(especially on the receiving end), and process scheduling. In contrast, ACES-FTP saturated this link with lower CPU consumption. First, ACES-FTP uses asynchronous RDMA operations to offload network operations into hardware. Second, ACES-FTP integrated direct I/O to bypass kernel cache and to load data directly from hardware into user space memory. Third, ACES asynchronous operations overlap the latencies in disk I/O, encryption processing, and network I/O.

Magnetic disks are cheaper, non-volatile compared to RAM disk and SSD drives and are still of primary use in many HTC applications and databases. This experiment targets a common storage environment and validates our asynchronous design. Thereby, we set up the SuperMicro 46 magnetic disks as a data sink to receive data from two IBM hosts which load data from iSER memory disks. The `fio` benchmark revealed the best throughput of those 46 disks is 7.5 GByte/s (sustained). Our experiments show that the best performance with ACES-FTP is 6 GByte/s while scp can achieve 3 GB/s at best. The small difference between the ideal case of the `fio` benchmark and real life end-to-end data transfer can be attributed to high CPU loads incurred by both file system operation and computation-intensive data encryption.

3.5 Conclusion

This chapter details ACES, a memory-centric software framework for high-throughput computing within a single server, and ACES-FTP, a reference implementation of a high-performance secure data transfer application on top of ACES. ACES-FTP relies on and integrates several types of asynchronous processing mechanisms, including (1) asynchronous direct I/O operations for disk access, (2) asynchronous RDMA operations for ultra high-speed networks, and, (3) asynchronous processing of computation-intensive data integrity and encryption using dozens of Intel CPU cores. In an end-to-end data transfer test

over high performance networks, ACES-FTP shows a throughput three times as high as that of scp with the same level of security enforcement. Furthermore, it achieves a remarkable performance gain, i.e., 80 Gbps (line speed) secure data transfer throughput with off-the-shelf computer hardwares. In summary, ACES-FTP provides an efficient approach for large-scale, secure data replication within data center networks and between data centers over the public Internet.

Chapter 4

Scalable RDMA-based Data Transfer Protocol

As data center network performance increases significantly under RDMA's support, an RDMA-based data transfer protocol enables applications to more efficiently utilize the hardware advances such as zero-copy and kernel bypass. To that end, we propose a scalable networks protocol based on RDMA primitives. Scalability in RDMA-based protocol design is twofold: First, the protocol is required to scale applications' network performance to ultra-high-speed networks, such as 100 Gbps and beyond; Second, the protocol is able to scale to wide area networks with extreme long latency, which means large bandwidth delay product and huge numbers of packet in flight that are managed by applications instead of kernel services.

In this chapter, we propose a scalable RDMA-based data transfer protocol. We first present a protocol overview. Then, we use a finite state machine to model the memory status transition in with zero-copy constraint. We also give out message format and multi-stream connection management. We construct an end-to-end data transfer software, RFTP, on top of this protocol. We present the comparison of RFTP and popular data transfer tools, such as GridFTP, in various testbed environment.

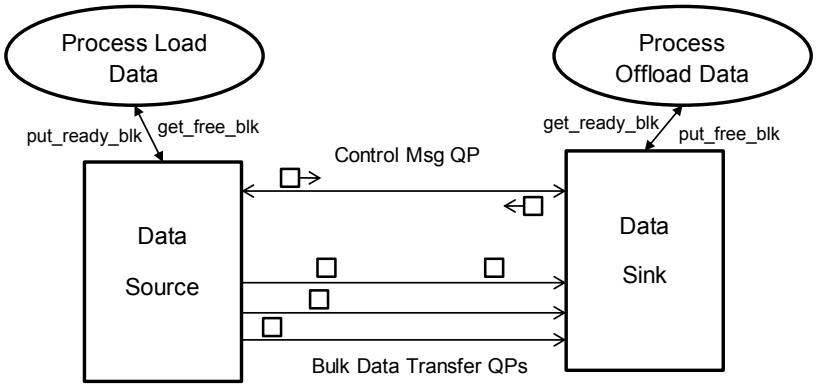


Figure 4-1: RDMA-based Data Transfer Protocol Overview.

4.1 Protocol Overview

The OFED standard supports two types of queue pairs for host-to-host communication: Reliable Connected (RC) and Unreliable Datagram (UD). Considering the requirements of performance *and* reliability, we selected RC queue pairs for our protocol. The application can divide the entire dataset to be transferred into large blocks, a feature that usually leads to low processing overhead. On the other hand, the UD type is supported only in channel semantics, and the block size is limited by the size of the MTU [63]. A small block size may incur high CPU consumption, since many small blocks trigger a large number of queue pair events and interrupts that must be handled at both the data source and sink. In our protocol, we use one dedicated queue pair for exchanging control messages between two communicating parties, and one or more for actual data transfer. Figure 4-1 illustrates how this protocol works. We use an event-driven design where different types of control message or regular data blocks trigger different events to be handled by pre-defined event routines.

To fully utilize the RDMA technology, our protocol design incorporates several optimizations. Firstly, the protocol keeps multiple data blocks in flight during the entire data transfer period. As we mentioned in the previous section, a high queue depth with several data blocks in flight is the key to achieving good

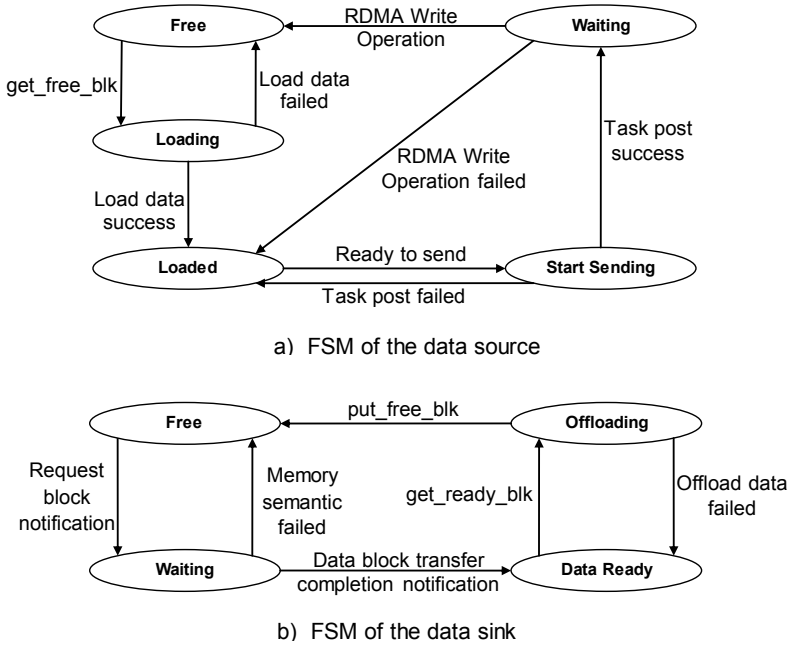


Figure 4-2: User payload block's finite state machine.

performance. Secondly, the protocol is capable of using parallel queue pairs to transfer multiple data blocks simultaneously, eliminating the performance limitation of a single queue pair. With multiple queue pairs, there is the possibility of out-of-order arrivals of data blocks at the data sink. The protocol implementation must therefore be able to reassemble such out-of-order blocks. Thirdly, since the protocol uses RDMA WRITE to deliver bulk user payload, credits (tokens with destination address) are required before transmitting the data. It takes one additional round trip time (RTT) if the source explicitly requests credit information from the data sink. To save this RTT, our protocol adopts an active feedback mechanism. The data sink will proactively send the available data block information (credits) to the data source, and the data source keep track of all available ones.

4.2 Finite State Machines Modeling

To better illustrate our protocol, we used a finite state machine to model buffer blocks and their status at both the data source and sink. In our data transfer protocol, unlike TCP sockets, the sender does not explicitly copy data from user space to kernel space. Instead, the sender only posts tasks via the OFED interface, and afterwards the network card directly retrieves data from user space. With this model, the finite state machine of buffer blocks explains our protocol's behavior. In the data source, a block (a chunk of memory resource for storing data) is initialized into a "free" state. A data transfer application can reserve a free block by `get_free_blk` which changes the state of the reserved block from "free" to "loading". The application then loads data from disk directly to the memory block, and the state then changes from "loading" to "loaded". Before the actual data transfer, the data source needs to know the remote memory's information, such as the unique identifier (rkey) and memory address of the data sink. Afterwards, the source encapsulates the block information into a memory semantic task, and posts it into the send queue. The state then changes from "Start sending" to "Waiting" if the task is posted successfully. "Waiting" means the content of the memory block is in flight. After the application polls the status of the memory semantic operation, the state is changed to "free" again if successful or "loaded" for re-sending if polling fails.

A block's state in the data sink finite state machine changes from free into waiting once either of the following two kinds of event is retrieved. One is a block request notification, which means the data source runs out of credits and is eager to get more credits as soon as possible. The other possible event is a completion notification of another memory block, which implies that the data source consumes one credit for that block. For efficient data transfer, a proactive feedback mechanism sends back one or two credits immediately to avoid the source running out of credit. A finish notification related to this block changes

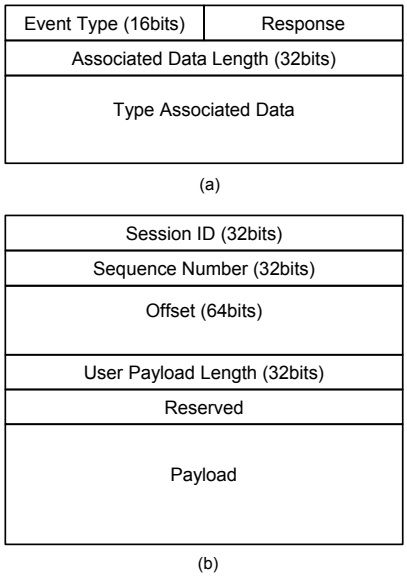


Figure 4-3: Message Format of (a) Control message, and (b) User Payload Bulk Data Block.

its state into "data ready". The application retrieves a block's payload from the protocol layer by `get_ready_blk`. After the application consumes the block's payload, i.e. offloading data into file system, the block's state is changed into "free" again by `put_free_blk`.

4.3 Connection Management and Message Format

As described in the previous subsection, the data source and sink manage buffer blocks and transfer data using asynchronous RDMA operations. Next, we detail connection management and message types during the process of moving data. Each instance of data movement consists of three phases: (1) Initialization and parameter negotiation; (2) data transfer and reordering, and (3) connection teardown. Figure 4-3(a) shows the format of the control message exchanged through the dedicated control message queue pair, and Figure 4-3(b) shows the format of the user payload data delivered through multiple data channel queue pairs.

In the first phase before data transfer, the data source sends requests to the sink to negotiate the block size, number of data channel queue pairs, and session identifier for each data transfer job.

- **Block size negotiation:** The data source selects a block size based on the user's input parameters, and copies the size information to the field of "Type Associated Data" of the control message to be sent to the sink. The sink sends back a reply on whether or not it accepts the block size for data transfer.

- **Number of data channels negotiation:** The protocol is designed to support multiple data channels, even when only transferring a single file. The source and the sink will exchange messages to agree on and establish a user-defined number of parallel queue pairs to deliver payload data.

- **Session identifier negotiation:** Each data transfer job, such as one file, is assigned a unique session identifier before the data is transferred. This identifier is placed into the header of every user payload block during the transfer of data. The application probably issues multiple data transfer tasks simultaneously. Each task is associated with a global session identifier which is available in both the source and sink. The sink is able to reassemble out-of-order blocks and deliver an in-order sequence of blocks to upper applications according to the session identifier and sequence number.

Our protocol supports asynchronous data transfer using OFED, viz., the key to enabling higher performance over the traditional TCP-based approaches. The source posts multiple payload data blocks in flight, and the sink actively acknowledges the successful receipt of data and returns the available memory region for the subsequent data transfer. There are three types of control message in this phase.

- **Memory Region (MR) block information request:** Once there is no available remote memory region for storing data before transferring, the data source sends this message to the data sink to request the next available memory region. The source is blocked until the sink sends back a response with MR information.

- **Block transfer completion notification:** The source sends a completion notification to notify a data sink that a data block is finished and available for the sink to read. This notification includes the block's ID and address, allowing the data sink to extract the payload from the memory block.

- **Memory region block information response:** The previous two types of control messages from the data source trigger the sink to send back any available memory region information. If the sink gets a **memory region block information request**, this indicates the source is idle and waiting for credits to proceed. The sink sends back one or multiple available addresses information according to the runtime status of the data transfer. If the sink gets a **block transfer completion notification**, the source must consume an available data address, and the sink grants back, at most, information on two available memory regions. This results in an exponential increase in the number of available remote MR in the data source at the beginning of a data transfer session. Such a design is similar to the slow start of TCP which allows the data transfer protocol to quickly fill up the available bandwidth. If at that time there is no available memory region in the data sink, the completion notification is simply ignored and the sink does not have to send a response. However, for the **memory region block information request**, the sink must send a response once there is at least one available memory regions. Otherwise, the responder will be delayed until one becomes available.

Finally, in the teardown phase, the source issues a **data set transfer com-**

pletion message indicating that the whole data set was transferred completely to the sink.

4.4 Discussion on Scalability

We design the RDMA-based data transfer protocol with the goals to scale the applications' performance to high performance networks with high bandwidth and long latency. In this section we discuss the scalability of our protocol design.

4.4.1 Scalability to Next Generation High Speed Networks

As the increasing network bandwidth in local area networks, end hosts require more computation resources on data processing and data copy in tradition TCP based approach. Because TCP/IP based applications rely on kernel level service as a relay agent, they involves significant amount of data copy, interrupt processing, and protocol processing. Our RDMA-based solution takes advantage of kernel bypass and zero-copy advances to eliminate the data copy overhead. This enables the protocol to be scalable to high throughput networks with 100 Gbps bandwidth or even higher.

4.4.2 Scalability to Wide Area Networks

In wide area networks, we focus on the scalability with long latency. RDMA data transfer relies on credit-based mechanism, and it takes one round trip time (RTT) to get such credits. To get over the impact of long RTT, we proposed a proactively feedback mechanism and pipelined implementation to overlap data transfer phase and credit management one. As a result, our protocol is able to be scalable to wide area networks with long latency.

4.5 Evaluation

To validate our protocol and its reference implementation, RFTP (RDMA-enabled FTP), we conducted comprehensive experimental studies on several LAN and WAN test environments. We begin this section describing the test configuration based on various RDMA architectures, including RoCE and InfiniBand, in LAN and WAN network environments. We then compare the performance of RFTP with GridFTP, a high performance data transfer tool widely used in the data-intensive science applications.

4.5.1 Testbed Setup

We consider both memory-to-memory and memory-to-disk data transfer between local and remote hosts. For the former, memory data in the source is generated from /dev/zero, transferred via RDMA, and copied into /dev/null at the sink. In this configuration, our focus is to evaluate the performance in terms of network bandwidth and the efficacy of protocol offloading. We did not access the performance of a test scenario with a file system considered since it is much slowed than our 40 Gbps network testbed. For modern data center applications, as suggested in [64], it is a reasonable simplification to avoid the disk I/O bottleneck. We consider a variety of network environments include the LAN (which plays a key role in today's data center and cloud computing applications) and the WAN (which is essential to inter-data center transfers and to upload or download data to remote clients). The details of our three configurations are as follows.

To test application performance over different RDMA architectures, we set up two local-area test platforms. The first one is a back-to-back connection testbed in Stony Brook University. The propagation delay between hosts is less than 0.1ms. Each host is equipped with a 40Gbps RoCE HCA. The second test platform includes two nodes at the NERSC Computational Center. Each

Table 4.1: RFTP testbed description

	InfniBand LAN	RoCE LAN	RoCE WAN
CPU * Cores	Intel Xeon X5550 2.67GHz 8 Cores	Intel Xeon X5650 2.67GHz 12 Cores	ANL: AMD Opteron Processor 6140 2.6GHz 16 Cores NERSC: Intel Xeon E5530 2.40GHz 8 Cores
Mem(GBytes)	48	24	ANL: 64 NERSC: 24
NICs(Gbps)	40	40	10
OS	RHEL 5.5	CentOS 6.2	ANL: CentOS 5.7 NERSC: CentOS 6.2
Kernel Version	2.6.18-238	2.6.32-220	ANL: 2.6.32-220 NERSC: 2.6.32.27
OFED Version	1.5.3.1	MLNX OFED 1.5.3	1.5.3
TCP Congestion Control Algorithm	cubic	bic	ANL: cubic NERSC:htcp
MTU Size	65520	9000	9000
RTT(ms)	0.013	0.025	49

node has a Mellanox InfiniBand HCA interconnected by a 4X QDR InfiniBand switch, theoretically providing 32 Gb/s of point-to-point bandwidth. The vendor reported that the actual bandwidth is about 25 Gbps during their product validation.

High-bandwidth long-latency WAN RoCE Testbed

For the WAN test with long-latency links, we used the Advanced Networking Initiative (ANI) 100Gbps testbed[1] between Argonne National Laboratory near Chicago, IL, and the National Energy Research Scientific Computing Center (NERSC) in Oakland, CA, about 2000 miles away. The hosts on the ANI Testbed are equipped with a 10 Gbps RoCE NIC.

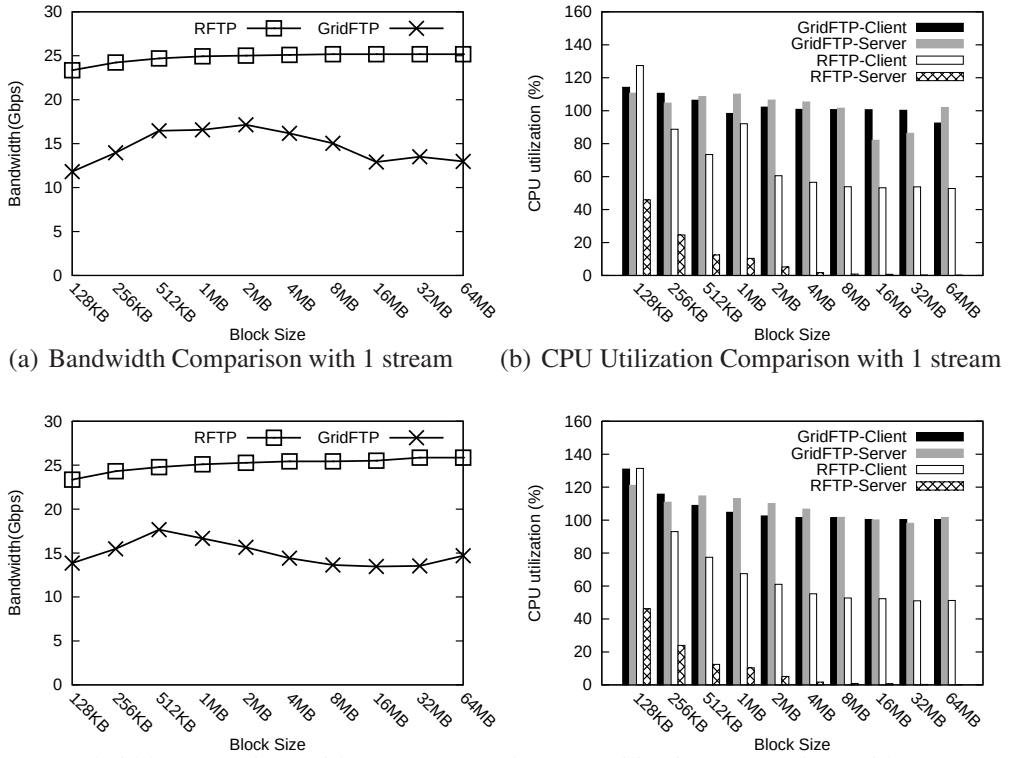
[1] ANI Testbed: http://ani-testbed.lbl.gov/

4.5.2 Parameter Configuration and Tuning

For a fair application comparison of these applications, we ran our test cases of RFTP and GridFTP on the same set of well-tuned hosts, and in a common network environment. Table 4.1 lists the detailed configurations for all the nodes, in all three testbeds described. To improve the performance of TCP for transferring bulk data, we tuned the parameters of O.S. kernel, NIC and the host's power setup according to the vendor supplied manual [65]. For certain hosts in the testbeds, we employed some variants of TCP algorithms. But we always evaluate RFTP and GridFTP with the same TCP variants. The size of MTU was set to 9000 bytes on all hosts. We also have optimized the configuration of GridFTP to ensure that it reached the best performance for network link with a large bandwidth-delay product (BDP). The GridFTP client, the globus-url-copy with extended block mode (MODE E) [66] was utilized for all data transfer; authentication was intentionally turned off to minimize the extra cost for data security. Both the GridFTP client and server here are threaded [67]. The size of TCP buffer is set to be the BDP of the link, a proven value for the optimal network performance. An important characteristic for GridFTP and RFTP is that they can both transfer a large file via multiple streams. Since there is no disk bottleneck in the memory-to-memory test, we transferred one file in each test case to assess the impact of the number of parallel streams. For the memory-to-disk test, we created a group of 400GB files spread across multiple RAID disks to achieve the best performance of the disk system.

4.5.3 Experimental Results over LAN

In this set of experiments, we used memory-to-memory data transfer as the baseline results to compare the performances of RFTP and GridFTP.



(a) Bandwidth Comparison with 1 stream (b) CPU Utilization Comparison with 1 stream

(c) Bandwidth Comparison with 8 streams (d) CPU Utilization Comparison with 8 streams

Figure 4-4: Bandwidth and CPU Utilization comparison between GridFTP and RFTP over RoCE in LAN.

