DISSERTATION

submitted to the

Combined Faculty of the Natural Sciences and Mathematics

of the

Ruprecht-Karls University
Heidelberg

for the degree of

Doctor of Natural Sciences

put forward by

Diplom-Informatikerin Sarah Marie Neuwirth

Born in
Mannheim, Germany

Heidelberg, 2018

Accelerating Network Communication and I/O in Scientific High Performance Computing Environments

Advisor: Prof. Dr.-Ing. Ulrich Brüning

Date of oral examination: ....

Abstract

High performance computing has become one of the major drivers behind technology inventions and science discoveries. Originally driven through the increase of operating frequencies and technology scaling, a recent slowdown in this evolution has led to the development of multi-core architectures, which are supported by accelerator devices such as graphics processing units (GPUs). With the upcoming exascale era, the overall power consumption and the gap between compute capabilities and I/O bandwidth have become major challenges. Nowadays, the system performance is dominated by the time spent in communication and I/O, which highly depends on the capabilities of the network interface. In order to cope with the extreme concurrency and heterogeneity of future systems, the software ecosystem of the interconnect needs to be carefully tuned to excel in reliability, programmability, and usability.

This work identifies and addresses three major gaps in today's interconnect software systems. The I/O gap describes the disparity in operating speeds between the computing capabilities and second storage tiers. The communication gap is introduced through the communication overhead needed to synchronize distributed large-scale applications and the mixed workload. The last gap is the so called concurrency gap, which is introduced through the extreme concurrency and the inflicted learning curve posed to scientific application developers to exploit the hardware capabilities.

The first contribution is the introduction of the network-attached accelerator approach, which moves accelerators into a "stand-alone" cluster connected through the Extoll interconnect. The novel communication architecture enables the direct accelerators communication without any host interactions and an optimal application-to-compute-resources mapping. The effectiveness of this approach is evaluated for two classes of accelerators: Intel Xeon Phi coprocessors and NVIDIA GPUs.

The next contribution comprises the design, implementation, and evaluation of the support of legacy codes and protocols over the Extoll interconnect technology. By providing TCP/IP protocol support over Extoll, it is shown that the performance benefits of the interconnect can be fully leveraged by a broader range of applications, including the seamless support of legacy codes.

The third contribution is twofold. First, a comprehensive analysis of the Lustre networking protocol semantics and interfaces is presented. Afterwards, these insights are utilized to map the LNET protocol semantics onto the Extoll networking technology. The result is a fully functional Lustre network driver for Extoll. An initial performance evaluation demonstrates promising bandwidth and message rate results.

The last contribution comprises the design, implementation, and evaluation of two easy-to-use load balancing frameworks, which transparently distribute the I/O workload across all available storage system components. The solutions maximize the parallelization and throughput of file I/O. The frameworks are evaluated on the Titan supercomputing systems for three I/O interfaces. For example for large-scale application runs, POSIX I/O and MPI-IO can be improved by up to 50% on a per job basis, while HDF5 shows performance improvements of up to 32%.

Zusammenfassung

Hochleistungsrechnen hat sich zu einem der bedeutendsten Standbeine im Bereich der technischen und wissenschaftlichen Errungenschaften entwickelt. Ursprünglich wurde die Leistungssteigerung solcher Systeme durch die kontinuierliche Steigerung der Taktfrequenz gewährleistet. Die Verlangsamung dieses Trends hat zu der Entwicklung von Mehrkernarchitekturen geführt, welche zusätzlich durch sogenannte Beschleuniger wie etwa Graphikprozessoren unterstützt werden. In Verbindung mit der bevorstehenden Exascale Ära haben sich vor allem der Gesamtstromverbrauch und die Kluft zwischen Rechenkapazität und I/O Bandbreite als limitierende Faktoren herauskristallisiert. Die gegenwärtige Systemleistung wird vor allem durch die Kommunikations- und I/O-Zeit beschränkt. Dieses Phänomen hängt insbesondere mit den Eigenschaften der Netzwerkschnittstelle zusammen. Um den extremen Anforderungen in Hinblick auf Parallelität und Heterogenität zukünftiger Systeme gerecht zu werden, bedarf es einer sorgfältigen Abstimmung der Software-Komponenten, insbesondere im Hinblick auf Zuverlässigkeit, Programmierbarkeit und Benutzerfreundlichkeit.