Bandwidth and CPU usage comparison over the RoCE link

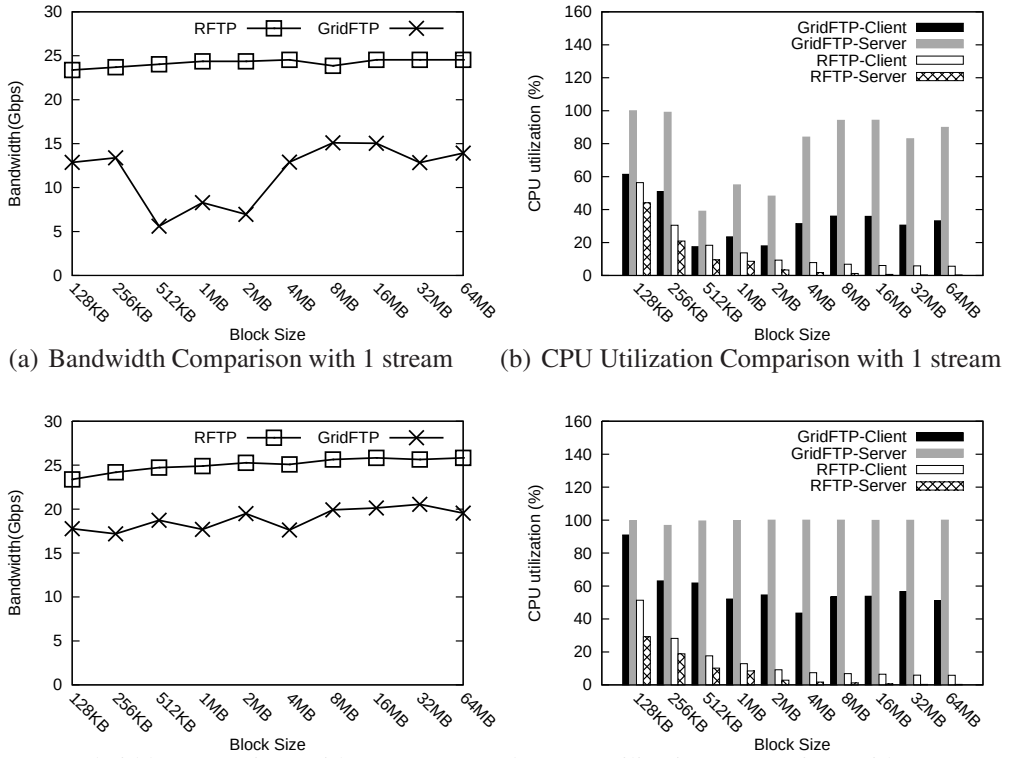
We consider the aggregate application bandwidth and CPU utilization as the primary performance metrics. The performance numbers obtained are as follows. For each test, we transferred 900GB data with both GridFTP and RFTP. The aggregate bandwidth was obtained by collecting the average transfer performance of all streams. To calculate the CPU usage, we employed the "nmon" [68] tool to record the CPU utilization of the application during the entire transfer period, and then determined the average usage. We note that if the host has 12 cores, the total CPU utilization can be up to $12 \times 100\%$.

Figure 4-4 shows the bandwidth and CPU utilization performance of GridFTP and RFTP over RoCE in LAN, with different block sizes and numbers of streams. We made the following observations:

- RFTP saturates the bare-metal bandwidth with different block sizes while CPU utilization declines as the block size increases. Block sizes play an important role in reducing the CPU load, since the number of control messages and CPU interruptions are fewer with larger blocks.

- Although the data transfer application can load data from /dev/zero with a high throughput it generates excessive CPU load to reset the memory content with `0x00`s. We monitored the CPU usage of the data loading thread using the "top" tool, finding that loading data from /dev/zero at 25Gbps leads to a 50% utilization of one core. According to Amdahl's law, the improvement to CPU utilization will be limited if loading data consumes a dominant share of the application's CPU usage. This is the case when the block size exceeds a certain threshold; for example, CPU utilization does not improve further when the block size is increased from 4MB to 64MB.

- A single GridFTP runtime process cannot archive bare-metal bandwidth, even with multiple streams or large block sizes. After we used the application debug tool "strace" to capture the underlying software behavior of the GridFTP application, we found that GridFTP only used a single thread to handle regular file operations, such as reading and writing data, and also network events, such as multiplexing, sending and receiving data. Consequently, good performance was not achieved once a single CPU became the bottleneck. As shown in Figure 4-4, both the GridFTP client and server always consume more than 100% of the CPU resource in a high bandwidth network environment. Furthermore, GridFTP's performance will be limited by a single core, while RFTP can take advantage of multi-core combined with multi-thread architecture simultaneously to handle more network events for a better transfer performance.



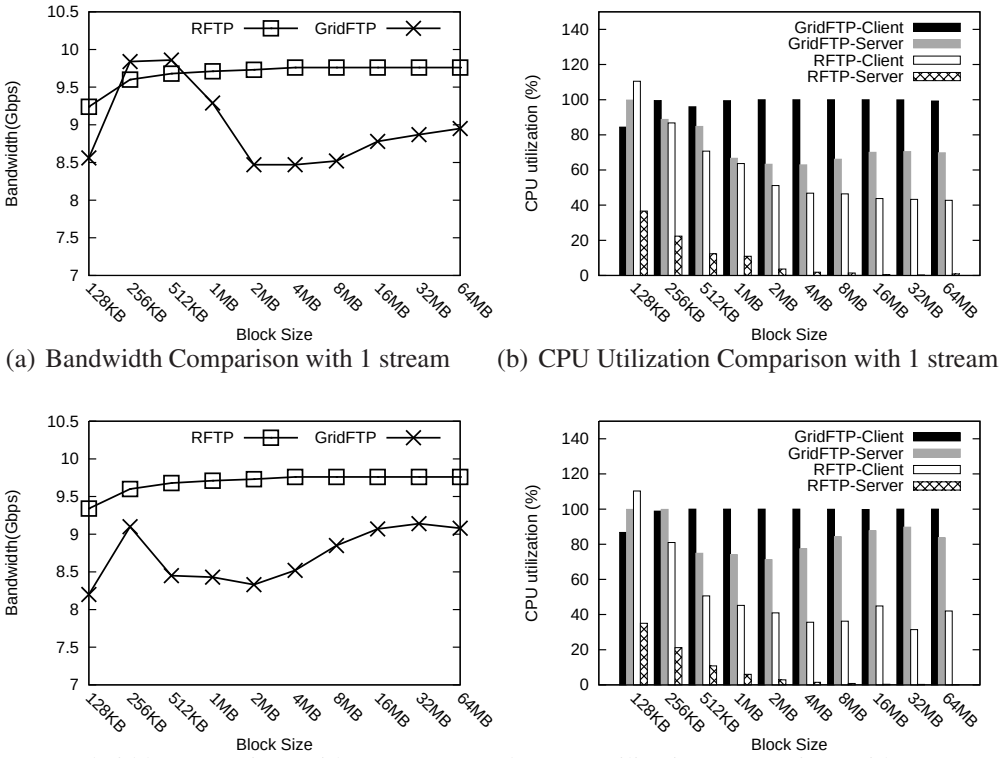
(a) Bandwidth Comparison with 1 stream (b) CPU Utilization Comparison with 1 stream

(c) Bandwidth Comparison with 8 streams (d) CPU Utilization Comparison with 8 streams

Figure 4-5: Bandwidth and CPU Utilization comparison between GridFTP and RFTP over InfiniBand in LAN.

Comparison of Bandwidth and CPU usage with the InfiniBand link

Figure 4-5 compares the bandwidth and CPU utilization between GridFTP and RFTP in the LAN environment with a 40Gbps InfiniBand link. We ran RFTP with one stream and eight streams. We also tested GridFTP with a single TCP connection and eight parallel connections. RFTP consistently outperforms GridFTP and attains high bandwidth in this setting. We also note that with RFTP, the bare-metal bandwidth is almost fully utilized when block size is sufficiently large, for example, 512K bytes. The bare-metal bandwidth is limited by the eight-lane PCI 2.0 (Peripheral Component Interconnect) network adapter. The observations in the previous section also are applicable in the InfiniBand environment. In addition, we made two more observations. First, compared with the results from the RoCE environment, the RFTP consumes less CPU in the



(a) Bandwidth Comparison with 1 stream (b) CPU Utilization Comparison with 1 stream

(c) Bandwidth Comparison with 8 streams (d) CPU Utilization Comparison with 8 streams

Figure 4-6: Bandwidth and CPU comparison between GridFTP and RFTP over RoCE in WAN.

InfiniBand environment. The reason is that *libibverbs* has lower overhead in the latter environment than that in the former one. Second, GridFTP's bandwidth performance fluctuates at different block sizes. This instability again might reflect GridFTP's single thread, and CPU power must be split between loading file data and network operations.

4.5.4 Experimental Results over WAN

We ran RFTP and GridFTP over the long-haul WAN RoCE link in the ANI testbed (the DOE's Advanced Network Initiative). In this set of experiments, we used both memory-to-memory and memory-to-disk data transfer to demonstrate the efficacy of our protocol design. Figure 4-6 compared the bandwidth and CPU utilization with one stream and eight streams. In most cases, RFTP again

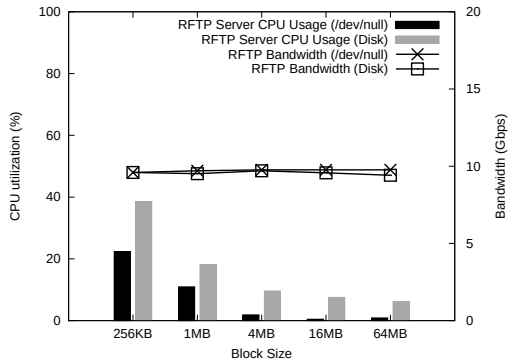


Figure 4-7: RFTP Bandwidth and CPU utilization comparison between Memory-to-Memory and Memory-to-Disk.

outperforms GridFTP in getting full bare-metal bandwidth with lower CPU utilization. The reason for bandwidth fluctuation of GridFTP is the same as we discussed in the previous subsection.

Figure 4-7 shows the bandwidth and CPU utilization of the RFTP server in the memory-to-memory and memory-to-disk test cases. We enabled the direct I/O feature of RFTP to save CPU usage and accelerate the RAID disk performance. To the best of our knowledge, GridFTP has not yet integrated direct I/O. Since writing data to disks with standard POSIX I/O consumes much more CPU time than direct I/O, GridFTP's performance is not comparable with RFTP using direct I/O. This figure shows that RFTP maintains the same bandwidth performance between memory and disk tests, with slightly higher CPU usage at the RFTP server since moving data into disk is more CPU intensive than simply writing into /dev/null. Hence, the design of our protocol and application are flexible in various testbed environments, including with disk operations.

4.6 Summary

RDMA is known as a promising high-performance protocol offload technique that supports zero-copy and kernel bypass. Several factors limit the use of RDMA techniques, including the lack of middleware support to RDMA hard-

ware and the lack of efficient protocols to fully utilize the available network bandwidth. In this chapter, we described our study of the design and performance issues of data transfer tools for high-speed networks such as 40 Gbps Ethernet and InfiniBand. Our work provides an RDMA-based middleware layer that provides simple resource abstraction and management, task scheduling, and parallel data transfer. Based on this middleware, we designed a data transfer protocol that supports high performance flow control and parallel data transfer.

To demonstrate the efficiency of our protocol and software design, we developed a reference implementation for the proposed FTP protocol. We set up testbeds with various RDMA technologies in various network environments to cover many different real-life data transfer scenarios. In particular, we demonstrated the performance of our protocol over the Department of Energy's ANI Testbed that includes multiple 10Gbps RoCE links over a 2000 mile path. The experiments show that our protocol and its intelligent design achieved remarkable bandwidth performance and fully maximized the RDMA hardware capacities.

Chapter 5

NUMA-Aware Cache for Storage Area Networks

Data centers often use non-uniform memory access (NUMA) technology to increase computation density per host and construct high-speed networks to offer low latency and high throughput for distributed applications. The asymmetric memory topology in NUMA systems results in a performance disparity in accessing different memory banks, and this disparity grows larger with an increasing number of CPUs per system, as shown in Figure 5-1. Meanwhile, two types of modern networks, modern intra data center high-speed interconnects between storage nodes and computer serves and communication networks including LAN and WAN, had significant technology improvements in the past decade, and shortened the throughput gap between memory and network accesses. Under several circumstances, the interconnect's performance even surpassed that of the inefficient NUMA-agnostic memory accesses, and thereby rendered memory access a new bottleneck along the end-to-end I/O path. For example, the current memory caching design in storage servers is affected by this problem and restricts the performance of end-to-end data access path.

In this chapter, we first analyze the I/O cost of iSCSI systems. Then we elaborate the design and implementation of a NUMA-aware cache for iSCSI

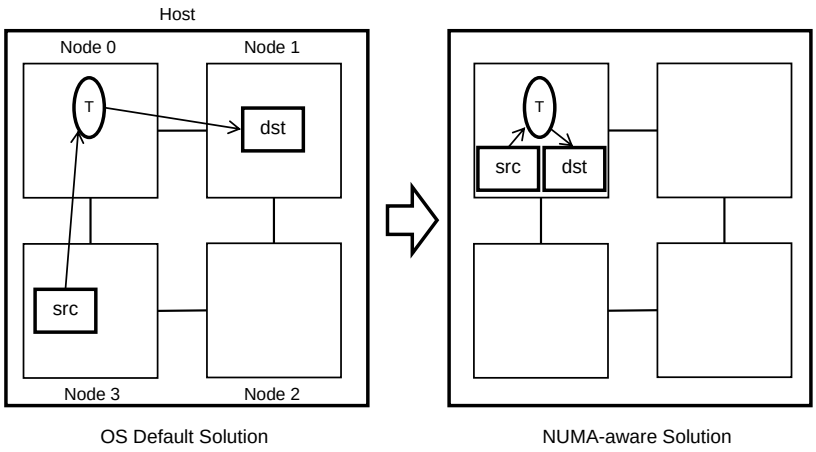


Figure 5-1: Memory copy routine on a four-node NUMA host.

systems. We further enhance the parallel capability to process the requests by introducing the decentralized event processing model to storage server. We also evaluate our design with both synthetic and real-life workloads.

5.1 I/O Cost Analysis with iSCSI

To illustrate how NUMA-aware caching improves the performance of SAN systems, we analyze the I/O cost associated with the iSCSI and iSER protocols. We focus on two performance metrics in serving cached data in the iSCSI/iSER systems. One is the response time (or processing time) of an individual request. It is important for real-time applications to get prompt response from backend storage systems. The other one is aggregated iSCSI system throughput which is critical for data-intensive applications to get high bandwidth.

5.1.1 Processing Time and Throughput Modeling

Let us first consider the processing time of an individual I/O request. Processing time measures the total time between when the I/O request is sent (i.e., passed to the network layer of an initiator) and when the last byte of the response data

is received. For a read request resulting in a cache hit, the processing time is approximately the sum of four latencies shown by Equation (5.1): 1) the propagation delay of the request and response that is equivalent to one round trip time (RTT); 2) memory access latency to copy data from either the NUMA-aware cache or OS page cache to the target's network buffer, which can be calculated as the size of data divided by memory bandwidth; 3) network transmission latency; and 4) queuing delay.

$$\begin{aligned} ProcessingTime = RTT &+ \frac{I/OSize}{MemoryBandwidth} \\ &+ \frac{I/OSize}{NetworkBandwidth} \\ &+ QueuingDelay \end{aligned} \tag{5.1}$$

In addition, we formulate aggregated throughput performance when the system is under the stress test with data-intensive applications. In this case, both system memory and network can be bottlenecks. Consequently, the overall throughput could be defined as $\min(MemoryBandwidth, NetworkBandwidth)$.

5.1.2 The Impact of Queuing Delay

The queuing delay is an important element in heavy-loaded systems. First, the queuing delay contributes to the processing time of each request: The higher the IOPS (IO requests per second) is, the more requests are queued up and the longer queuing delay contributes to the entire processing time. Second, the total bandwidth is determined by the bottleneck section along the data path. For example, even though there are many requests in queue before they are responded and served, the total bandwidth is still determined by network adapter's capacity if network is the bottleneck. In this case, the queuing delay is a result of the bottleneck section.

5.1.3 Cost Analysis with Our Testbed System

In storage area networks, previous tuning approaches focused on network systems, as network was the dominant bottleneck for both latency and throughput. For example, memory bandwidth, usually 1 GB/s to 20 GB/s [69], was a factor of ten or more faster than the network transmission speed of early gigabit networks; memory access latency was measured in the scale of microsecond, while network was measured in milliseconds. However, 40 Gbps quad data rate (QDR), 56 Gbps fourteen data rate (FDR), and 100 Gbps enhanced data rate (EDR) InfiniBand and converged Ethernet bridge the performance gap between network and memory. The increase in network bandwidth greatly reduces the transmission latency within LAN, and therefore, the latency and bandwidth of memory copy become significant and can no longer be ignored. Furthermore, the performance ratios of bandwidth and latency between local and remote memory accesses vary from 1.5 to 5.5 within a 4-node NUMA system [70]. Therefore, memory access likely can become a new bottleneck in modern NUMA hosts if not carefully optimized.

We look into the memory portion of Equation (5.1), and derive the average memory-related latencies and bandwidth for applications with page cache or NUMA-aware cache, respectively. Due to the imbalance in memory access in a NUMA system, we define the average memory bandwidth of an application as a weighted sum of all nodes' memory bandwidth, as shown in Equation (5.2). In the performance model of a page cache in a NUMA system, the source memory location of cached data and the destination (network buffer) location are uniformly distributed across all NUMA nodes. Meanwhile, the performing thread is not aware of those memory locations. Therefore, the *Weight* of each node is equal to any other node: $1/n$, e.g., 25% for each node in a 4-node NUMA system. In contrast, our NUMA-aware solution aims to increase the *Weight* of local memory access and decrease that of remote memory access. The perfor-

mance gain on memory access in the NUMA-aware solution is determined by the performance disparity between local and remote access and the increment on the *Weight* of local memory.

$$MemBW_{CachedData} = \sum_{i=0}^{n} Weight_i * MemBW_i \tag{5.2}$$

We did preliminary experiments in a 4-node NUMA system to validate the aforementioned analysis. The local memory bandwidth is about 18.9 GB/s, while the memory bandwidths for three remote nodes are about 3.3 GB/s, 2.9 GB/s, and 3.3 GB/s, respectively. The detailed performance analysis can be found in [27, 30]. According to Equation (5.2), the memory bandwidth for an application using the default page cache is about 7.1 GB/s. If the NUMA-aware solution achieves 90% access to be local, i.e. *Weight* of local access = 90%, the memory bandwidth for an application is about 17.32 GB/s. Furthermore, NUMA-aware cache may reduce the processing time of each individual I/O request by 20% as compared to the default page cache. We present and discuss the synthetic and real-life experimental performance results in Section 5.4 and Section 5.5.

5.2 NUMA-aware Cache Design and Implementation

In this section, we describe our NUMA-aware cache solution, including cache memory organization, I/O request scheduling methods, and I/O interpreting function.

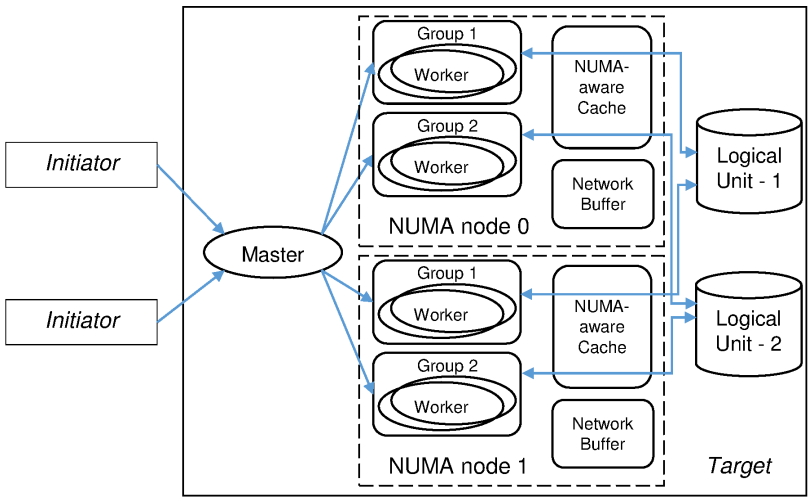


Figure 5-2: Construct a NUMA-aware solution for the master-worker threads model.

5.2.1 Software Overview

We design and implement NUMA-aware cache based on the latest version of Linux SCSI Target Framework. The existing framework employs a master-worker model to achieve parallelism in serving concurrent I/O requests in a multi-core system. A daemon process of the framework, named *tgtd*, manages multiple logical units in iSCSI, and exports each logical unit as a disk drive to the corresponding front-end initiator(s). Each logical unit is assigned to a group of dedicated worker threads. Here we use an example of serving an I/O request to illustrate the mechanism of the master-worker thread model in the existing framework. The master thread manages multiple network connections from several initiators. It receives an iSCSI command from one network connection, identifies the corresponding logical unit number (LUN), and then dispatches the command to the iSCSI command list associated with the requested LUN. A worker thread for the corresponding LUN retrieves the I/O request from the command list, and initiates a read or write operation. In the event of a cache hit, the worker thread copies data from the OS page cache into an available network buffer, and subsequently sends it to the initiator.

To construct a NUMA-aware cache for the framework, we consider the following requirements:

- **Performance scalability.** The cached data of a logical unit is evenly distributed across all the NUMA nodes to scale performance and balance system loads. Meanwhile, the overhead of managing cache should not increase with the number of NUMA nodes on a large multi-core server.

- **NUMA-awareness on cached data.** The memory location of target data can be calculated for an incoming request that happens to be cache-hit, i.e., with its starting address and data length.

- **NUMA-awareness on worker threads.** Because of the **performance scalability** requirement, each NUMA-node handles a fraction of all requests. Therefore, a group of worker threads in each NUMA node handles corresponding requests to each logical unit.

- **NUMA-awareness on network buffers**. Network buffers are the destination memory locations for buffering the requested data before sending them over networks. The framework needs to allocate a group of dedicated networks buffers on each NUMA-node.

- **Avoiding potential bottlenecks.** The NUMA-aware solution improves access performance by minimizing remote accesses. Meanwhile, it also introduces several types of inevitable overhead. For example, **NUMA-awareness on cached data** means there should be a procedure to identify the relevant NUMA node. This additional procedure may lead to a fixed amount of latency for every I/O request. Therefore, we need to take such a potential overhead into consideration in the system design.

To meet these requirements, we reconsider the model of master-worker threads with an integrated storage cache and network buffer allocation. As depicted in

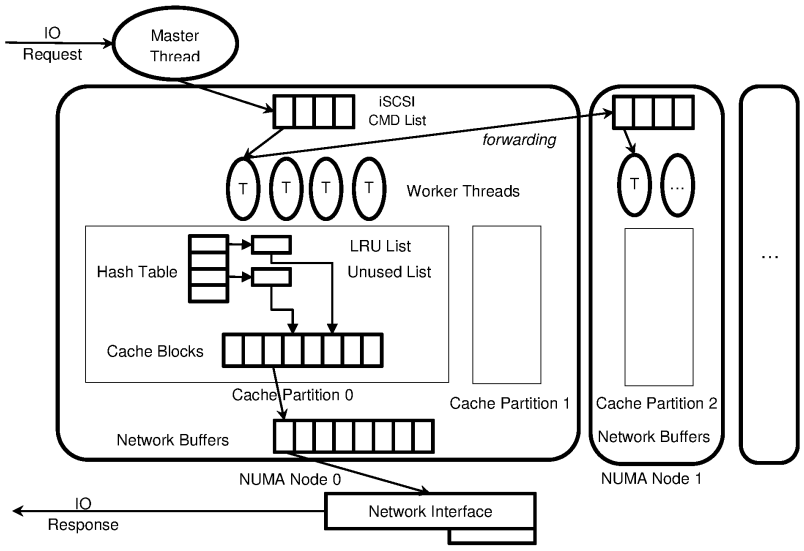


Figure 5-3: Software architecture of NUMA-aware cache. It achieves NUMA-awareness for iSCSI I/O accesses.

Figure 5-2, each NUMA node contains its own NUMA-aware cache, network buffers, and worker thread groups. Different from the existing framework, the worker threads in our model try to copy data from their local cache to a local network buffer in a cache-hit case. As all three entities involved in the I/O request are in the same NUMA node, the framework is able to gain the best performance to serve cached data. We use `libnuma` [71] in our implementation, the de facto library for Linux NUMA programming at the user level, which schedules threads and binds memory regions to a specified NUMA node.

5.2.2 Cache Organization

We first describe the cache organization in our design. In a NUMA host, each NUMA node maintains a cache region in its local memory banks. The SCSI storage corresponds to a storage address space that consists of one or more logical units, which are further divided into many data blocks, each of which has a unique "cache block number". A cache region can host many cache blocks, and each of them is used to cache one data block belonging to a logical unit. Here,

"cache block" is the unit of operation to our proposed cache.

Instead of allocating a single cache partition per NUMA node, a practice which is commonly adopted by others, we organize each NUMA node's cache into a number of "cache partitions", each of which is an independent, fully associative cache area that manages cache blocks with several different types of data structure, e.g., a hash table for valid caches, a linked list for invalid caches (i.e., empty buffers), and a linked list for LRU replacement candidates, as shown in Figure 5-3. This cache partitioning method has several advantages over the single partition approach. First, it eliminates contention from a global lock per NUMA node. With one cache partition per NUMA node, multiple worker threads compete for a global lock and wait for a critical section, thereby creating a bottleneck. To overcome this drawback, we chose to use a multi-partitioned approach with locks of smaller granularities. Multi-partitioning can also reduce hash collisions in comparison with a single partition. It is scalable to various systems, as its number of cache blocks is manageable within each cache partition, regardless of the total memory capacity of a particular system.

By hashing its cache block number, a cache block is associated with a unique partition. The hash function plays a critical role in improving cache performance. In our initial cache design, the hash function striped neighboring cache blocks into neighboring partitions with a modulo operation, i.e., Partition ID of a Cache Block = Cache Block Number *mod* Number of Partitions. When this method was applied to serve a large I/O request with a small cache block size, it led to excessive locking and unlocking contentions. For example, processing a 512 KB request with 4 KB cache block size required 128 partitions, and hence 128 locking and unlocking operations. To solve this problem, our current hash function carefully places a group of neighboring cache blocks in the same cache partition: Partition ID of a Cache Block = (Cache Block Number / Number of Neighbouring Blocks per group) *mod* Number of Partitions. In the previous example, if 128 neighboring cache blocks are configured within the same parti-

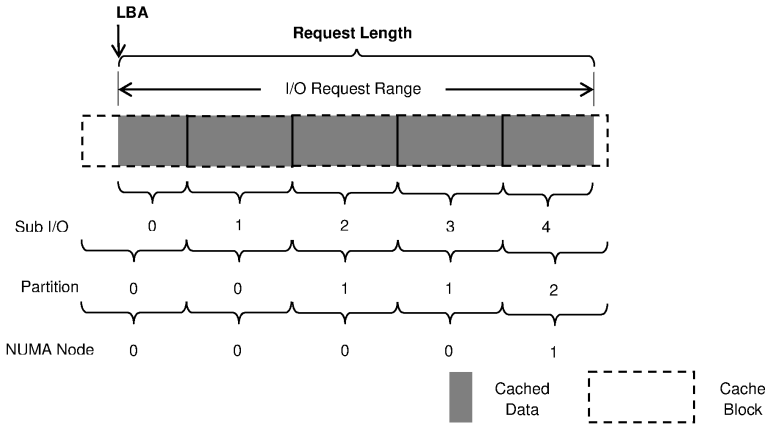


Figure 5-4: I/O request decomposition. In this 2-node NUMA system example, NUMA node 0 will be chosen because it has a better affinity to handle this I/O request.

tion, then a 512 KB request involves at most two partitions and two locking and unlocking operations.

Finally, our software also allocates a number of network buffers in user space at each NUMA node for both TCP/IP and RDMA network communication. When serving a request, it chooses a network buffer in the same local node of the designated worker thread. To avoid the overhead of dynamic memory allocation and release, our software pre-allocates both cache blocks and network buffers, and makes repeated use of them afterward.

5.2.3 Routing I/O Tasks to NUMA Nodes

As an I/O request arrives at the target host, the framework needs to interpret the request and to determine which NUMA node has the closest affinity to handle this request. To understand this interpretation step, we revisit the address space of an iSCSI storage system. The backend storage of a target consists of an array of hard drives or NAND-based flash memory that can be accessed via physical block addresses. The iSCSI target exports each backend storage device as a series of logical blocks, and an initiator addresses an iSCSI device by logical

block address (LBA). The size of a logical block is configured to be a multiply of physical block size. A cache block, described in the previous subsection, further contains one or several logical blocks.

Algorithm 1: I/O interpreting routine for calculating closest affinity NUMA node

Input: LOGICAL BLOCK ADDRESS, I/O LENGTH
Output: A CLOSEST AFFINITY NODE, SUB I/O REQUESTS

```
1  n ← number of Sub I/O Requests
2  m ← number of NUMA nodes in the host
3  create affinity array, aff[], with size m
4  create sub IO requests array, sio[], with size n
5  for i ← 1 to n do
6      Calculate cache operation related parameters for cache hit
7      Calculate disk operation related parameters for cache miss
8      Calculate PARTITION ID by hashing the Cache Block Number
9      Calculate NUMA NODE ID of sio[i] according to PARTITION ID
10     Accumulate I/O length of sio[i] to aff[NUMA NODE ID]
11 set CLOSEST AFFINITY NODE to the node with the maximum value in
   aff[]
12 return CLOSEST AFFINITY NODE
```

In practice, the data size of an I/O request varies. A large I/O request involving multiple cached blocks may spread across multiple cache partitions, as shown in Figure 5-4. The NUMA-aware cache decomposes the request into several independent sub-requests by Algorithm 1, and each sub-request is aligned with a cache block. To minimize data transfer between NUMA nodes, the algorithm chooses the node with the best locality (i.e. the node with the largest number of related cached data blocks) to process the request, schedule the request there, and aggregate all related cache data to a network buffer associated with the chosen NUMA node. For instance, in the situation depicted by Figure 5-4, the algorithm would choose node 0. Algorithm 1 involves complex address translation: Its processing time increases linearly with the number of sub-requests that belong to the same I/O request, and therefore could itself become a performance bottleneck.

After an I/O request has been forwarded to the NUMA node with the closest affinity, a worker thread running on that node will search for the requested data in the cache, and perform a memory copy on a cache hit and disk operations on a cache miss. We note that the NUMA-aware cache utilizes direct I/O by setting the `O_DIRECT` flag in Linux to bypass the page cache at the kernel level and accelerate data loading from disks to user memory directly.

5.2.4 Placement of the I/O Interpreting Function

Given the complexity of the I/O interpreting function, its placement also affects the feasibility of NUMA-aware caching and must be addressed carefully. We consider three different methods for placing this function:

- First, the master thread executes the interpretation routine and then dispatches the request to a worker thread on the selected NUMA node. Since the single master thread is already shared by all LUNs, this extra computation-intensive dispatching task could potentially create a bottleneck, and thus affect the overall system performance.

- Second, the master-worker architecture can be extended to a three-tier, master-scheduler-worker architecture. A group of dedicated scheduler threads could be arranged between the master thread and worker threads. The master thread immediately sends an incoming request to a randomly chosen scheduler thread. The scheduler thread executes the I/O interpreting routine, and forwards the request to a worker thread at the NUMA node with the closest affinity.

- Third, the master thread sends the request to a worker thread on a randomly chosen NUMA node. After performing the I/O interpreting routine, the worker thread either serves the request if its local NUMA node has the closest affinity, or forwards the request to the corresponding NUMA

node, as shown in Figure 5-3. Note that in this method, to avoid a duplicated interpreting routine from being executed again, we turn on a flag when the request is forwarded to the target NUMA node.

Our preliminary tests showed that the first approach increases the load of the master thread's CPU to 100% with many small I/O requests that are less than 32KB. The second and third approaches essentially adopt a similar distributed re-scheduling method, but with slightly different implementations. The third one relies on intelligent worker threads to avoid a forwarding operation if the first randomly picked worker thread affiliates with the corresponding NUMA node. We chose the third approach in our customized framework.

5.2.5 Discussions on Overhead and Scalability

Our NUMA-aware solution significantly reduces remote memory accesses and leads to efficient memory copies when serving cached data. However, it also introduces extra computation and scheduling overheads. Hence, there is a tradeoff between the benefit of NUMA-aware memory access and overhead. On one hand, to enable NUMA-aware memory organization and thread placement, the I/O interpreting and forwarding functions are indispensable and incur overhead in terms of I/O latency and CPU consumption. On the other hand, this overhead is compensated when I/O requests are larger, and the performance improvement from local memory access surpasses the overhead. In summary, the tradeoff is determined by the size of I/O requests: The larger I/O request size means the greater benefits realized with the NUMA-aware solution.

Software scalability is another consideration in our design. First, we divide each NUMA node's memory into partitions, and each partition has its own data structure to maintain its status. This avoids competition on a global data structure in a non-partitioned solution, and thereby increases efficiency and flexibility in serving concurrent requests. Second, as we described, we also carefully de-

sign our hashing method for placing cache blocks, such that neighboring cache blocks fall into the same partition. This optimized design reduces the overhead of locking/unlocking cache partitions. In summary, our design is scalable to large NUMA systems.

5.3 Decentralized Event Processing

The scalability of the I/O event processing becomes critical in IOPS-bound workloads, such as those with many small I/O requests. Modern network device can deliver millions of requests per second, and each request often incurs several I/O events, such as request arrival event and response completion event. This demands an efficient processing model that has scalable software processing performance even with many intensive I/O events.

The standard iSCSI/iSER software uses a centralized single-threaded event processing model, as shown in Figure 5-2. The master thread is responsible for processing and scheduling all the I/O events related to the requests from multiple initiators. Although the forwarding mechanism, presented in section 5.2.4, offloads the data affinity calculation from the master thread to worker threads, the master thread in such a model still constitutes a severe bottleneck for processing IOPS-bound workloads.

In this section, we present a decentralized event processing model to further improve the iSCSI server's performance in term of IOPS which is particularly important to process a huge number of small-scale requests. This design creates multiple event processing threads to better utilize multi-core computing resources. It not only benefits the overall processing throughput, but also binds the event processing thread to the affiliated CPU core to reduce event detecting latency. In addition, the decentralized design ensures load balance for the whole storage system because it distribute CPU-intensive event processing procedure to all NUMA nodes.

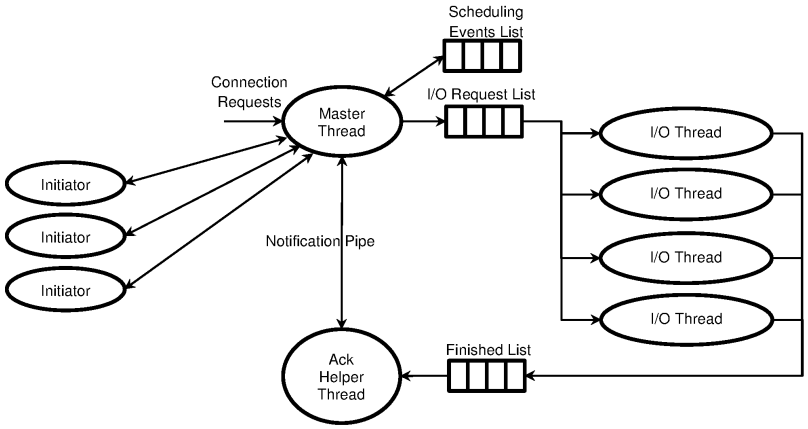


Figure 5-5: Event processing model in the standard iSCSI/iSER software.

5.3.1 Scalability Limitations in Standard iSCSI/iSER Servers

The conventional software design principle assumes that CPU is alway much faster than I/O devices in terms of throughput and latency. Therefore, the number of I/O event thread is much smaller than that of I/O operation thread. This principle broadly influences the web server design and implementation. For example, Nginx, a popular open-source web server implementation, uses an asynchronous event-driven approach to handling requests [72]. There is no dedicated I/O thread in the Nginx implementation: it assumes web server to be always I/O-bound and uses asynchronous I/O to overlap computing tasks and I/O tasks.

In the standard iSCSI/iSER server design, it uses a single master thread to detect and process I/O events that are generated by multiple clients, and deploy multiple I/O helper threads to perform read and write operations, as shown in Figure 5-5. This model worked for non-NUMA hosts and the single master thread can keep up with I/O performance because the I/O processing is always the bottleneck along the end-to-end path, and CPU is much faster than networks and disks. The master thread is the only entity that manages and controls the resources including connections and event queues. Therefore, the access on such resources can be lockless. Lockless resource operations simplify the event

processing implementation and maximize the full capability of a single CPU core. The state-of-the-art storage media, such as SSD, can deliver millions I/Os per second, and the I/O worker threads are not the bottleneck in such cases, while the master thread becomes the new bottleneck. Therefore, we need a new design to scale the master thread's capability on event processing.

5.3.2 Events Categories in iSCSI/iSER Servers

To scale up event handling in a multi-core system, we need to re-think the event processing model in iSCSI/iSER servers. We first present the event categories in iSCSI/iSER servers.

- **Incoming connection event.** Before performing actual iSCSI I/O operations, the initiator connects to the target, and triggers an incoming connection event. To handle this type of event, the iSCSI/iSER target server checks the eligibility of the connection by applying a white-list rule: each storage target media is configured to be available for a group of legitimate initiators. Afterward the server process allocates resources for the initiator, such as file descriptors and command lists.

- **Management I/O request event.** There are two types of I/O requests: management and data. Management I/O requests retrieve the properties of one particular storage media, such as block size and storage capacity. This meta data of storage is often handled by the master event thread instead of an I/O thread.

- **Data I/O request event.** Another type of I/O request is block-layer Data I/O request. For example, in the RDMA based iSER implementation, the initiator first submits requests using RDMA two-sided operations. The server detects incoming requests by checking the completion status of its receiving queue. Once a new requests arrives and is fetched from the

receiving queue, the server process parses the incoming request, and then retrieve data from (I/O write request) or send data to (I/O read request) the initiator.

- **I/O scheduling event.** In responding to an **Incoming I/O request event**, the iSCSI server divides the processing into multiple steps: I/O polling step, I/O submission step, and network transmission step. These steps, each of which is represented as an event, are queued in the list of scheduling events, as shown in Figure 5-5. An event thread processes multiple scheduling events in a batch to better utilize CPU resources.

- **Inter-thread communication event.** The master event thread and the I/O thread need inter-thread communication to synchronize I/O tasks. Because the inter-thread communication based on event queue involves block-and-wait operations, the standard programming model creates a helper thread for transforming the blocking operation to the event-driven Operating System pipe operation. Here pipe is a technique for passing information from one program process to another, or one thread to another. As a result, the master event thread also needs to manage the inter-thread communication event to check the availability of finished I/O requests.

5.3.3 Decentralized Event Processing Model

The event-driven paradigm constructs an event pool that tracks all resource handlers (i.e. file descriptors) and leverages multiplex interfaces, such as `select` and `epoll`, to detect I/O events associate with these resources. The standard iSCSI/iSER software relies on only one event pool managing multiple resources Although the master event thread can detect multiple concurrent events with a single event detecting call, the detected events are processed sequentially in the standard implementation.

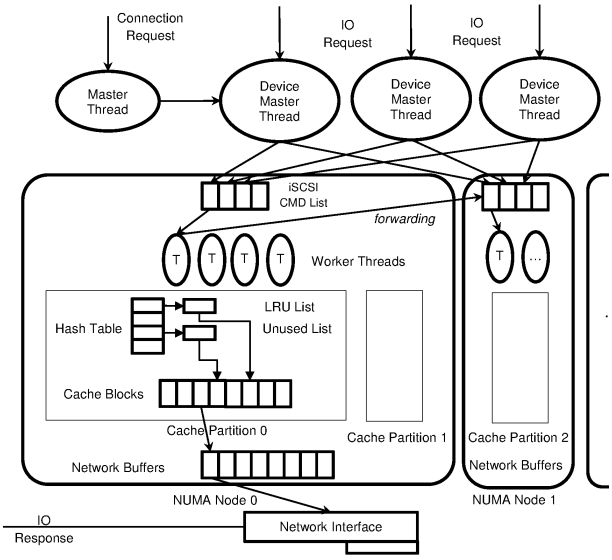


Figure 5-6: Decentralized event processing Model for the iSCSI/iSER servers.

We propose a decentralized event processing model to ensure high concurrency in event processing for iSCSI/iSER servers. The core idea is to design a parallel event processing model and to distribute many I/O related events to parallel threads on multi-core systems as fast as possible. To this end, for each logical unit, we create a group of threads to handle related I/O events and I/O operations. Different logical units can process their I/O events exclusively and in parallel. We enumerate three different aspects between our decentralized model and the standard one as follows:

First, each logical unit has a dedicated event processing thread. As shown in Figure 5-8, once the master thread detects an **incoming connection event**, it constructs a new event pool. In the iSER implementation, each initiator has two resource handler: a connection file descriptor and a completion channel descriptor. The event thread manages these two types of resource and uses the `epoll` system call to detect the upcoming events. Different logical units and initiators are managed by different event threads, and therefore this design can parallelize the event processing.

Second, each logical unit has a dedicated acknowledgement thread in the

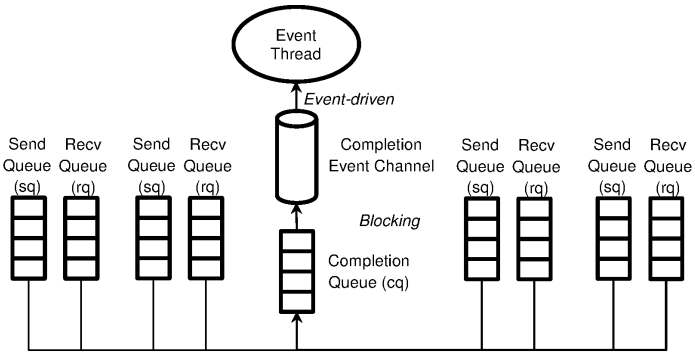


Figure 5-7: Centralized processing for RDMA network events.

decentralized solution, which is in contrast to the single threaded solution for the standard iSCSI server. The acknowledgement thread, also termed a helper thread, translates the completion of a series of RDMA network events into the corresponding I/O scheduling events, and queues them into the event queue for the designated logic unit. This acknowledgement thread utilizes an OS pipe utility to trigger I/O events, to pass the transfer status information back to the storage system, and to wake up the event thread dedicated to a logic unit.

Third, the decentralized design creates private network buffers for each initiator connection to attain lockless operations on buffer management and maximize parallelization. This is different from the standard iSCSI/iSER design where a single network buffer pool is shared by multiple initiators. In our design, when a new connection event arrives, the device master thread acquires a group of network buffers from the global network buffer pool, and use them repeatly during the life time of connection. The device master thread of a connection returns the network buffers to the global pool upon a disconnection event.

In summary, this decentralized event processing model improves the scalability of event processing in a multi-core environment by employing multiple event processing threads.

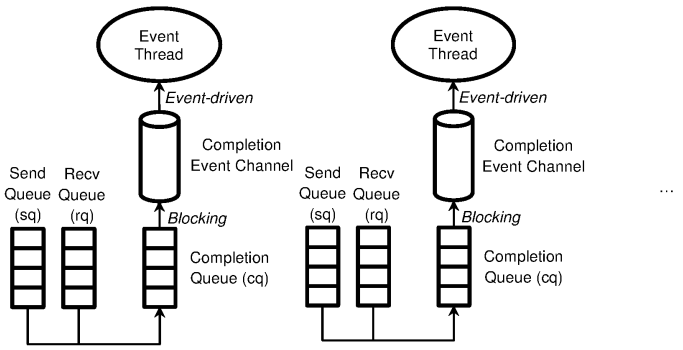


Figure 5-8: Decentralized processing for RDMA network events.