Diese Arbeit identifiziert und widmet sich insbesondere drei Leistungslücken (engl. gaps), die in den heutigen Softwareumgebungen im Bereich der Verbindungsnetzwerke beobachtet werden können. Die sogenannte *I/O Gap* beschreibt die Leistungslücke, die entsteht durch die unterschiedlichen Taktfrequenzen von Rechenkapazitäten und der Speicherhierarchie. Die sogenannte *Communication Gap* beschreibt den Kommunikationsoverhead, der bei der Synchronisierung von Großanwendungen entsteht, aber auch die gemischte Kommunikationslast, die auf das Netzwerk ausgeübt wird. Die letzte Leistungslücke wird durch die sogenannte *Concurrency Gap* beschrieben. Diese Leistungslücke entsteht durch die extreme Parallelität von modernen Hochleistungsrechnern und dem für Programmierer dadurch verbundenen Mehraufwand, die Hardware-Eigenschaften dieser Systeme auszunutzen.

Im ersten Beitrag wird der Ansatz der Network-Attached Accelerators („Netzwerk-Beschleuniger“) als neue Kommunikationsarchitektur vorgestellt. Dieses neuartige Konzept ermöglicht es, Beschleuniger aus der bislang statischen Hardware-Anordnung zu entkoppeln und diese über das Netzwerk allen Rechenknoten gleichförmig zur

Verfügung zu stellen. Vorteile dieser Architekturform sind unter anderem, dass Grafikprozessoren direkt über das Netzwerk miteinander kommunizieren können, aber auch das dynamische Abbilden von Anwendungen auf Rechenressourcen zur Laufzeit. Die Leistungsfähigkeit dieses Ansatzes wird anhand der Intel Xeon Phi Co-Prozessoren und NVIDIA Grafikprozessoren analysiert.

Im nächsten Beitrag wird der Fokus auf die Unterstützung von sogenannten Legacy Anwendungen und Protokollen über Hochgeschwindigkeitsnetzwerke wie Extoll gelegt. Durch die Erweiterung der Extoll Softwareumgebung zur Unterstützung der TCP/IP Protokollfamilie können Legacy Anwendungen das Leistungsspektrum der Extoll Technologie voll ausschöpfen, ohne dafür modifiziert werden zu müssen. Dies öffnet die Tür für ein breiteres Spektrum an Anwedungen.

Der dritte Beitrag liefert zunächst eine ausführliche Analyse des Lustre Netzwerkprotokolls, welches LNET genannt wird. Im Anschluss werden diese Erkenntnisse genutzt, um die Protokoll-Semantik von LNET effizient auf die Extoll Netzwerktechnologie abzubilden. Es wird ein voll funktionsfähiger Lustre Netzwerktreiber (Lustre Network Driver) für Extoll implementiert. Eine initiale Leistungsanalyse zeigt vielsprechende Ergebnisse, gerade im Hinblick auf die erzielte Bandbreite und Nachrichtenrate.

Der letzte Beitrag besteht in der Konzeption, Implementierung und Evaluation zweier benutzerfreundlicher Anwender-Frameworks, welche die Datenlast einer Großanwendung transparent über alle verfügbaren Komponenten des Speichersystems verteilen. Diese beiden Lösungen dienen der optimalen Parallelisierung und Maximierung der Bandbreite in Bezug auf das Lesen und Schreiben von Dateien. Die beiden Frameworks werden auf dem Hochleistungsrechner Titan mit drei verschiedenen I/O Schnittstellen (I/O interfaces) evaluiert. Für Großanwendungen kann die Leistung von POSIX und MPI-IO beispielsweise um bis zu 50% gesteigert werden, für HDF5 kann eine Leistungssteigerung von bis zu 32% erzielt werden.

Acknowledgements

First and foremost, I would like to thank my parents, Manfred (†14.01.2011) and Michaela Neuwirth, for their unconditional love and endless support. They have always encouraged me, and are still encouraging me, to pursue my dreams and have lit the fire for my passion in sciences at a very young age. A special thank you goes to my mother for always motivating me. Without her support, I would not have been able to accomplish this work. I will always be grateful for being blessed with such wonderful parents and dedicate this work to them.