5.3.4 RDMA Network Events Processing

The iSER protocol has a sophisticated mechanism for processing network events as each I/O request triggers at least three RDMA operations. For example, to read data from a target, an initiator sends a request to the target using the two-sided SEND/RECEIVE semantics. Upon the request arrival, the target process issues an RDMA write to send data back to the initiator. The target composes another notification event and sends it to the initiator using a two-sided SEND/RECEIVE operation. More importantly, the three RDMA operations are asynchronous and their completion events are delivered through a shared completion queue. As shown in Figure 5-7, the single completion event channel is responsible to deliver all completion events from several connection pairs in the entire target system, and presents a bottleneck in a multi-core system.

We allocate multiple completion channels for different initiator connections, named queue pairs in the RDMA context, and associate a dedicated completion queue to each completion channel, as shown in Figure 5-8. Each completion channel belongs to a particular event pool managed by the event thread for an initiator. In addition, one server may contains multiple network interfaces, each of which is attached to different NUMA nodes. We bind the event thread to the CPU core that has the closest affinity to the network interface card for transferring data between the target/initiator pair. This greatly shortens the event deliver

Table 5.1: Testbed configuration for NUMA-aware cache

	Target Dell R820	Initiator IBM x3650	Initiator HP DL380
CPU * Cores	Intel Xeon E5-4620 2.20GHz 32 Cores	Intel Xeon E5-2660 2.20GHz 16 Cores	Intel Xeon E5-2650 2.00GHz 16 Cores
NUMA nodes	4	2	2
QPI Speed (GT/s)	7.2	8	8
Memory(GB)	768	128	384
Local Memory Bandwidth (GB/s)[1]	18.71	25.21	20.62
Remote Memory Bandwidth (GB/s)[2]	3.26	13.99	11.40
Max Memory Bandwidth (GB/s)[3]	42.6	51.2	51.2
Network Adapters	56Gbps IB FDR	56Gbps IB FDR	56Gbps IB FDR
OS	CentOS 6.3	CentOS 6.3	CentOS 6.3
Kernel Version	2.6.32-279	2.6.32-279	2.6.32-279
MLNX OFED Version	1.5.3-3.1.0	1.5.3-3.1.0	1.5.3-3.1.0
MTU Size (IB)	65520	65520	65520
RTT(ms)	0.07	0.07	0.07

[1,2] The practical memory bandwidth is from the multi-threaded STREAM benchmark;

[3] The theoretical maximum memory bandwidth is from http://ark.intel.com;

path and minimizes the NUMA overhead.

5.4 Evaluation with Synthetic Workloads

In this section, we study the effects of NUMA-aware caching using synthetic workloads by comparing it with the default OS page cache available on the current storage servers. Our goal is to confirm the intuitive discussion presented in Section 5.1. We conduct different sets of experiments with synthetic workloads.

5.4.1 System Setup

Our iSCSI-based SAN testbed consists of four initiators and a target connected to a stand-alone InfiniBand storage fabric. All hosts are equipped with two Mellanox ConnectX-3 FDR InfiniBand adapters which all connect to a Mellanox SX6018 FDR InfiniBand switch. All hosts use the Intel sandy-bridge chipset which supports PCI Express Gen3. We set up a 4-node Dell R820 as the target server. Each NUMA node contains an eight-core Intel Xeon E5-4620 CPU with 16-MB shared L3 cache and 192 GB local main memory for a total of 768 GB main memory. All NUMA nodes are interconnected via QPI I/O bus for inter-node traffic. The internal disk array has an aggregate capacity of 3.3 TB. Configurations details for all the hosts are shown in Table 5.1.

To serve multiple initiators simultaneously, we export a dedicated portion of disk space, a.k.a. a logical unit, from the target to each initiator. We use *XFS* file system to format the entire target disk array and create eight pre-allocated 300 GB files as logical units. Each logical unit goes through a dedicated adapter, while all the devices share a global NUMA-aware cache.

We use Flexible I/O Tester (`fio`) [32], an I/O benchmark with a number of fine-grained parameters, including I/O engine, block size, and random distribution of I/O sequences. It also reports I/O performance statistics including bandwidth, IOPS, CPU usage, and I/O latency. To eliminate the thread and memory migration overhead in on the initiator side and ensure a fair comparison, we integrate the *libnuma* into `fio` and statically place each thread of a single experiment into the same node, using the same CPU cores and memory banks. On the target side, we use the "nmon" tool to record the CPU utilization by application during each experiment.

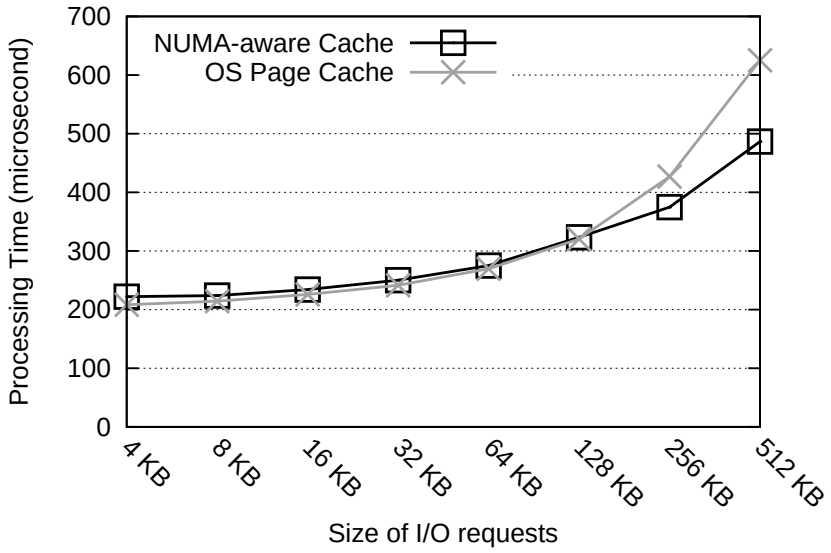


Figure 5-9: Processing time of I/O requests with different sizes.

5.4.2 Evaluation of Request Processing Time

In the first set of experiments, we investigate the NUMA effect on the request processing time of the iSCSI/iSER protocol. The processing time for each single I/O request is formulated in Section 5.1. We preload a 720GB dataset into the NUMA-aware cache or the page cache by sequentially reading data from disks. For the NUMA-aware cache, the data is evenly distributed across all the NUMA nodes according to the partitioning and hashing method. For the page cache, the dataset is cached inside the kernel memory by default provided that sufficient space is in the testbed. To simulate how I/O is handled by a real system, we use *numactl* to interleave the page cache in several NUMA nodes. For the NUMA-aware cache, we set a 512 KB cache block size, 36 partitions in each NUMA node, and every four consecutive (neighboring) cache blocks in the same partition. To eliminate the effect of CPU register cache on data re-use and prefetching, we randomly access data and assign a unique random seed (by setting `randrepeat=0` in `fio`) in each run of the experiments. The test cases include I/O requests ranging from 4 KB to 512 KB.

In each test case, we run `fio` for 300 seconds with a fixed rate of 10 IOPS,

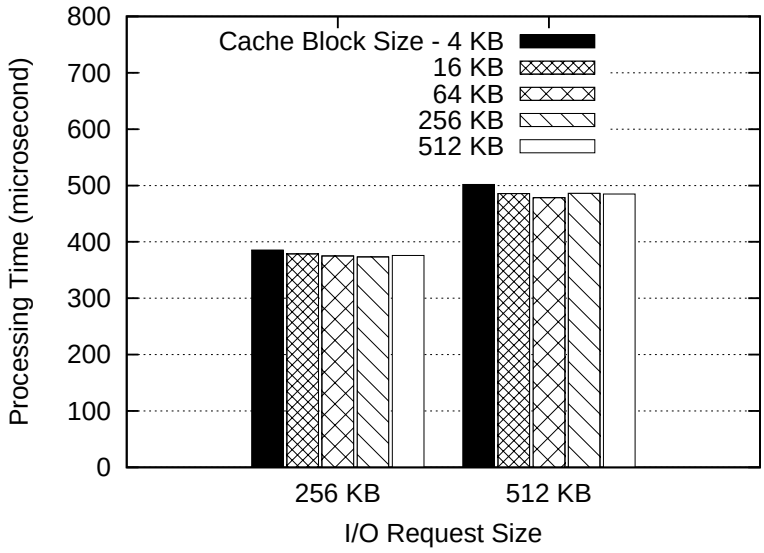
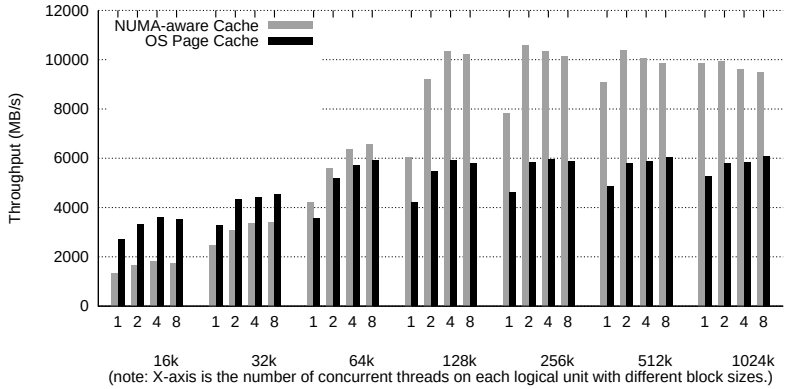
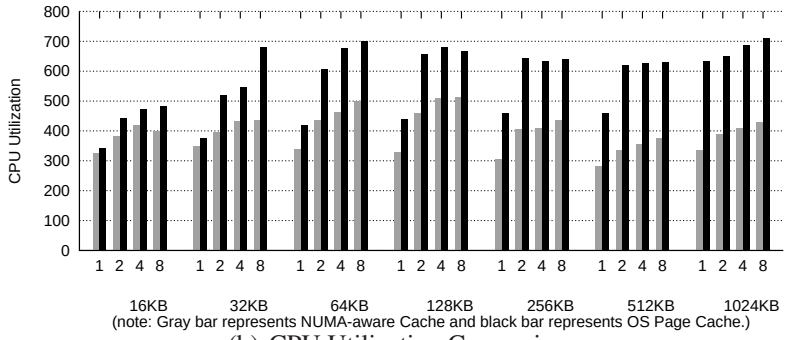


Figure 5-10: Processing time with various cache block sizes.

and obtain performance statistics for processing time, as shown in Figure 5-9. We have made the following observations. First, the NUMA-aware cache reduces average processing time by about 13% to 23%, for 256 KB and 512 KB requests, respectively. Second, the actual improvement on memory access latency is more pronounced with large I/O requests. These two observations confirm the calculated result by Equation (5.1), i.e., with both cache methods, the processing time includes a fixed RTT between initiators and target (about 70 microseconds), a fixed network transmission time (about 156 microseconds for a 512 KB block), and queuing delay (about 30 to 50 microsecond for 512 KB blocks at 10 IOPS rate). According to Equation (5.1), the average memory access latency can be reduced by about 66% in the best case scenario with our NUMA-aware cache with large I/O requests. Third, with small I/O requests (4 KB to 32 KB), the overhead of NUMA-aware modules outweighs the benefit of NUMA-aware memory accesses. In these test cases, we configure the NUMA-aware cache to avoid the overhead of forwarding I/Os. As a result, the NUMA-aware cache and the page cache demonstrate a compatible processing time even for small I/O requests, as shown in Figure 5-9.



(a) Throughput Comparison



(b) CPU Utilization Comparison

Figure 5-11: Throughput and CPU utilization comparison between NUMA-aware cache and page cache with random access pattern.

In addition, we investigate the effect of cache block size for large I/O requests. As described in subsection 5.2.2, the hash function places neighboring cache blocks into the same partition to reduce the overhead of locking and unlocking multiple partitions related to a given I/O request. As shown in Figure 5-10, there is no notable difference in processing time for an identical I/O request with different cache block sizes. This validates our choice with regard to the hash function.

5.4.3 Random Access on Fully Cached Data

In this set of experiments, we evaluate the aggregate throughput of the whole system with I/O requests ranging from 16 KB to 512 KB. At each initiator, we run parallel I/O threads to generate contention workloads. We use the same

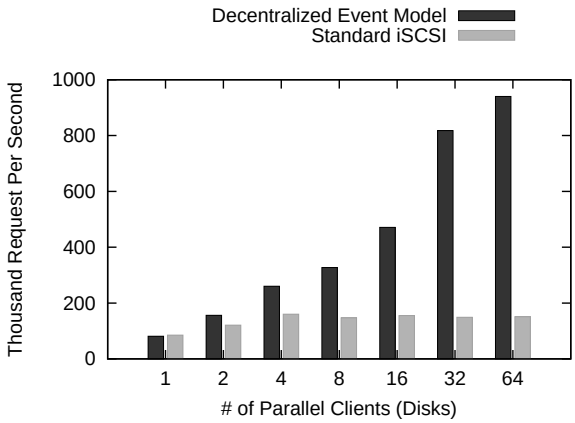
method, as mentioned in the last section, to load data from disk to memory and spread data across all the NUMA nodes in the target. We configure 400 GB cache memory in total, and load 50 GB memory for each logical unit. To avoid any CPU cache effect, each I/O thread is assigned with a unique random seed. The `tgtd` process is configured with 16 worker threads for each logical unit. For the NUMA-aware cache, we set 10 partitions in each NUMA node, and each partition manages 10 GB memory with 512 KB cache blocks.

Figure 5-11 shows the aggregate read throughput and corresponding CPU utilization performance of both NUMA-aware cache and page cache. We have made the following observations:

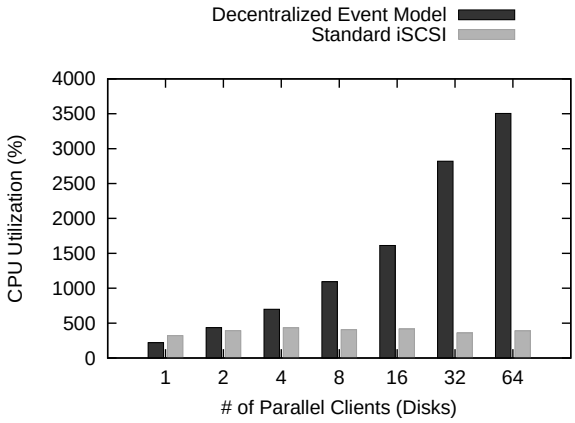
- For requests larger than 64 KB, `tgtd` with the NUMA-aware cache outperforms the standard `tgtd` with the page cache by up to 80%. This improvement is more impressive than the 20% reduction in processing time. Under the stressed/overloaded condition, the page cache inevitably incurs more inter-node traffic, such as remote memory access and CPU cache synchronization by the MESI (Modified-Exclusive-Shared-Invalid) protocol, than the NUMA-aware cache in the target host. The page cache generates more contention and congestion on QPI and suffers longer latency. On the other hand, with the NUMA-aware cache, `tgtd` is able to saturate network bandwidth with the appropriate cache block size and number of parallel I/O threads.

- For smaller I/O requests, such as those for 16 KB and 32 KB, the overhead of remote memory access in the page cache is less than that of Algorithm 1 and the I/O forwarding in NUMA-aware cache. Thus, the size of I/O requests plays an important role in determining the tradeoff between the NUMA-aware cache and the OS page cache.

- Finally, the NUMA-aware cache reduces CPU load in all our experiments, as remote memory access incurs a synchronization cost in NUMA sys-



(a) Throughput Comparison



(b) CPU Utilization Comparison

Figure 5-12: Throughput and CPU utilization comparison between the decentralized event processing and the standard one with random accesses. The experiments use 512-byte I/O read requests, and the dataset is cached entirely in the target main memory.

tems and leads to higher CPU consumption. Here, each fully occupied CPU core is shown with 100% CPU utilization, and the total CPU utilization can be up to $32 \times 100\%$ on the 32-core target host. In experiments wherein the NUMA-aware cache fully utilizes network bandwidth, the *tgtd* consumes 300% less CPU usage, or equivalently saves three CPU cores.

During additional tests, we changed the file access pattern and used a Zipf-like distribution [73] to evaluate cache performance again. We found that the

NUMA-aware cache achieves similar performance gain in these new tests to that in the random distribution case.

5.4.4 Decentralized Event Processing Evaluation

We evaluate the decentralized solution for serving small I/O requests. The previous evaluation reveals the scalability problem of the single-threaded event processing design with small I/O requests for both the NUMA-aware cache solution and the OS page cache. The master thread is the bottleneck to high IOPS workloads as it can not utilize the concurrent processing capability of multiple cores. To evaluate the scalability and performance of the decentralized solution, the iSER target exports multiple iSER disks to many initiators. We use the fio benchmark again to generate concurrent I/O requests. We enable the direct I/O option to bypass the kernel cache and to eliminate its caching effect at the initiator side.

Figure 5-12 shows the performance comparison between the iSER with decentralized event processing and the standard iSCSI/iSER software. We made four observations as follows:

First, the decentralized solution demonstrates linear increase in performance when the number of concurrent clients increases to the physical limitation. Here the iSER server has 32 physical CPU cores with hyper-threading enabled. Given the computation intensive event processing, the 64 clients with hyper-threading enabled have 15% improvement over the 32 clients in term of throughput. Secondly, the standard iSCSI/iSER software does not scale up with more concurrent clients. Its performance is capped at 160K IOPS: the maximum performance that a single CPU core delivers in the testbed. Third, our decentralized implementation achieved 940K IOPS, a factor of six speed up over the standard iSER/iSCSI. To the best of our knowledge, this is the best performance that software iSCSI/iSER achieves for a single target server. Forth, the CPU utilization

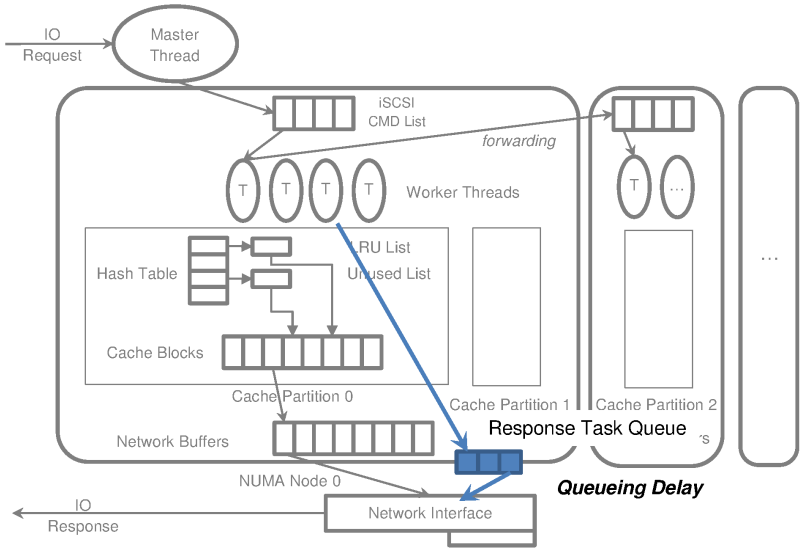
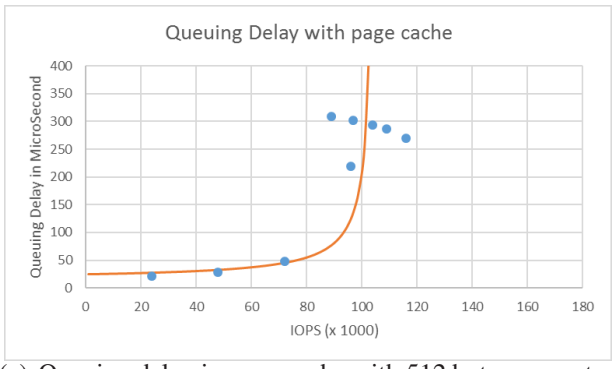


Figure 5-13: Queuing delay in iSCSI target system. We monitor the delay in the network buffer queue at user space.

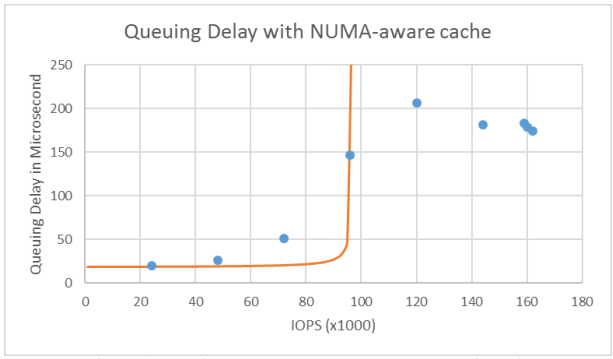
in the decentralized solution is proportional to its throughput. This confirms the scalability of the distributed event processing: Our decentralized solution scales up its performance by utilizing more CPU cores.

5.4.5 Queuing Delay Analysis

In this section, we profile the impact of queuing delay to the entire processing latency. We first define it in the iSCSI target software, then construct various workloads with different I/O patterns (request sizes and IOPS), and analyze the experiment results. We consider the queuing delay in an iSCSI system as the interval between when a response data is placed in the network sending queue and when a network adapter starts its transmission. We note that there are potentially other queues and queuing delays in the iSCSI target system, and our experiments intend to qualitatively illustrate the impact of the queuing delay. In Linux SCSI target framework implementation, a worker thread enqueues the task that responds to a data request into a response task queue once the system fills the relevant data per a data request into network buffers. At this stage, it



(a) Queuing delay in page cache with 512 bytes requests.



(b) Queuing delay in NUMA-aware cache with 512 bytes requests.

Figure 5-14: Queuing delay with small I/O requests. The points represent the delay in the network buffer under different loads, and the lines represent M/M/1 model approximation.

enters the final stage of sending the requested data to the requesting party. Then a network thread checks the task queue and locates those tasks that are in the ready-to-transmission stage. Once it finds such a task, it dequeues the task from the queue and engages network to send out data immediately. The queuing delay for a data transmission task in iSCSI target system is approximately the interval between when it is enqueued to and dequeued from a response task queue.

In practice, the queuing delay in Linux SCSI Target Framework (tgtd) software consists of the waiting time in a response task queue and the associated overhead for queue management, e.g., mutex locking and condition variable notification. Figure 5-13 shows the queuing delay of the response task queue. The M/M/1 queue is an appropriate model to approximate the queuing delay in

Table 5.2: IOPS comparison under different workloads.

Proposed IOPS	Actual IOPS with Page Cache	Actual IOPS with NUMA-aware Cache
24k	24k	24k
48k	48k	48k
72k	72k	72k
96k	96k	96k
120k	116k	120k
144k	109k	144k
168k	89k	159k
192k	97k	160k
216k	104k	162k

such a systems: As the arriving data rate (requests per second) approaches and exceeds the limit of the request processing capacity (also known as the transmission rate of measuring how many requests can be processed within a given time interval), the queuing delay increases exponentially. To profile the queuing delay of a running program, we use the clock_gettime() function to generate timestamp (with a resolution of nanosecond) and calculate the queuing delay accordingly. We deploy the tgtd software in a 4-node NUMA system, and use three hosts to generate workloads to randomly read data from the cache that is already warmed up and managed by either the default OS page cache or our customized NUMA-aware cache. Then we comprehensively measure the queuing delay in the tgtd software that is overloaded for serving cached data.

Each of the hosts contains two 56 Gbps InfiniBand adapters that connect to an InfiniBand FDR 56Gbps switch. During each test, we run fio for 60 seconds to generate I/O requests. We calculate the average queuing delay from tgtd, and the average total processing latency and the actual number of completed I/O requests served by each of two caching systems.

In this experiment, we use fio to generate many workloads with a small I/O request size: 512 bytes per I/O request. Figure 5-14 captures the variations of queuing delay and the number of IO requests that are actually served when the number of IO requests to be sent out every second linearly increases. This experiment explores the capability of serving concurrent requests in the tgtd process. Table 5.2 lists the actual IOPS being served under different incom-

ing IOPS rates. We make the observations from Figure 5-14 and Table 5.2 as follows:

- With the OS page cache or the NUMA-aware cache, the average queuing delay consistently follows an exponential growth curve before tgtd reaches its performance limit. The page cache sustains about 110K IOPS while the NUMA-aware cache sustains about 160K.
- The queuing delay contributes 15% to 25% to the total processing delay time (Equation (5.1)).
- when IOPS is low, the minimum delay of 20 microsecond is due to the overhead associated with queue management.
- Initiators (clients) use synchronous I/O interfaces (read()) with parallel threads, and thus each thread suffers long queuing delay and cannot submit more I/O requests into the target when heavily loaded. Eventually, the client and server reach a saturation point, and no more requests can be submitted anymore unless earlier requests are completed. Therefore, queuing delay has an upper bound.
- A network adapter, along with its device driver and the system thread which manage and handle network events from the adaptor, not only has the bandwidth limitation (bit rate), but also has the IOPS limitation. Even with the highest IOPS (162K), all 512 byte requests add up about 648 Mbps bandwidth, which is far more less than the bare-metal network bandwidth. In such a high IOPS rate, the system thread for the network device already hits its IOPS limitation and cannot scale up anymore.
- Its counter-intuitive that the queuing delay in the NUMA-aware cache system decreases when the sending rate increases from 120K/second to a higher IOPS. This result is caused by sending multiple responses in a

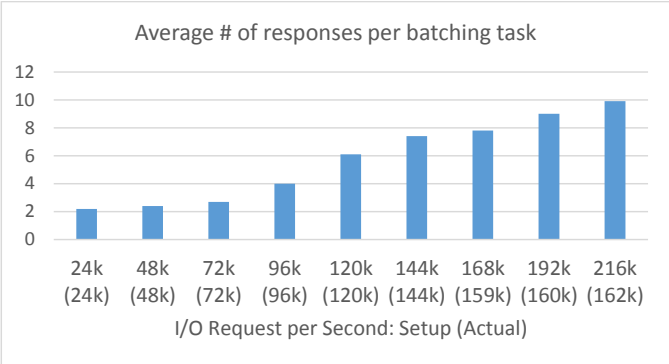


Figure 5-15: Average number of responses within a batching task in NUMA-aware cache solution.

batch. The tgtd's RDMA driver sends a number of responses into the network adapter via a batch system call (ibv_post_send()). In Figure 5-15, a higher IOPS of workloads increases the average number of responses within a batch task, and consequently shifts the queuing delay from software into hardware by queuing more responses earlier in the hardware queue. Under this circumstance, our profiled queuing delay indeed decreases moderately, nevertheless the hardware queue delay increases and is difficult to measure.

We conduct another experiment with large I/O requests (512 KB). In this experiment under stressed conditions, the queuing delay is 1% to 5% of the total processing time. To process larger I/O requests, the total processing time is more likely to be dominated by the nodal processing step (e.g., data copy from cached memory into network buffers).

In summary, the queuing delay we observed in the iSCSI system is consistent with the classical queuing theory, and it cannot be neglected especially at a high I/O request rate. We note that our NUMA-aware cache gains advantages mostly when I/O request size is large and queuing delay is not the main factor.

Table 5.3: PostMark performance

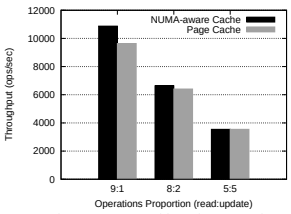
Data set	500 GB			1 TB		
Number of files	1,000	10,000	100,000	1,000	10,000	100,000
Transaction time on NUMA-aware cache (sec)	988	974	1006	5997	6110	8112
Transaction time on page cache (sec)	1401	1491	1611	8516	9236	14387
Ratio	0.705	0.653	0.624	0.704	0.662	0.568

5.5 Evaluation with Real-life Workloads

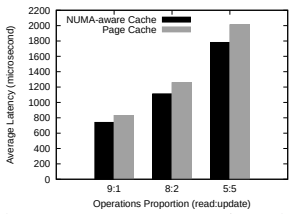
In this section, we evaluate both NUMA-aware cache and OS page cache using two real-life workloads, PostMark and YCSB over MongoDB. Our goal is to quantify the performance gains of NUMA-aware cache with realistic benchmarks. In this set of experiments, we configure the Dell host as the iSCSI target to export a 1.3 TB disk data (in an *XFS* file system) to an IBM host with 128 GB main memory used as the iSCSI initiator. The performance of disk array is about 600 MB/s with sequential read and 15 MB/s with random read. The NUMA-aware cache is configured with a total cache size of 720 GB (the entire physical memory in the Dell host) for all NUMA nodes, 32 partitions in each NUMA node, and a cache block size of 512 KB.

5.5.1 The PostMark Workload

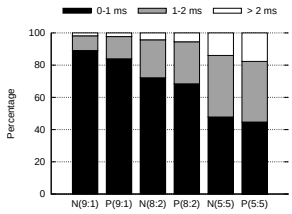
PostMark [74] is an email server workload simulator. It first creates a number of files repeatedly and randomly chooses a file, and then sequentially performs transactions (either read or write data) upon the file. The workload generator is executed on the initiator host. We generate a 500 GB dataset, which can be loaded entirely into the main cache memory of the target server, and another 1 TB data set, with about 75% cache hit rate, on the storage server. We use a request size of 512KB in PostMark, the same as the cache block size, and configure the workload generator to access each file three times on average during



(a) The overall throughput (ops/sec) with different read-to-update ratios. The relative improvement of NUMA-aware cache are 12.8%, 3.5% and 0%.



(b) Average transaction latency of read operations with various ratios. The relative improvement of NUMA-aware cache are 10.98%, 11.56% and 11.43%.



(c) Latency distribution of read operations with different read-to-update ratios. 'N' stands for NUMA-aware cache, and 'P' for Page cache.

Figure 5-16: The results of YCSB over MongoDB.

the entire transaction test.

Table 5.3 shows the performance of the PostMark test when the data set is served by the NUMA-aware cache and the page cache. The ratio is calculated as the transaction time of the NUMA-aware cache divided by that of the page cache. The NUMA-aware cache shows a 30% to 44% improvement in total transaction time. Furthermore, it achieves a comparable improvement regardless of whether actual disk operations are performed.

5.5.2 The YCSB Workload

Yahoo! Cloud Serving Benchmark (YCSB) [75] is a framework and common set of workloads for evaluating the performance of different "key-value" and "cloud" serving stores. It generates configurable workloads by a multi-threaded client, and submits them to a data serving system. We chose MongoDB, a state-of-the-art "key-value" database as the data serving engine, and stored all database files on the iSCSI disk at the target server. Database records are configured with 10 fields and 1000 bytes per field, and the total dataset size is 500 GB. Three test workloads have a read-to-update ratio of 9:1, 8:2, or 5:5, respectively. The YCSB client is configured to run with 10 parallel threads.

On our testbed, the 500 GB data set is fully cached by either the NUMA-

aware cache or the page cache on the storage (target) server during the data preloading step. The request distribution is uniform. The main memory of the database server is 128 GB. About 75% of read requests cause page faults in the database server, and are subsequently dispatched to the storage server.

Figure 5-16(a) shows the overall throughput of MongoDB with different read-to-update ratios. NUMA-aware cache achieves 12.8% improvement in the 9:1 case. Update operations dominated the transaction time in the 8:2 and 5:5 cases, as they introduced much higher locking and synchronization costs. Therefore, the effect of the NUMA-aware cache on the storage server is not obvious in update-heavy workloads.

We focus on read operations in the following discussion. First, the NUMA-aware cache improves the average transaction latency of read operations in all three test cases, as shown in Figure 5-16(b). The relative improvements of the NUMA-aware cache over the page cache for three test cases are 10.98%, 11.56%, and 11.43%, respectively. Second, as shown in Figure 5-16(c), the NUMA-aware cache improves individual transaction latencies of read requests. For example, in the 9:1 case, the latency of 89.08% read requests under the NUMA-aware cache is below 1 ms, while with the page cache, fewer (about 83.90%) read requests achieve a similar performance.

5.5.3 Decentralized Event Processing Evaluation with YCSB

We use YCSB and MongoDB to evaluate the performance of our decentralized event processing model. The iSER target exports block devices to MongoDB server. MongoDB uses the mapped memory call, `mmap`, to synchronize data between memory and block device. During runtime, if the storage block is not mapped into MongoDB's process memory space or is evicted from main memory, the OS generates a page fault to fetch data from the iSCSI/iSER storage system. In data-intensive workloads, the page fault handling is critical to the

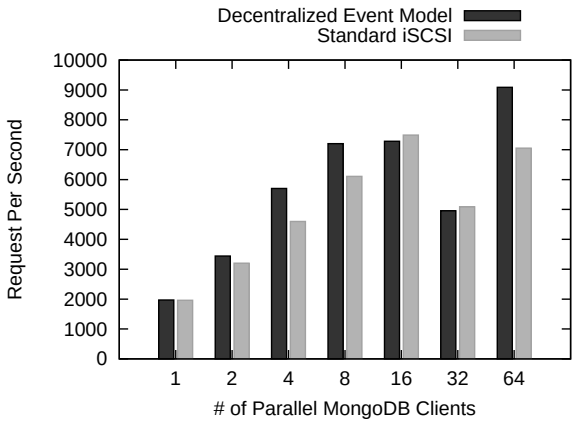


Figure 5-17: Throughput of YCSB workloads over MongoDB which in turn retrieve data from iSER target. Each MongoDB instance contains 200,000 key-value pairs, and each client operate on an independent Mongodb instance. The YCSB clients operates on the dataset for 5 minutes following uniform access pattern. The 1 client to 16 clients are in a single host. The 32 clients are in two hosts. The 64 clients are in four hosts.

overall performance of the storage system.

In the data loading phase, the YCSB clients insert 200,000 key-value pairs, each of which contains a 512-byte value. We use Linux control groups to limit the size of mapped memory for the MongoDB process during runtime to be 20 mega-bytes. We deploy 1, 2, 4, 8, and 16 YCSB clients in a single physical host, 32 clients in two hosts, and 64 clients in four hosts. The MongoDB instance resides at the same physical server that also host the YCSB clients.

Figure 5-17 shows the performance comparison between the decentralized solution and the standard solution. First, with concurrent 2, 4, 8 clients, the decentralized solution outperforms the standard solution by up to 24%. The decentralized solution provide parallel event processing capability, and can improves the performance on page fault handling. Second, with 16 and 32 clients, the MongoDB and Java-based YCSB benchmark have resource contentions at the initiator. Therefore, the target server is not the bottleneck in this testbed configuration. Third, we use 64 clients to generate many page faults from four physical hosts, and the decentralized solutions shows 28.8% performance gains

comparing to the standard solution.

5.6 Summary

In this chapter, we have shown that an inefficient NUMA remote memory access becomes a bottleneck to achieving high throughput and low latency in state-of-the-art SANs. We resolved the inefficiency problem and implemented a NUMA-aware cache for iSCSI-based storage area networks. Our NUMA-aware cache was already integrated into the Linux SCSI target software. Here we detailed the implementation details, including software structure, cache organization, and I/O request interpretation and processing. To tackle the scalability problem on small I/O request processing, we design and implement a decentralized I/O event processing model for multi-core and NUMA systems. We evaluated our solution on a 4-node NUMA testbed, and showed its sizeable performance improvement with both synthetic and real-life workloads. During the synthetic workload evaluation, the average processing time for individual I/O requests in an underutilized iSCSI system decreased by about 13% to 23% for large requests (256 KB and 512 KB), and the NUMA-aware cache and the default page cache achieved a compatible processing time for small I/O requests (4 KB to 64 KB). For the applications of handling bulk data in a busy iSCSI system, the aggregate throughput was improved by up to 80% with I/O request size ranging from 128 KB to 1024 KB. In real-life workloads, the NUMA-aware cache achieved up to 43.2% throughput improvement in the PostMark benchmark, and up to 12.8% throughput improvement for the read-dominant workloads in YCSB over MongoDB tests.

In summary, the NUMA-aware techniques proposed in this chapter provided a viable alternative to the standard OS page cache, and can greatly enhance the performance of iSCSI protocols over ultra high-speed networks. The decentralized event processing model further improved the software scalability on

intensive small requests processing.

Chapter 6

RDMA-Based NUMA-Aware End-to-End Performance Optimization

Data-intensive applications place stringent requirements on the performance of both back-end storage systems and front-end network interfaces. However, for ultra high-speed data transfer, for example, at 100 Gbps and higher, the effects of multiple bottlenecks along a full end-to-end path, have not been resolved efficiently. In this section, we describe our implementation of an end-to-end data transfer software at such high-speeds.

6.1 Introduction

Various data-intensive applications require ultra high-speed data transfer capability, such as those in data centers, cloud-computing environments, and distributed scientific computing. They frequently need data transfer software to support true end-to-end data and file delivery, i.e., between the storage systems attached to the source and the destination hosts. The Department of Energy's (DOE's) Magellan cloud data centers [4] are interconnected by the 100 Gbps

links of the DOE's Advanced Network Initiative (ANI). Such an architectural layout is often found in the DOE's National Laboratories, for example, three leadership computing facilities hosted at Argonne National Laboratory, Oak Ridge National Laboratory, and the National Energy Research Scientific Computing Center, respectively, and the tier-1 Large Hadron Collider computing facilities at Brookhaven National Laboratory and Fermilab that play a vital role in searching through petascale to exascale experimental data for scientific insights and discoveries [5]. The science programs (climate simulation, astrophysics, high-energy physics, material science, and system biology) at these DOE Laboratories frequently rely on high-performance supercomputers and server clusters, along with back-end storage systems encompassing hundreds of petabyte disk and tape storage, to run computing and data intensive applications, and to move data from experiments and simulations between computing and storage infrastructures and frequently across wide-area networks. Our primary goal is to design and deliver an efficient, extremely high-performance data transfer tool for these computing facilities that share an infrastructural layout similar to that depicted in Figure 1-1. To scale up data transfer to 100 Gbps and higher, we must overcome at least three different types of bottlenecks along the end-to-end paths that consist of hosts, networks, and storage systems.

First, to overcome the processing bottleneck of individual hosts, multi-core hosts often are employed for ultra high-speed data transfers. As the number of CPU sockets and cores per CPU die grows in the multi-core architecture of modern computer hosts, it becomes increasingly difficult and inefficient to have the same latency in memory access across all CPU cores. A state-of-the-art CPU architecture integrates a memory controller as a core component within the CPU die, and discards the external memory controller hub, a component that might become a bottleneck in a multi-core architecture [13]. Memory banks in different locations of a motherboard are attached to their corresponding CPUs. Therefore, the access latencies from a specific CPU core to different memory

banks are no longer same. With green computing restricting volume and power consumption, vendors turn to the Non-uniform Memory Access (NUMA) model to achieve higher resource density [15–17]. Although the high-speed connectivity between CPUs greatly facilitates arbitrary memory access, for example QuickPath Interconnect [13] and Hyper Transport [14], an application tuned for local memory access always performs much better than those that are not.

Second, advanced network technologies and protocols are employed to fully utilize the bare-metal bandwidth of ultra high-speed networks, at 100 Gbps and higher, and to eliminate network performance bottlenecks. Remote direct memory access (RDMA) [8] is one of these promising technologies because it can boost the performance of high-speed networks significantly. By enabling network adapters to transfer bulk application memory blocks to or from remote ones, and eliminating data copies in protocol stacks, RDMA achieves low latency and high bandwidth. InfiniBand [9, 10], the original RDMA implementation, dominates the technology market of intra-data center interconnections, while RDMA over Converged Ethernet (RoCE) [11] extends RDMA's capabilities to the networks between data centers that might be thousands of miles apart. Consequently, RDMA offers an opportunity to assure that large data synchronization and movement within or between data centers for applications to accomplish their routine tasks in a highly efficient manner.

Third, back-end storage systems within a server often become a severe bottleneck due to the low bandwidth of traditional magnetic disks or even recent flash solid-state disks (SSDs). One alternative to overcome this bottleneck of back-end storage systems is to build storage area networks wherein one assembles multiple storage components to provide aggregated bandwidth commensurate with a host's processing speed and its bare-metal network bandwidth. To configure and adapt high-performance RDMA networking technology into storage area networks, researchers [40] implemented an iSCSI extension for RDMA (iSER) [37], to enable SCSI commands and objects to be transferred

over RDMA-based networks, such as InfiniBand and RoCE.

In this section, we describe the design, tuning, and performance evaluation of a novel high-speed data transfer system for delivering data at 100 Gbps in an end-to-end fashion. The system utilizes a pair of multi-core front-end hosts (sender and receiver). Our research includes the follows. First, our back-end storage systems use the standard iSER protocol that is configured for high-speed data access. The protocol enables InfiniBand based data delivery from the back-end storage systems to the front-end hosts. This design allows us to eliminate the back-end storage bottleneck with the scalable InfiniBand. Second, between the front-end hosts with multiple network connections, we integrate our RDMA-based file transfer protocol, RFTP [21, 22, 24], into the end-to-end data transfer system, and optimize its performance to maximize bandwidth throughput and minimize host processing overhead. Third, for all hosts along an end-to-end path, we optimize their performance via NUMA tuning. Thus, we minimize the impact of host processing overhead. We note in the current implementation of iSER or RFTP, the NUMA factor is not considered, and we have observed the performance benefit of simple NUMA tuning in this chapter. To summarize, our design is the first to achieve 100 Gbps and higher end-to-end real data file transfers between one pair of commodity hosts, and to do so, we have overcome several aforementioned bottlenecks. We evaluate our system comprehensively, using the testbeds that closely resemble the production environments, common in large national laboratories and commercial cloud computing providers. Furthermore, more tests were performed with inter-data center data transfers along long-haul high bandwidth links of over 4000 miles long.

The rest of this Chapter is organized as follows. In Section 6.2, we present the background information, and the motivations of our research. We describe our system design in Section 6.3, and comprehensively evaluate the entire end-to-end system in Section 6.4. Finally, we offer our conclusions and highlight our contributions.

6.2 Background

In this section, we present evidences to show that the advances in hardware technology improve bare-metal performance, but existing software is not developed to take advantage of these advances. Consequently, multiple bottlenecks and issues still exist along end-to-end data transfer paths. Among them, special efforts are needed to improve the efficiency of memory access in multi-core systems, along with techniques for hardware acceleration to maximize the capacity of network protocols. A clear understanding of these advances and a subsequent holistic approach to tackle these new issues are necessary since they are not available in the existing software systems.

6.2.1 Memory Access in NUMA Multi-core Systems

The stubborn speed disparity between the CPU and memory, named the "Memory Wall", common in the previous single-core architecture era, will continue to exist and even deteriorate with multi-core architecture. As detailed in [3], latency in memory access will be a major bottleneck in the computer system. Pursuing higher CPU frequency is not sustainable due to the power wall: increasing transistor current leakage leads to uncontrollable power consumption and generates excessive heat that is hard to dissipate. From system architecture aspect, memory latency might partially negate a high CPU clock rate and the associated computing power. As a result, chip designers might well turn to exploring multi-core architectures and pack more cores into a single CPU die. Consequently, the speed imbalance between fast-growing number of CPU cores and memory will become more severe in the multi-core architecture.

The state-of-the-art NUMA architecture introduces a non-uniform hierarchy of memory latency. Most operating systems often provide only standard scheduling methods and shift to applications the burden of NUMA-related scheduling and tuning. Within this paradigm, applications with high performance re-

quirements must be aware of the physical locations of main memory and even peripheral devices, and implement location-aware mapping functions to co-schedule CPU cores, memory, and devices for application threads with the overall goal of reducing the latency and increasing bandwidth in memory access. The NUMA architecture is not proposed to overcome the memory wall problem. However, it offers applications a hardware platform so as to improve their aggregate performance in a multi-core environment via a suitable policy of memory allocation.

6.2.2 Protocol Offloading

Another technological advance is the hardware protocol offloading to reduce the processing cost of network protocols in computers. For example, there are at least two memory copies for each data packet sent/received by TCP/IP applications. One is between applications and operating system (kernel), and the other is between operating system (kernel) and network interfaces. For high performance computing, data copies limit a system's overall performance due to inefficient utilization of memory bandwidth and high CPU consumption. Recently, the bandwidth for a single network adapter has reached 40, 56, or even 100 Gbps [76]. Furthermore, a high-end server is often equipped with multiple adapters for load balancing and fault tolerance. Thus, traditional TCP/IP applications may hit the memory wall problem due to the performance penalty resulted from multiple data copies long before reaching the limit of bare-metal network bandwidth. Consequently, adding network capacity does not improve the actual data transfer performance.

For end-to-end data-transfer systems, both back-end storage network and front-end data movement components must reduce data copy operations and avoid the associated performance penalties. The RDMA protocol and its zero-copy techniques efficiently satisfy this requirement since it offloads network

protocol processing directly into hardware and avoids data copies from/to the kernel space. For example, to build a back-end storage system using SAN, Dalessandro *et al.* [40] implemented iSCSI extensions for RDMA (iSER) [37].

6.2.3 A Motivating Experiment

To illustrate the importance of the aforementioned technology advances to data transfer applications, we describe a simple experiment carried out in our testbed with multi-core NUMA technology. Two IBM X3650 M4 hosts are connected by three pairs of 40 Gbps RoCE connections (RDMA over Converged Ethernet). Each RoCE adapter is installed into an eight-lane Peripheral Component Interconnect (PCI) Express 3.0 slot. The theoretical maximum bandwidth of the bi-directional network of such a system is 240 Gbps.

First, we measured the maximum memory bandwidth of our hosts. We compiled STREAM [69], the de facto memory bandwidth benchmark, with the OpenMP option enabled to support multi-threaded test. The *Triad* function showed that the peak memory bandwidth for two NUMA nodes is 50 GB/s, or 400 Gbps. For socket-based network applications, there are two data copies for each network operation at each end of a TCP/IP session. Therefore, the maximum TCP/IP bandwidth that the system can support is 200 Gbps.

Then, we tested TCP/IP stack performance via *iperf* [61] to assess the maximum bi-directional end-to-end bandwidth offered by this testbed. With the default setting, iperf uses only a small chunk of memory, and reuses the same data in the memory chunk. Since the data is always cached within CPU, and it avoids one memory read access. Under these conditions, the result of iperf's performance matches that of RDMA-based data transfer because it has the same number of memory accesses as RDMA. However, such a test does not reflect real data transfer applications that need to continuously refill data from memory and back-end storage. To eliminate this cache effect, we purposely enlarged the

sender's buffer to exceed the size of the CPU cache. Since iperf is lightweight in user space, and it only transfers memory data to or receives it from network interfaces, most CPU cycles are spent on processing the TCP/IP protocol stack in the kernel space. We captured the percentage of CPU cycles in the kernel through *perf* [77], a Linux kernel profiling tool. During a ten-minute test with the Linux default scheduling policy, the average aggregate bandwidth was 83.5 Gbps. The kernel space and the user space memory copy routines, viz., the *copy_user_generic_string*, consumed about 35% of the overall CPU usage.

For comparison, we optimized "iperf" by tuning the NUMA locality, and repeated this same test. The aggregate bandwidth increased to 91.8 Gbps, about 10% higher than the previous iperf test with the default Linux scheduler.

The experiment and results afford two observations. First, the TCP/IP protocol stack requires multiple data copies and incurs a significant amount of processing overhead that further complicates multi-core memory access, and increases the severity of the memory wall problem. Consequently, the bottleneck of an end-to-end path is host processing operations, rather than network bandwidth. Second, the NUMA memory access incurs additional hardware (CPU cores) cost; for example, latency for synchronization with remote cores can further aggravate the ensuing bottleneck.

The main objective of our work thus is to carefully design an end-to-end data transfer system to eliminate the bottlenecks along a data path. These bottlenecks can comprise transferring hosts, back-end storage systems, and front-end host-to-host network communication channels. We have designed, implemented, and evaluated our RDMA-based system for wide-area data intensive applications [21, 22]. We are aware of several other studies [33, 64, 78] on integrating the RDMA capability into data-transfer applications and evaluating the resulted systems. However, those studies have not yet been validated along the entire end-to-end path, including high performance back-end systems and wide-area network links.

6.3 Characterization of System Design and Network Application

In this section, we describe the design of our end-to-end data transfer system. It encompasses one back-end storage system designed as a storage area network, one pair of sending and receiving front-end hosts, and a data transfer application over the entire infrastructure. We detail each component and analyze their performance.

6.3.1 Back-End Storage Area Network Design

We use the iSER protocol for data communication between a pair of the front-end client and back-end storage server within a storage area network. In this protocol, we follow the definitions in the iSCSI architecture, and call this pair of client and server "initiator" and "target", respectively. An initiator starts the data transfer process by sending I/O requests to the target that then proactively transfers the data. For example, to handle a read block I/O request sent by the initiator, the target will compose an RDMA *Write* work request to send data to the initiator, while a write I/O request triggers an RDMA *Read* from the target to fetch data from the initiator.

The default target process has a multi-threaded implementation that takes advantages of multi-core architecture to handle multiple I/O requests simultaneously to assure a high throughput. However, with the default setting, the NUMA factor and locations of the Peripheral Component Interconnect (PCI) devices are not considered. There are two possible methods to integrate the NUMA technology into a target. One is to use the *numactl* utility to bind a dedicated target process to each logical NUMA node; the other is to integrate the *libnuma* [71] programming interface into the target implementation. The former needs an explicit, static NUMA policy, while the latter relies on scheduling algorithm for

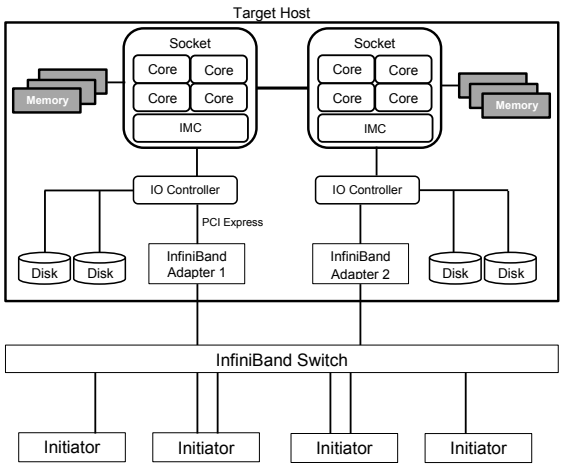


Figure 6-1: iSER tuning in NUMA architecture with multiple adapters.

each I/O request. Redesign of iSCSI with the *libnuma* API and libraries is beyond the scope of this chapter. We only implement the former solution and here present its effect on NUMA.

We use Linux *tmpfs* as the back-end storage in our prototype system. By adjusting the location of the memory file with the *mpol* and *remount* options [79], we pin each file into a specified NUMA node memory. Thereafter, we assign each NUMA node to a dedicated target process to handle local I/O requests. Therefore, all NUMA nodes have low latency in accessing local memory, and thereby, the best throughput performance. For a system with multiple adapters, as shown in Figure 6-1, this choice of design ensures that different I/O requests are handled via different links, hence resulting in the best aggregate performance. The effectiveness of this design will be evaluated by our experiments later.

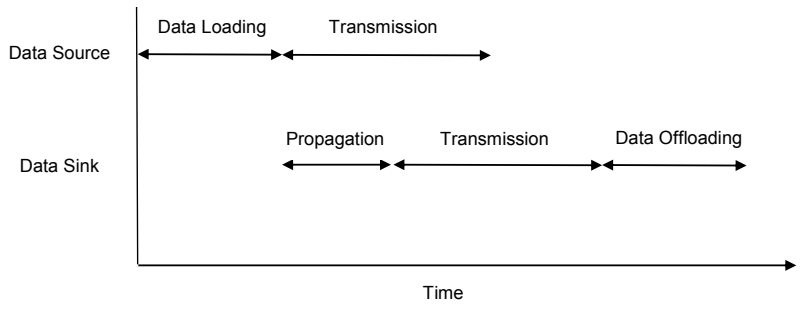


Figure 6-2: Data block transfer delay breakdown.

6.3.2 RDMA Application Protocol: Cost Analysis and Implementation

Three major components are involved in an end-to-end data transfer application: Data loading, data transmission, and data offloading. Figure 6-2 shows these components at data source and data sink. Depending on the type of data storage (such as local disks and SANs) and transmission networks (LAN and WAN), the throughput and latency of each component may vary. Any one of the three components can become a bottleneck.

We use our RDMA-based file-transfer protocol, RFTP, to move data within our system. RFTP supports pipelining and parallel operations and configures itself efficiently to utilize system resources and raw network bandwidth. To confirm the benefits of RFTP's performance, we break down its cost and compare it with the traditional TCP-based data transfer protocols.

To gain insights into the performance of data transfer applications and the efficiency of protocol offloading, we undertake a five-minute test in our local test environment. The data source loads data from /dev/zero, and then transfers it over a 40 Gbps RoCE link to the data sink. The latter will dump data into /dev/null, i.e., simply discard the received data. Both RDMA-based RFTP and TCP-based iperf accomplish this task at the transfer rate of 39 Gbps. To attain RFTP's the resource usage of each data transfer thread, we call Linux `getrusage` interface on both the client side and server side. We again analyze

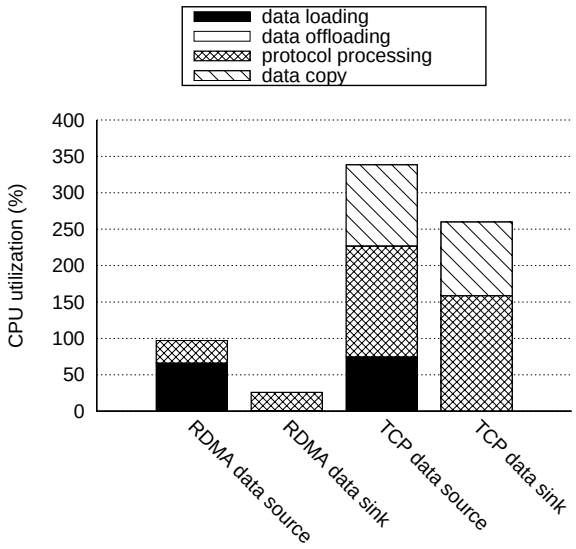


Figure 6-3: The breakdown of data transfer cost at 40 Gbps rate.

the cost of iperf data transfer using the Linux perf tool.

As shown in Figure 6-3, our RDMA based solution consumes a total of 122% CPU, among which the user space protocol processing uses 56%. [1] In contrast, TCP needs a total of 642% CPU consumption; its kernel space protocol processing accounts for 311%. RDMA's data-copy overhead is 0% because of zero-copy, while TCP pays 213% on copying data between user space and kernel space. Since the data sources load data from /dev/zero, the kernel must flush the user memory block with 0s without involving any DMA. In both the RDMA-based and TCP cases, the data sources require about 70% CPU cycles of one core to accomplish the task. Dumping data into /dev/null involves less than 1% overhead for the RFTP, while iperf does not offload data. In both cases, the overhead from offloading can be omitted in this experiment. To summarize, through this cost evaluation, RDMA demonstrates its efficient protocol offloading and zero-copy in high-speed data transfer environment.

[1] Note, we use absolute CPU time in the measurement, for example, 122% CPU consumption means the total CPU usage is equivalent to 1.22 $\times$ one fully utilized CPU core.

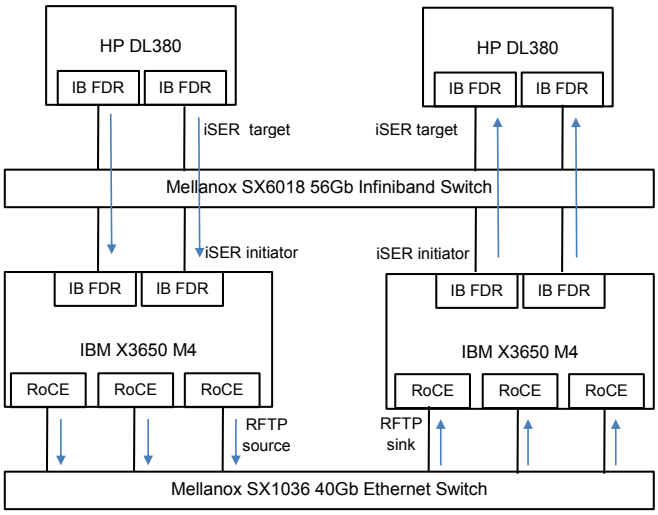


Figure 6-4: RDMA-based end-to-end system connectivity in LAN.

6.4 Experimental Results

In this section, we validated the end-to-end data throughput of our software and its CPU consumption, and experimentally confirmed the effectiveness and efficiency of our innovation of NUMA awareness and RDMA hardware offloading. We undertook comprehensive experiments on both the LAN and WAN testbeds. We first describe the testbed's configurations that consist of the RDMA implementation with both InfiniBand and RoCE interconnection. We evaluated our developed system in three different scenarios: First, the back-end system's performance with NUMA-aware tuning; second, the application performance in an end-to-end LAN setting; and third, the network performance over a 40 Gbps RoCE long distance path in wide-area networks.

6.4.1 Testbed Setup

As shown in Figure 6-4, the LAN experimental system consists of back-end and front-end sections. At the back-end, each initiator or target has two Mellanox InfiniBand adapters, each of which is fourteen data rate (FDR, 56 Gbps) and connected to a Mellanox FDR InfiniBand switch. Therefore, the maximum

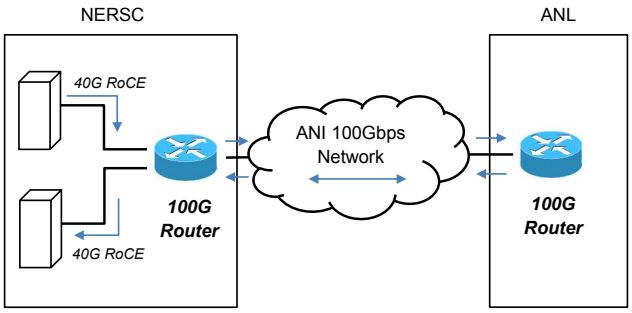


Figure 6-5: The DOE's ANI 40 Gbps RoCE WAN between NERSC and ANL. This 4000-mile link is a loopback network from NERSC to ANL and then back to NERSC. The RTT of the link is about 95 milliseconds.

bandwidth for loading and offloading data is 112 Gbps. At the front-end, three pairs of quad data rate (QDR) 40 Gbps RoCE network cards connect the RFTP client and server with a maximum aggregate bandwidth of 120 Gbps.

In our iSER software setup, we deployed open-iscsi utility version 2.0-872.41 at the initiator host, and SCSI target daemon with version 1.0.31 on the target host. As discussed in the previous section, the target incurs the largest fraction of the cost of the iSER protocol processing among all the hosts in our iSER configuration. We first investigated the characteristics of the processes in the target host, including the impact of the NUMA's binding configuration and the block size of I/O requests.

Initially, we attempted to set up a back-end storage based on Fusion IO's PCI-based SSD flash drives. However, we found that when applications read or wrote 100 gigabytes data or more continuously to the SSD drive, the thermal-throttling technology of SSDs proactively took actions to throttle the system's performance to prevent overheating the on-board circuits. These preventive operations degraded the I/O's performance to about 500MB/s, a severe bottleneck that made our performance evaluation impossible at the speed of 100 Gbps. Thus, in our experiments, we built back-end storage in the main memory of target hosts that dissipates heat much quickly, and so performs consistently over a

Table 6.1: End-to-end testbed configuration

	Front-end LAN	Back-end LAN	Front-end WAN
CPU * Cores	Intel Xeon E5-2660 2.20GHz 16 Cores	Intel Xeon E5-2650 2.00GHz 16 Cores	Intel Xeon E5-2670 2.90GHz 12 Cores
NUMA nodes	2	2	2
Mem(GBytes)	128	384	64
Network Adapters	40 Gbps RoCE QDR	56 Gbps IB FDR	40 Gbps RoCE QDR
OS	CentOS 6.3	CentOS 6.3	Fedora release 17
Kernel Version	2.6.32-279	2.6.32-279	3.4.3-1
OFED Version	MLNX OFED 1.5.3-3.1.0	MLNX OFED 1.5.3-3.1.0	OFED 1.5.4
TCP Congestion Control Algorithm	cubic	cubic	cubic
MTU Size	9000 (RoCE link)	65520 (IB link)	9000
RTT(ms)	0.166	0.144	95

wider range (from air-conditioned data centers to laboratory environment with normal temperature) of operational temperature. We created six logical units (LUNs) with on the target host, split and load-balanced all I/O requests between the two available InfiniBand links. Each LUN's size was 50 gigabytes, and the total size of the dataset was 300 gigabytes. The large size memory (we borrowed 768G byte dual in-line memory modules from HP vendors for building the high performance backend storage in our testbed) can be formatted to host any data files and represent a real-world storage solution with low latency and high throughput.

In addition to the testbed configuration within the local network, we also utilized the WAN testbed provided by the DOE's Advanced Networking Initiative (ANI). For these tests, the DOE's ANI supplied a 40 Gbps RoCE wide-area network to evaluate data transfer with RFTP over a long-haul 40 Gbps link. This 4000-mile link is a loopback network from the National Energy Research Sci-

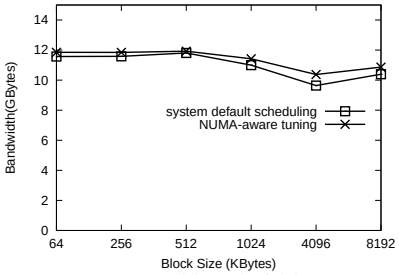
entific Computing Center (NERSC) in Oakland, CA to Argonne National Laboratory near Chicago, IL and then back to the NERSC. As shown in Figure 6-5, the two hosts are connected to wide area networks by a 100 Gbps router, the Alcatel-Lucent Model SR 7750 border. The corresponding routing records on NERSC router are configured as a loopback via ANL's 100 Gbps router [80]. Table 6.1 lists the detailed configurations for all the hosts, in both the LAN and WAN testbeds described.

6.4.2 Evaluation of Memory-Based Storage System Performance

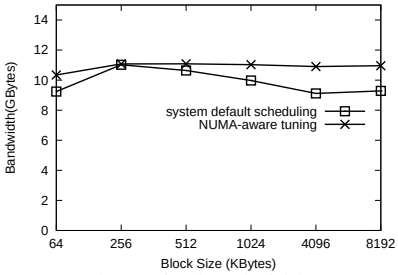
In this set of experiments, we evaluated the improvement in the iSER's performance with our NUMA-aware tuning policy and compared it with the Linux's default scheduling policy.

The iSCSI target host (the back-end storage) is based on Linux tmpfs file system created out of the system's main memory to eliminate the inefficient magnetic disk or SSD bottleneck. The system's memory is large enough (300 GB) to be comparable to a regular SAS disk in terms of the volume, whilst offering a performance a hundred of times faster than that of a magnetic disk. We created six logical units to spread parallel IO requests into different banks of the main memory. To minimize the overhead from the file system, the target exported the back-end memory file as raw devices to allow the initiator to choose any type of file systems as appropriate. We choose the flexible I/O tester [32], fio, as the benchmark software, and each test case lasted five minutes.

Figures 6-6 and 6-7, respectively, show the bandwidth and CPU consumption, in these tests. In each figure, we separately illustrate the data read and data write performance. To achieve the best performance, multiple I/O threads run simultaneously against each LUN. The gain in performance levels off once the number of threads reaches a certain threshold. Beyond that, too many I/O



(a) Read bandwidth



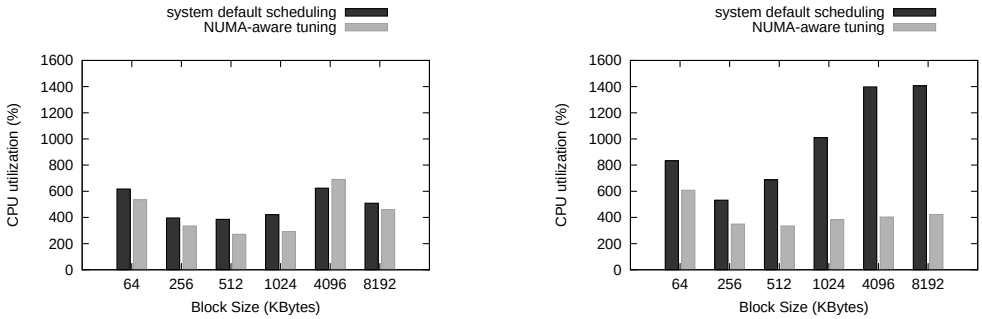
(b) Write bandwidth

Figure 6-6: iSER bandwidth comparison between default scheduling and NUMA-tuning.

threads would introduce more contention, and impact the overall performance. In our testbed, we found that the optimal configuration is to use four threads for each LUN.

For read operations, the bandwidth improvement is merely 7.6% with NUMA binding, and neither is the saving on CPU consumption significant. However, for write operations, we observed an improvement in bandwidth up to 19% for the block size larger than 4 megabytes, and using the default Linux binding policy increases CPU consumption threefold. We argue that the main reason of this disparity between read and write operations is the significant overhead of cache coherency and synchronization that is needed for write operations. We note that we performed the read and write operations on a memory-based tmpfs file system. A write request essentially is a memory-write operation, and if it is executed without NUMA-aware tuning, one such operation will invalidate all other data copies in the caches at other NUMA nodes. With NUMA-aware tuning, this invalidation occurs only locally and thus, the overhead is low. When read requests are executed, with or without NUMA-aware tuning, the data copies are always "cached" or "shared" instead of "modified", and hence, the overhead from cache coherency is minimal.

We also notice that the bandwidth performance of serving read requests out of iSER is slightly better by 7.5% than that of serving write requests. We believe



(a) Read CPU utilization

(b) Write CPU utilization

Figure 6-7: iSER CPU utilization comparison between default scheduling and NUMA-tuning.

this reflects the better performance of RDMA Write (used by read requests) than RDMA Read (used by write requests). This difference in performance between RDMA Read and Write was revealed in a previous study [22]. When an iSER initiator sends a read request to a target, the target would use RDMA write to write (send) data directly to the memory of the initiator for the actual transfer of data; thus, the observed performance of a read request has a better bandwidth.

6.4.3 End-to-End Data Transfer Performance

In this section, we describe our tests and our validation of data-transfer applications in an integrated testbed environment and gather its performance with an end-to-end perspective. Our goal is to gain insight into how our application can be adapted to day-to-day real transfer scenarios.

In mimicking a realistic data transfer scenario, we adopted one wherein a transfer application first interacts with file systems via the POSIX interfaces rather than via raw devices. The details of implementing the functions of different file systems are hidden by the POSIX interfaces. Furthermore, our preliminary tests in our testbed demonstrated that the throughput differences among the raw block devices exported by target via the iSER protocol, Linux universal ext4 file system, and the XFS [81] built over the exported block devices via the

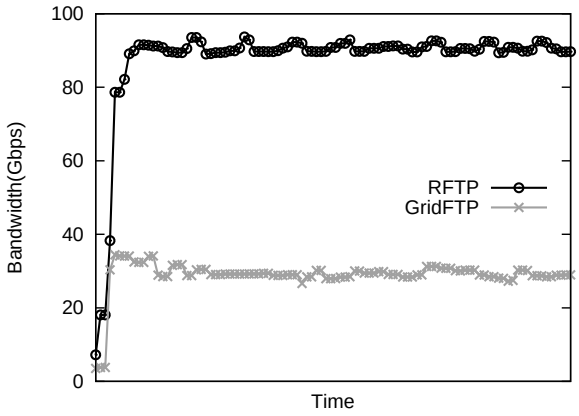


Figure 6-8: Throughput of end-to-end data transfer over 25 minutes.

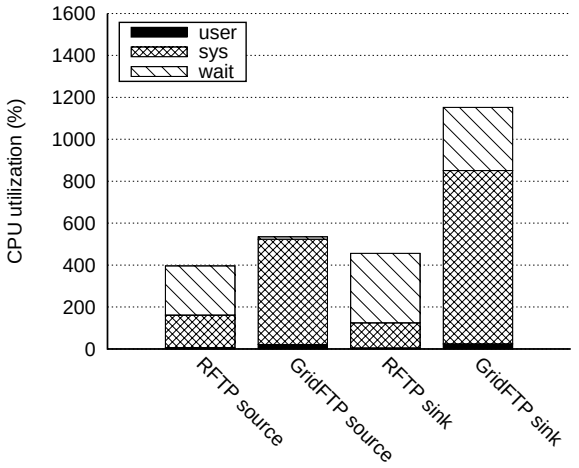


Figure 6-9: CPU utilization breakdown for RFTP and GridFTP.

iSER protocol, are comparable. Since the XFS file system particularly is efficient for parallel I/O and better aligned with our testing requirements, without losing generality, we chose XFS over other file types and formatted the exported block device with XFS from the initiator side.

We combined the back-end system and front-end system to show the end-to-end performance between RFTP and GridFTP, a widely used data transfer tool in high performance computing. To assure a fair comparison between GridFTP and RFTP, and to assure the achievement of the best performance of both applications, and minimize the penalty of remote memory access for each of them,

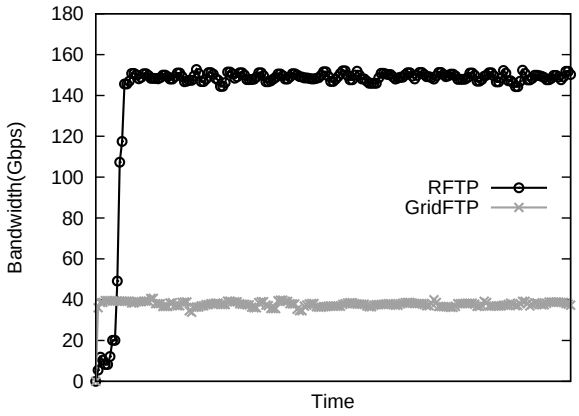


Figure 6-10: Throughput of bi-directional end-to-end data transfer over 50 minutes.

we used numactl to bind the RFTP and GridFTP processes to a specified NUMA node. Figure 6-8 shows the two applications' performance during 25-minute-long tests. Our prior "fio" tests revealed that the narrowest section along the end-to-end data transfer path resides on file I/O write operation; its performance is 94.8 Gbps. Therefore, the best performance of the testbed for end-to-end data transfer is 94.8 Gbps. RFTP achieved 91 Gbps, i.e., 96% of the effective bandwidth of the overall system, while GridFTP obtained 29 Gbps, i.e., only 30% of the bandwidth for the following reasons. First, TCP stack processing incurs a substantial overhead, such as data copy between kernel space and user space and interrupt handling for an I/O-intensive application. The high "sys" CPU in GridFTP, shown in Figure 6-9, reveals the high TCP/IP stack-processing cost of GridFTP. Second, GridFTP has a single-threaded design that causes the network to be in an idle state when this thread performs I/O request with long waiting time. Consequently, GridFTP is not able to fully utilize the resources of the whole I/O system. Running multiple processes simultaneously may alleviate this problem, but at the price of higher CPU consumption. Third, without support for direct I/O, GridFTP suffers the I/O cache effect at the front-end sender and receiver hosts.

To further clarify the best end-to-end host performance in terms of band-

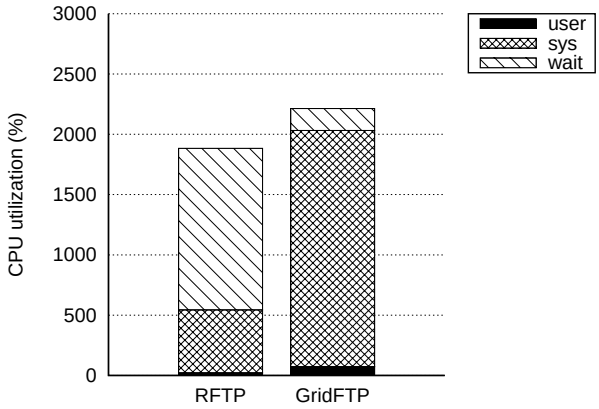


Figure 6-11: CPU utilization breakdown for RFTP and GridFTP bi-directional test.

width and CPU consumption, we performed bi-directional data transfer tests with RFTP and GridFTP. In this experiment, we initiated data transfer simultaneously from each end of the end-to-end path to the other end. The configurations are the same as in the previous experiment. We expected that the aggregate bandwidth in the bi-directional tests would have been twice the performance in the unidirectional experiment due to the full duplex property of each component in the transfer path. However, the experimental results did not match with our expectation. Contention for resources increased for bi-directional tests because of more intensive parallel I/O requests to the back-end hosts, memory copies, and higher protocol processing overhead at the front-end hosts.

As shown in Figure 6-10, our RFTP demonstrated an impressive 83% bandwidth improvement in the bi-directional experiments versus the unidirectional ones, and almost doubles the unidirectional performance (17% less). On the other hand, GridFTP demonstrated only about a 33% improvement due to its high CPU contention, as shown in Figure 6-11.

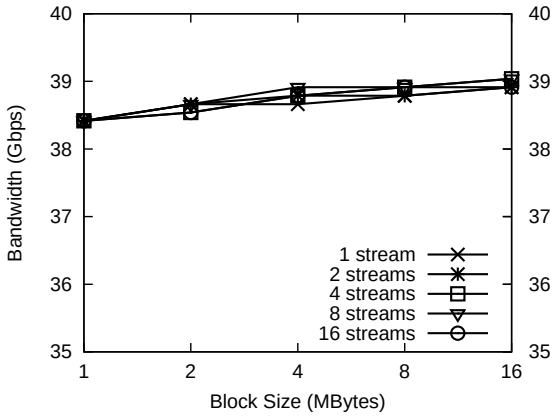
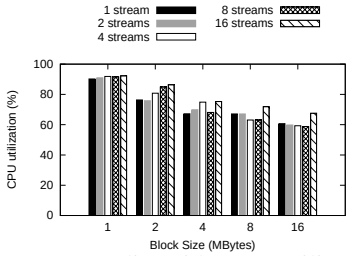


Figure 6-12: RFTP bandwidth with various block sizes and numbers of streams.

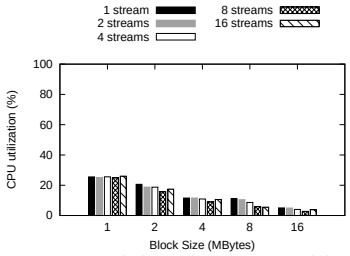
6.4.4 Experimental Results over 40 Gbps WAN RoCE Link

Long-haul fat links introduce much higher latency than local area networks, and have a large bandwidth delay product (BDP). It is challenging for traditional network protocols (between front-end hosts) to fill up the network pipe at the speed of 100 Gbps and beyond. Here, we evaluate the effectiveness of RFTP in eliminating this bottleneck effect. We ran RFTP over a RoCE link in DOE's ANI testbed. This link has an RTT of about 95 milliseconds, and the BDP is close to 500 megabytes. We cannot relocate our entire testbed system to the point of presence (POP) site of the DOE's ANI testbed for a suite of full WAN tests due to the restriction in testbed management and administration in a remote data center. We had to leverage the existing end systems provided by the DOE's ANI testbed. For our experiments now, we only can conduct memory-to-memory data transfers (between front-end hosts) to show that our RFTP affords a good solution to support end-to-end data transfers in the wide-area networks. We expect that if RFTP performs well over the RoCE link, then our full end-to-end data transfer system would perform equally well if it were deployed in the ANI testbed.

Figure 6-12 shows the bandwidth performance of RFTP when we use different numbers of parallel data streams. The x-axis shows the block size of



(a) RFTP sending side CPU utilization



(b) RFTP receiving side CPU utilization

Figure 6-13: RFTP CPU utilization with various block sizes and numbers of streams.

data transferred, while the y-axis shows the actual payload data transfer bandwidth, excluding the protocol overhead. We verify that RFTP utilizes 97% of the raw bandwidth of the testbed link due to its efficient design (including the use of proactive feedbacks and asynchronous control message exchanges [22]). A small processing overhead is needed for control messages (for example creating RDMA channels and passing credit tokens), and this overhead decreases in a direct relation to increase in the message block size. The overhead reduction also is reflected by the lower CPU consumption as shown in Figure 6-13(a) for the sender, and in Figure 6-13(b) for the receiver.

As we noted earlier that this wide area data transfer did not involve storage area networks due to the difficulty of deploying and managing them remotely at the NERSC. Our prior experiments show that we can get the maximum performance along each segment of end-to-end LAN and WAN data transfers, i.e., back-end data upload to the front-end system, data transfer over networks, and data offload to the back-end storage system again.

Furthermore, the performance of network data transfer is unaffected by long latency. Based on these observations, we conclude that our RFTP easily can scale up the performance that is commensurate with full-fledged end-to-end distributed testing or production infrastructures consisting of large-scale storage area networks with hundreds or thousands of target servers and long latency inter-data center network links. These infrastructures often are found at the

DOE's National Laboratories and cloud data centers with intensive data transfer requirements.

6.5 Conclusions

Modern data centers of scientific computing must transfer and synchronize a large amount of data continuously either locally within themselves or remotely with other data centers for visualization, analysis, and disaster recovery. Accordingly, end-to-end high performance data transfer software must combine efficient design and performance tuning, to eliminate any potential bottlenecks within storage systems, at front-end hosts, and along network communication paths. In this chapter we have described a novel design of such a system that integrates RDMA protocol implementation, multi-core NUMA tuning, and an optimized back-end storage area network.

To demonstrate the efficiency of our solution, we set up testbeds in both LANs and WANs. We studied the processing cost of TCP/IP stack and our RDMA based protocol. We demonstrated the performance benefits of an RDMA based solution adopted by both our RFTP and iSER. We also compared our solution with the high performance GridFTP software in various end-to-end configurations. Our performance evaluation demonstrated that our solution is three times faster than GridFTP. We also validated our protocol design in a 4000-mile network path provided by the Department of Energy's ANI testbed. These evaluations and studies verified that our solution can achieve remarkable bandwidth utilization of 97% and fully utilize the available hardware capabilities.

Chapter 7

Related Work

This chapter surveys the research efforts and real system implementations that are related to this dissertation. We first introduce the literature of software architectures and frameworks for processing tasks with high concurrency. Then we discuss existing RDMA-based and TCP-based high performance data transfer software. We also review recent works on storage caching management.

7.1 Software Architecture for Highly Concurrent and Asynchronous Data Processing

Asynchronous I/O is widely used in high performance computing (HPC) to reduce communication overheads and latencies and Major Operating System and software utilities support asynchronous I/O: for example, Linux OS has been offering a set of system calls for asynchronous I/O since the early Linux version. Recently, Linux started to distribute OpenFabrics [82] that implements and supports asynchronous network transport. Several recent research and development efforts aim at supporting asynchronous direct file system I/O. For example, Oracle supports both asynchronous I/O and direct I/O and allows the caller function to continue processing while sending non-blocking I/Os requests. Such a con-

current processing leads to a higher I/O performance [83] than a synchronous one.

Recently, asynchronous network I/O gained popularity beyond HPC applications. For example, Mitchell, et al. utilized asynchronism and zero-copy kernel bypass of RDMA to implement an efficient key-value store called Pilaf [84]. To ensure a good trade-off between the high-performance RDMA operations and the complexity of memory synchronization, Pilaf restricts clients to only use RDMA for read-only *get* operations and bypasses the server's CPU. Clients need to explicitly submit/post the *put* operations, and the server sequentially handles each write operation in the same way as a regular key-value store does. The separate processes of *get* and *put* clearly improve the read performance in terms of operations/sec per core when *get* is the dominant workload. In [56], a high performance RSA and a SSL proxy (called SSLShader) offload computing-intensive cryptographic operations to graphics processing units (GPUs). This enables multiple threads for cryptographic operations to work in parallel and achieve 92K operations/sec for 1024-bit RSA and 13 Gbps encrypted data streaming throughput with a single GPU card. GPU is optimized for the computing-intensive tasks of access-once-process-many (i.e. CPU-bound), its throughput is limited by the PCI bus and the high communication overheads between the GPU and its main processor and memory. The newest GPU card (NVIDIA TESLA K80 GPU ACCELERATOR) supports 16-lanes PCI Express gen3 connection which has a maximum theoretical speed of 120 Gbps [85]. Actual performance is much lower than that because of the high communication overhead over PCI bus, as shown in [56].

Asynchronism implemented with an event-driven architecture is particularly attractive to the web service design because event-driven asynchronous computing processes a large number concurrent requests and enables high throughput. SEDA (a staged event-driven architecture) was proposed in [59] to provide efficient, scalable I/O interfaces and adaptive resource scheduling. The architecture

was implemented with two use cases: a high performance HTTP server and a packet router for the peer-to-peer file-sharing network of Gnutella [86]. SEDA demonstrated higher throughput and better stability than those traditional designs. The authors in [87] presented another portable Web server architecture, asymmetric multi-process event-driven (AMPED), and its implementation of a flash web server. The flash server outperformed Zeus Web server by up to 30%, and Apache by up to 50% with real web workloads.

Several research efforts studied the merits and shortcomings of event-based concurrency and thread-based concurrency for the web applications with high concurrency requirements [88–90]. These efforts concentrated on programming language and OS support. The performance improvement in peripheral devices recently gave rise to the new opportunities of developing suitable asynchronous and concurrent methods and harnessing the abundant throughput. Nevertheless, we expect that no single policy fits and integrates all types of I/O behaviors and computation functions, and a hybrid method is needed to entail a wide range of hardware advances.

7.2 High Performance Data Transfer Protocol and Software

The remarkable performance advantages of RDMA technology for data center networks and high performance computing have attracted a great amount of interests from academia and industry. The original RDMA system, known as InfiniBand (IB) [9], supports a top-down RDMA message service with its own implementation of layer two to layer four protocol (sometimes including layer-1) of the OSI stack. It provides a message passing service to applications and offloads all protocol processing operations to specialized hardware. Different from the best-effort frame delivery service in Ethernet, the link layer of Infini-

Band offers reliability and maintains packet order through its credit-based flow control and virtual lane mechanism. However, extending IB to WAN requires proprietary hardware to encapsulate IB in an Ethernet frame. This limitation restricted its wide adoption in today's Internet.

Two other implementations, Internet Wide Area RDMA Protocol (iWARP) and RDMA over Converged Ethernet (RoCE), were proposed to extend the advantages of RDMA to the ubiquitous IP/Ethernet-based networks and Internet. iWARP offloads the whole TCP/IP stack to the network adapter. The Direct Data Placement (DDP) layer of the iWARP stack implements and supports zero-copy and kernel-bypass; it transfers data in the user-space buffer directly to the application memory on the remote server. iWARP enables RDMA to seamlessly and transparently run over the best effort IP networks, a.k.a., the Internet. RoCE supports the IB transport protocol over Ethernet and offers the advantages of IB, such as high bandwidth and low latency IB, in an Ethernet environment. RoCE is a natural extension of message-based data transfer, and therefore, of the two alternatives to IB (iWARP and RoCE), offers better efficiency [11], and is increasingly adopted by modern data centers to consolidate and converge heterogeneous networks for storage, supercomputer interconnects, and cluster/cloud communication fabrics.

One objective of our design is to support applications across all these RDMA architectures. In RDMA architecture, a verb defines how an application requests action from the messaging service [91]. We build our system with the common Verb Application Programming Interface (API) of the OpenFabrics Enterprise Distribution (OFED) [82] that is a unified, cross-platform, transport-independent software stack for RDMA. OFED offers a uniform application programming interface, known as native IB verbs, to access various RDMA architectures and functionalities. Applications mainly use the *libibverbs* and *librdmacm* libraries. OFED software also offers several middleware packages, such as IP over IB [92] and Sockets Direct Protocol (SDP) [10], to allow the

socket-based applications to run over the RDMA devices without the tedious reimplementation of their original software programs. Other studies [93, 94] chose to use the User Direct Access Programming Library (uDAPL) [95] that provides simple APIs to manage and control the RDMA capabilities. Nevertheless, these extensions introduce additional overhead and performance penalties compared to the native RDMA IB verbs [64].

Lai [64] implemented an RDMA-enabled FTP application based on the two-sided zero-copy operation of IB networks. However, the two-sided SEND/RECEIVE operations, originally proposed for delivering small control messages, can not maximize the full throughput capacity of the new generation RDMA networks shown in Figure 2-3. The one-sided RDMA READ/WRITE is a better choice for high-speed large-scale data transfer because it decouples the data transfer (data plane) entirely from the OS kernel software (control plane). Other researchers [96, 97] demonstrated that although there are some benefits of using RDMA over LAN and WAN with short latency, it is still challenging to attain good RDMA performance in WAN with a long latency because RDMA READ operation is sensitive to packet loss and latency. Based on these prior studies [31, 96, 97], our middleware is designed to exploit the full benefit of RDMA, in particular RDMA WRITE operation, yielding better performance and lower communication cost for synchronizing senders and receivers than existing RDMA applications.

Tian et al. [33] implemented a RDMA extension driver for GridFTP to utilize high-speed InfiniBand. Similar to our approach, they employed RDMA WRITE to transfer large blocks of data. However, their design is not fully optimized. In their design, data source needs one RTT to get transfer credits (tokens for available buffers at destination) from data destination. This feature forces source to wait in idle, and potentially slows down data transfer in WAN that often has a large RTT. Moreover, Compared to their protocol, our implementation explicitly enforces flow control between the two communicating

parties, and tries to maximally parallelize RDMA operations. Subramoni [78] also presented another driver to incorporate the capability of InfiniBand into the GridFTP framework. Panda et al., extended the RDMA technology to Message Passing Interface (MPI) [94, 98, 99] and to enable parallel applications of taking advantage of RDMA's low latency and high speed communication. This is work-in-progress as its scalability and performance need to be tested and validated in the newly available 40Gbps/100Gbps InfiniBand and Ethernet network environments.

7.3 Hardware and Software Accelerated Key Value Stores

Key-value stores and caches become popular in large data centers for serving data directly from memory. They often provide a simple set of interfaces, such as `GET`, `PUT`, and `DELETE`, for applications to improve I/O performance by keeping user data in low latency RAM. TCP-based solutions are common and mature, such as Memcached [100], Redis [101], and MongoDB [102]. Their design focuses on two topics: server indexing data structure and network data transmission based on TCP. However, these general purpose solutions often suffer long latency and low throughput as a result of the excessive network stack overhead. The problem deteriorates especially for serving a large number of small key-value pairs.

As the TCP/IP network stack is often the bottleneck, many research efforts have been focusing on bringing the zero-copy network stack into design. They often holistically consider the data indexing structure and network protocol together: server expose data index (i.e., memory locations to "value") to clients and client use RDMA READ/WRITE, along with supplied memory addresses to fetch/update remote memory blocks directly. The entire key-value process

itself follows exactly the native communication protocol of RDMA. Pilaf [84] proposed using one-sided RDMA operations to get key-value object from server directly. It requires two RDMA read operations: one for metadata and the other for data. Herd [103] improves the key-value performance by employing operations within only one round trip time to obtain a key-value pair. It uses one-sided RDMA write to send a request over reliable connections, and uses two-sided send/receive to send back the response. In addition, it uses unreliable data transmission to save the on-board address translation table. Mica [104] explores the zero-copy capability of Intel network interface for small network package transmissions.

While this dissertation focus on the design and implementation of zero-copy network protocol for wide area network, the protocol itself can be applied to key value stores that target intra-data-center workloads. The I/O caching optimization at the lower layer in this dissertation also provides noticeable benefit to key-value stores at the upper layer.

7.4 Storage Cache Performance Optimization

In Section 2.3 and Chapter 5, we introduced the NUMA architecture and presented its research challenges. Several research efforts, including ours, proposed parallel implementations to harness multi-core for improving storage throughput and reducing latency.

Joglekar *et al.* proposed two optimization methods to improve iSCSI target performance: replacing the old cyclic redundancy check (CRC) algorithm with a new one, and eliminating data copy overheads between the iSCSI layer and the TCP layer [105]. Zheng *et al.* designed a parallel page cache approach to partition global cache into many independent page sets and to eliminate locking contentions in a multi-core system [106]. A hybrid access mode was proposed for storage area networks (SAN) to cache I/O traffic on a metadata server, and

to avoid traffic between client and data servers if the requested data is already cached by the metadata server [107]. Studies on performance impacts under different workloads to the iSCSI protocol and comparisons between iSCSI and NFS revealed that the two protocols are comparable for data-intensive workloads, while iSCSI outperforms NFS by a factor of 2 or more for meta-data intensive workloads [108–110].

Chapter 8

Conclusions and Future Work

Technology advances in state-of-the-art computer hardware provide both opportunities and challenges. To achieve bare-metal performance and fully utilize these hardware advances, data I/O systems and middleware must undertake careful new design and performance tuning, to tackle various scalability problems along the entire end-to-end data path, and to eliminate any potential bottleneck inside storage systems, at front-end hosts, and along network communication paths. In this dissertation, we detail our solution on scalable systems design and implementation, including zero-copy based data transfer protocol over WAN and NUMA-aware caching solution for iSCSI/iSER systems over high-speed Interconnects. This chapter concludes our research and contributions, and proposes the future directions based on the thesis work.

8.1 Conclusions

Modern data centers transfer and synchronize an unprecedented amount of data either locally inside themselves or remotely to other data centers for visualization, analysis, and disaster recovery. This new trend engenders the need for scalable data I/O solutions. In this dissertation we demonstrated the system design for scalable end-to-end data I/O including RDMA-based data transfer

protocol for high performance networks, multi-core NUMA-aware tuning along the whole data path, and highly optimized NUMA-aware caching for storage area network.

We presented the Asynchronous Concurrent Event-driven Staged (ACES) software architecture for high throughput data I/O systems. It divided the data path into stages, connected these stages with explicit queues, and orchestrated hardware resources by keeping track of each stage's status. Then we described a scalable RDMA-based data transfer protocol for WAN. To ensure a high-degree zero-copy in data transfer, we adopted the memory-centric design principle. To cope with the complexity of the design, we introduced finite state machine to model the memory status, and reserved dedicated control path to deliver control messages and to synchronize memory status between the communication parties. We also elaborated the connection management and the metadata format for data transfer. We designed an end-to-end data transfer software, RFTP, on top of this protocol, and implemented the software system based on the ACES software architecture.

For storage I/O, we studied the scalability of iSCSI/iSER serving cached data. We analyzed the I/O cost, latency and throughput, of the iSCSI/iSER protocol with and without NUMA-aware tuning. We designed and implemented a userspace NUMA-aware cache to replace the default OS page cache for high performance storage systems. Furthermore, we introduced a decentralized event processing model and scaled up its performance of processing many small-scale requests. We designed the caching-layer optimization for accessing data blocks to be universal so to expedite more application with cache, such as file systems and databases.

To demonstrate the efficiency of our solution, we set up testbeds in both LANs and WANs. We studied the processing cost of conventional TCP/IP stack and our RDMA based protocol. We demonstrated the performance benefits of RDMA that was adopted by RFTP and iSCSI/iSER. We also compared RFTP

with the high performance GridFTP software in various end-to-end configurations. Our performance evaluation demonstrated that our solution was 2× faster than GridFTP. We also validated our protocol design in a wide area network network path belonging to the Department of Energy's ANI testbed. These evaluations and studies verified that our solution achieved a remarkable 97% utilization of bare-metal network bandwidth, and fully utilized the available hardware capabilities. In addition, we evaluated our NUMA-aware cache solution on a 4-node NUMA testbed, and showed its sizeable performance improvement over OS page cache for both synthetic and real-life workloads.

8.2 Future Work

In this section, we present future directions for high-speed data I/O that are inspired by our research work.

8.2.1 Efficient I/O for High Speed Parallel Hardware Accelerator

PCI-based (Peripheral Component Interconnect) computing accelerators, such as GPGPU and Intel Coprocessor, extend single server processing capacity by partitioning data/tasks into small chunks and offloading them to many parallel cores of accelerator. These cores often need to directly exchange data and synchronize with other servers that are connected by high performance networks in a distributed system. In this work, we focuses on the data that reside in main memory. However, the data I/Os by offloaded computing involve on-board RAM, and require special fine-grained accesses to eliminate data copies between main memory and device RAM. For example, the GPUDirect technology bypasses main memory, and exposes accelerator's onboard RAM to network adaptor directly. The new technology enables new research in mitigating the

severe I/O bottleneck in the computation by coprocessors and improving data processing throughput in distributed systems. This future work can leverage our network protocol design and software architecture to move data among many offloading computing cores of cluster nodes within a data center and even across data centers.

In addition, the ACES software architecture also offloads some computation intensive tasks to coprocessors to harness their parallel processing capability. For example, we plan to investigate and design parallel algorithms for computation-intensive public/private key cryptography and data integrity assurance on Nvidia GPGPU and Intel Coprocessors, and integrate them into the ACES software architecture.

8.2.2 Asynchronous I/O Event Scheduling

Traditional event-driven software design consists of non-blocking routines (callback functions) that correspond to various events. There are many types of events: probing event for I/O device availability (readable/writable), I/O error, timeout, and system call interruption. To achieve better processing throughput, applications often need a scalable event scheduling algorithm for handling different types of events efficiently. Several factors need to be considered for design the scheduling algorithm: fairness, efficiency, non-blocking, and fine granularity. The optimization on I/O event scheduling will improve data intensive applications such as the data transfer software and the storage server applications presented in this dissertation.

8.2.3 Memory-based Data Backup over SAN

The memory capacity in a single Intel server increases to terabyte level recently. For example, An Intel Xeon 8800 processor connects up to 12 Terabyte main memory and the NUMA architecture further scales up the total memory capa-

bility linearly with the number of processors in a single system. Storage servers often use a large amount of memory for caching "hot data" and buffering write data. Buffering write data always causes the problem of consistency, potentially damages data integrity, undermines server stability, and even triggers service outage. On the other hand, synchronous write operation without buffering incurs much longer latency for clients. A careful trade-off is needed to reduce I/O latency while attaining data reliability. For example, I/O mirroring-on-write assures necessary redundancy, i.e., instead of writing data directly into persistent storage media, we can replicate data into the main memory of other servers by background threads and hardware offloading with RDMA. The server can safely respond to application once data replications to other servers complete successfully. Future research can focus on sorting the memory copies of buffered data and writing to their nearest persistent storage media in an optimized order. Any single point of failure will not lose buffered write data. This design assures better performance in terms of I/O latency and throughput than does the standard persistent I/O. Many cloud storage systems, such as, Google File System [111] and its open-source version of Hadoop Distributed File System [112], adopt this mirroring mechanism. However, they focus on reliability instead of performance, and do not utilize the highly efficient off-loading by RDMA.

8.3 Summary

Scalable data I/O systems, including data transfer protocol and storage caching system, are essential components in data centers. Because the bare-metal capacity of computer hardware improved significantly in the last decade, software design must cope with the hardware characteristics to ensure line-speed performance and scalability. This dissertation details the research efforts of designing and implementing the scalable end-to-end data I/O system, the RDMA-based data transfer protocol for wide area networks, and the NUMA-aware cache sys-

tem for iSCSI/iSER servers. We build real systems to incorporate these research outcomes and to show their advantages in the high performance network environment.

Bibliography

[1] Youngjae Kim, Scott Atchley, Geoffroy R. Vallée, and Galen M. Shipman. Lads: Optimizing data transfers using layout-aware data scheduling. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 67–80, Santa Clara, CA, February 2015. USENIX Association.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[3] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[4] The Magellan report on cloud computing for science. Technical report, 2011.

[5] Dennis Overbye. Physicists find elusive particle seen as key to universe, July 2012.

[6] Krste Asanovic. Firebox: A hardware building block for 2020 warehouse-scale computers. *Keynote presentation given at the 12th Usenix Conference on File and Storage Technologies*, 2014.

[7] The ASCAC Subcommittee on Exascale Computing. The opportunities and challenges of exascale computing. 2010.

[8] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. A remote direct memory access protocol specification. RFC 5040 (Proposed Standard), October 2007.

[9] InfiniBand Trade Association. InfiniBand Architecture Specification. *Release 1.2.1*, 2006.

[10] IBTA. Infiniband Trade Association. http://www.infinibandta.org/, 2010.

[11] David Cohen, Thomas Talpey, Arkady Kanevsky, Uri Cummings, Michael Krause, Renato Recio, Diego Crupnicoff, Lloyd Dickman, and Paul Grun. Remote Direct Memory Access over the Converged Enhanced Ethernet fabric: Evaluating the options. In *2009 17th IEEE Symposium on High Performance Interconnects (HOTI)*, pages 123–130, 2009.

[12] R. Tecio, P. Culley, D. Garcia, and J. Hilland. An RDMA protocol specification. *RDMA Consortium*, October 2002.

[13] An introduction to the Intel QuickPath Interconnect, January 2009.

[14] HyperTransport I/O technology overview, June 2004.

[15] Ulrich Drepper. What every programmer should know about memory. September 2007.

[16] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, pages 16–29, 2010.

[17] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating Amdahl's Law through EPI throttling. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 298–309, May 2005.

[18] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. Gpunet: Networking abstractions for gpu programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 201–216, Broomfield, CO, October 2014. USENIX Association.

[19] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[20] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[21] Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, and Thomas Robertazzi. Middleware support for rdma-based data transfer in cloud computing. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12,

pages 1095–1103, Washington, DC, USA, 2012. IEEE Computer Society.

[22] Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, Thomas Robertazzi, Brian L. Tierney, and Eric Pouyoul. Protocols for wide-area data-intensive applications: design and performance issues. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 34:1–34:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[23] Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, and Thomas Robertazzi. Design and performance evaluation of numa-aware rdma-based end-to-end data transfer systems. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 48:1–48:10, New York, NY, USA, 2013. ACM.

[24] Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, and Thomas Robertazzi. Design and testbed evaluation of rdma-based middleware for high-performance data transfer applications. *Journal of Systems and Software*, 86(7):1850 – 1863, 2013.

[25] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 277–289, Santa Clara, CA, July 2015. USENIX Association.

[26] ESnet. Energy Sciences Network: http://www.es.net/, 2012.

[27] Tan Li, Yufei Ren, Dantong Yu, and Shudong Jin. Resources-conscious asynchronous high-speed data transfer in multicore systems: Design, optimizations, and evaluation. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1097–1106, May 2015.

[28] Globus Group. GridFTP online page: http://www.globus.org/toolkit/docs/latest-stable/gridftp/, 2012.

[29] Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, and Thomas Robert azzi. Design, implementation, and evaluation of a numa-aware cache for iscsi storage servers. *IEEE Transactions on Parallel & Distributed Systems*, 26(2):413–422, 2015.

[30] Tan Li, Yufei Ren, Dantong Yu, Shudong Jin, and Thomas Robertazzi. Characterization of input/output bandwidth performance models in NUMA architecture for data intensive applications. In *Proceedings of the 2013 International Conference on Parallel Processing*, ICPP '13, 2013.

[31] Philip W. Frey and Gustavo Alonso. Minimizing the hidden cost of RDMA. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, June 2009.

[32] Jens Axboe. Flexible I/O Tester: http://freecode.com/projects/fio, 2012.

[33] Yuan Tian, Weikuan Yu, and Jeffrey Vetter. RXIO: Design and implementation of high performance RDMA-capable GridFTP, 2011.

[34] Open MP. Openmp: http://openmp.org/wp/, 2014.

[35] Linux. Kernel asynchronous i/o (aio) support for linux: http://lse.sourceforge.net/io/aio.html, 2014.

[36] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[37] M. Ko, M. Chadalapaka, J. Hufferd, U. Elzur, H. Shah, and P. Thaler. Internet Small Computer System Interface (iSCSI) Extensions for Remote Direct Memory Access (RDMA). RFC 5046 (Proposed Standard), October 2007.

[38] Fujita Tomonori and Mike Christie. tgt: Framework for storage target drivers. In *Proceedings of the Linux Symposium*, LinuxSymposium'06, pages 303–312, 2006.

[39] Fujita Tomonori and Ogawara Masanori. Analysis of iscsi target software. In *Proceedings of the international workshop on Storage network architecture and parallel I/Os*, SNAPI '04, pages 25–32, New York, NY, USA, 2004. ACM.

[40] Dennis Dalessandro, Ananth Devulapalli, and Pete Wyckoff. iSER storage target for object-based storage devices. In *Proceedings of Fourth International Workshop on Storage Network Architecture and Parallel I/Os*, September 2007.

[41] Xubin He, Qing Yang, and Ming Zhang. A caching strategy to improve iscsi performance. In *Local Computer Networks, 2002. Proceedings. LCN 2002. 27th Annual IEEE Conference on*, pages 278 – 285, nov. 2002.

[42] Jun Wang, Xiaoyu Yao, C. Mitchell, and Peng Gu. A new hierarchical data cache architecture for iscsi storage server. *Computers, IEEE Transactions on*, 58(4):433 –447, april 2009.

[43] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction based cache placement for storage caches. In *in USENIX Annual Technical Conference. Usenix*, 2003.

[44] Yuanyuan Zhou, Z. Chen, and K. Li. Second-level buffer cache management. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):505–519, 2004.

[45] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association.

[46] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '05, pages 145–156, New York, NY, USA, 2005. ACM.

[47] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[48] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[49] Ian Foster and Carl Kesselman, editors. *The Grid 2, Second Edition: Blueprint for a New Computing Infrastructure (The Elsevier Series in Grid Computing)*. Elsevier, Nov 2003.

[50] HTCondor. Computing with htcondor: http://research.cs.wisc.edu/htcondor/, 2014.

[51] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[52] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 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, Berkeley, CA, USA, 2013. USENIX Association.

[53] The OpenBSD Project. Openssh: http://www.openssh.org, 2014.

[54] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A.G. Shet, G. Chrysos, and P. Dubey. Design and implementation of the linpack benchmark for single and multi-node systems based on intel Xeon Phi coprocessor. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 126–137, May 2013.

[55] David Luebke, Mark Harris, Naga Govindaraju, Aaron Lefohn, Mike Houston, John Owens, Mark Segal, Matthew Papakipos, and Ian Buck. GPGPU: General-purpose computation on graphics hardware. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[56] Keon Jang, Sangjin Han, Seungyeop Han, Sue Moon, and KyoungSoo Park. Sslshader: Cheap ssl acceleration with commodity processors. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[57] D. Demidov, K. Ahnert, K. Rupp, and P. Gottschling. Programming cuda and opencl: A case study using modern c++ libraries. *SIAM Journal on Scientific Computing*, April 2013. Accepted.

[58] The OpenSSL Project. Openssl: http://www.openssl.org, 2014.

[59] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 230–243, New York, NY, USA, 2001. ACM.

[60] Ganglia. Ganglia monitoring system: http://ganglia.info/, 2014.

[61] NLANR/DAST : Iperf - the TCP/UDP bandwidth measurement tool, 2003.

[62] HPN-SSH:http://www.psc.edu/index.php/hpn-ssh/640, 2012.

[63] Mellanox. RDMA aware networks programming user manual, Jan 2010.

[64] Ping Lai, Hari Subramoni, Sundeep Narravula, Amith Mamidala, and Dhabaleswar K. Panda. Designing efficient FTP mechanisms for high performance data-transfer over InfiniBand. In *Proceedings of International Conference on Parallel Processing (ICPP)*, September 2009.

[65] Performance tuning guidelines for Mellanox network adapters, March 2012.

[66] Globus Group. GT 4.0 GridFTP Glossary: http://www.globus.org/toolkit/docs/4.0/data/gridftp/gridftp_glossary.html, 2012.

[67] Globus Developer Group. GridFTP Threaded Flavors: http://www.globus.org/toolkit/docs/5.0/5.0.0/data/gridftp/admin/, 2012.

[68] Nigel Griffiths. nmon performance: A free tool to analyze AIX and Linux performance: http://www.ibm.com/developerworks/aix/library/au-analyze_aix/, Feb 2006.

[69] John D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[70] Mark Wagner. Achieving top network performance. March 2012.

[71] Andi Kleen. An NUMA API for linux, August 2004.

[72] Will Reese. Nginx: The high-performance web server and reverse proxy. *Linux J.*, 2008(173), September 2008.

[73] George Kingsley Zipf. Relative frequency as a determinant of phonetic change. *Harvard Studies in Classical Philology*, 40:pp. 1–95, 1929.

[74] Jeffrey Katcher. Postmark: A new file system benchmark. 1997.

[75] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[76] New mellanox interconnect to break 100g throughput, June 2012.

[77] perf : Linux profiling with performance counters, 2012.

[78] H. Subramoni, P. Lai, R. Kettimuthu, and D. K. Panda. High performance data transfer in grid environment using GridFTP over InfiniBand. In *Int'l Symposium on Cluster Computing and the Grid (CCGrid)*, May 2010.

[79] NUMA memory allocation policy for tmpfs, 2006.

[80] Brian Tierney, Ezra Kissel, Martin Swany, and Eric Pouyoul. Efficient data transfer protocols for big data. In *Proceedings of the 8th International Conference on eScience*, October 2012.

[81] Adam Sweeney. Scalability in the XFS file system. In *Proceedings of USENIX Annual Technical Conference*, pages 1–14, 1996.

[82] OpenFabrics. OpenFabrics Alliance: http://www.openfabrics.org/, 2012.

[83] ORACLE_BASE. Direct and asynchronous i/o: http://www.oracle-base.com/articles/misc/direct-and-asynchronous-io.php, 2014.

[84] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 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, Berkeley, CA, USA, 2013. USENIX Association.

[85] NVIDIA. TESLA K80 GPU ACCELERATOR . http://images.nvidia.com/content/pdf/kepler/tesla-k80-boardspec-07317-001-v05.pdf, 2015.

[86] Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can heterogeneity make gnutella scalable? In *In Proceedings of the first International Workshop on Peer-to-Peer Systems*, pages 94–103, 2002.

[87] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 15–15, Berkeley, CA, USA, 1999. USENIX Association.

[88] John Ousterhout. Why threads are a bad idea (for most purposes). *Presentation given at the 1996 Usenix Annual Technical Conference*, 5, 1996.

[89] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[90] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[91] Paul Grun. Introduction to infinibandfor end users. 2010.

[92] The Internet Engineering Task Force (IETF). RFC 4392 - IP over InfiniBand (IPoIB) Architecture, April 2006.

[93] Anthony Danalis, Aaron Brown, Lori Pollock, and Martin Swany. Introducing gravel: An MPI companion library. In *Proceedings of IEEE International Symposium of Parallel and Distributed Processing (IPDPS)*, Miami, Florida USA, April 2008.

[94] Anthony Danalis, Aaron Brown, Lori Pollock, Martin Swany, and John Cavazos. Gravel: A communication library to fast path MPI. In *Euro PVM/MPI 2008*, October 2008.

[95] DAT Collaborative. uDAPL: User Direct Access Programming Library. http://www.datcollaborative.org/udapl_doc_062102.pdf, June 2002.

[96] Nageswara S. V. Rao, Weikuan Yu, William R. Wing, Stephen W. Poole, and Jeffrey S. Vetter. Wide-area performance profiling of 10GigE and InfiniBand technologies. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, November 2008.

[97] Weikuan Yu, Nageswara S.V. Rao, Pete Wyckoff, and Jeffrey S. Vette. Performance of RDMA-capable storage protocols on wide-area network. In *Proceedings of Petascale Data Storage Workshop*, November 2008.

[98] M. Luo, S. Potluri, P. Lai, Emilio, P. Mancini, H. Subramoni, K. C. Kandalla, S. Sur, and D. K. Panda. High performance design and implementation of nemesis communication layer for two-sided and one-sided MPI semantics in MVAPICH. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, 2010.

[99] Hari Subramoni, Ping Lai, Miao Luo, and Dhabaleswar K. Panda. RDMA over Ethernet: A preliminary study. In *Proceedings of Cluster Computing Workshops, CLUSTER'09*, August 2009.

[100] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[101] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[102] Eelco Plugge, Tim Hawkins, and Peter Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. Apress, Berkely, CA, USA, 1st edition, 2010.

[103] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 295–306, New York, NY, USA, 2014. ACM.

[104] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[105] Abhijeet Joglekar, Michael E. Kounavis, and Frank L. Berry. A scalable and high performance software iscsi implementation. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 20–20, Berkeley, CA, USA, 2005. USENIX Association.

[106] Da Zheng, Randal Burns, and Alexander S. Szalay. A parallel page cache: Iops and caching for multicore systems. In *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems*, HotStorage'12, pages 5–5, Berkeley, CA, USA, 2012. USENIX Association.

[107] A. Singh, S. Gopisetty, K. Voruganti, D. Pease, and Ling Liu. A hybrid access model for storage area networks. In *Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE / 13th NASA Goddard Conference on*, pages 181 – 188, april 2005.

[108] Peter Radkov, Li Yin, Pawan Goyal, Prasenjit Sarkar, and Prashant Shenoy. A performance comparison of nfs and iscsi for ip-networked storage. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 101–114, Berkeley, CA, USA, 2004. USENIX Association.

[109] S. Aiken, D. Grunwald, A.R. Pleszkun, and J. Willeke. A performance analysis of the iscsi protocol. In *Mass Storage Systems and Technologies, 2003. (MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pages 123 – 134, april 2003.

[110] Yingping Lu and D.H.C. Du. Performance study of iscsi-based storage subsystems. *Communications Magazine, IEEE*, 41(8):76 – 82, aug. 2003.

[111] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[112] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010.