I would like to express my sincere gratitude to Prof. Dr. Ulrich Brüning for his invaluable advice and support during the course of this work. With his knowledge and experience, he has been a fantastic counterpart in countless discussions and has provided valuable input to my research. I am also grateful to all the members, staff and students, of the *Computer Architecture Group* and the employees of the *EXTOLL GmbH* for all the valuable discussions and advice throughout this work. In addition, I would like to express my gratitude to the *Ruprecht-Karls University* and the *Faculty for Mathematics and Computer Science* for providing me with the opportunity to pursue a doctoral degree.

Special thanks go to James H. Rogers, Dr. Sarp Oral, Dr. Feiyi Wang, and the *Technology Integration Group* of the Oak Ridge National Laboratory for providing me with the opportunity to spend two summers as a visiting research scholar in their team. The insights in the operation and evaluation of large-scale systems such as the Titan supercomputer have provided me with a unique chance to expand my horizon in the area of scientific high performance computing systems.

Given that it is impossible to thank every single person who has accompanied me during the journey of my PhD, I would like to express my sincere gratitude to all the inspiring people I have met throughout the years. Thank you for all the cheering, support, discussions, and friendships. I am grateful to each one of you.

Contents

1 Introduction **1**
1.1 Motivation and Challenges . . . 4
1.2 Contributions . . . 5
1.3 Outline . . . 7

2 Communication and I/O in HPC Systems **9**
2.1 Generic Communication Architecture Overview . . . 10
2.2 Network Communication Hardware . . . 12
2.2.1 Interconnection Networks . . . 12
2.2.2 Network Interface Controllers . . . 14
2.3 Communication in Distributed Memory Systems . . . 15
2.3.1 Communication Schemes . . . 15
2.3.2 Synchronization . . . 18
2.3.3 Performance Metrics . . . 18
2.3.4 Interprocess Communication Interfaces . . . 19
2.4 Introduction to Parallel I/O . . . 25
2.4.1 Scientific I/O . . . 26
2.4.2 Parallel File Systems . . . 27
2.4.3 High-level I/O Libraries and Middleware . . . 28
2.4.4 Access Patterns . . . 31

3 Extoll System Environment **35**
3.1 Technology Overview . . . 35
3.2 Functional Units . . . 36
3.2.1 Remote Memory Access Unit . . . 37
3.2.2 Virtualized Engine for Low Overhead Unit . . . 39
3.2.3 Virtual Process ID . . . 40
3.2.4 Shared Memory Functional Unit . . . 41
3.2.5 Address Translation Unit . . . 43

3.2.6 Register File . . . 45
3.2.7 PCIe Bridge . . . 45
3.3 Software Environment . . . 47
3.3.1 Kernel Space . . . 47
3.3.2 User Space . . . 48
3.3.3 EMP: Network Discovery and Setup . . . 48
3.4 Related Interconnection Standards . . . 49
3.4.1 Infiniband . . . 49
3.4.2 PCI Express . . . 50
3.5 Performance Overview . . . 52
3.5.1 Test Setup . . . 52
3.5.2 Performance Results . . . 53

4 Network-Attached Accelerators 55
4.1 Motivation . . . 56
4.2 DEEP Project Series . . . 57
4.3 Introduction to the PCI Express Subsystem . . . 58
4.3.1 PCI Express Address Spaces . . . 59
4.3.2 Linux PCI Express Enumeration . . . 62
4.3.3 PCI Express Expansion Cards . . . 64
4.4 Related Work . . . 66
4.4.1 Intel Xeon Phi Coprocessor-based Communication Models . . . 66
4.4.2 GPU Virtualization and Communication Techniques . . . 67
4.4.3 Hardware-related Research . . . 69
4.5 NAA Software Design . . . 69
4.5.1 System Architecture and Problem Statement . . . 70
4.5.2 Objectives and Strategy . . . 72
4.5.3 Design Space Analysis . . . 77
4.6 DEEP Booster Architecture . . . 79
4.6.1 Hardware Components . . . 79
4.6.2 Prototype Implementation . . . 80
4.6.3 Prototype Performance Evaluation . . . 84
4.6.4 GreenICE – An Immersive Cooled DEEP Booster . . . 90
4.6.5 Lessons Learned . . . 90
4.7 Virtualization of Remote PCI Express Devices . . . 91
4.7.1 Concept Overview of VPCI . . . 91
4.7.2 PCI Express Device Emulation . . . 92

4.7.3 Forwarding PCI Configuration Space Requests 93
4.7.4 Device Enumeration 94
4.7.5 Forwarding Memory-Mapped I/O Requests 94
4.7.6 Interrupt Delivery 95
4.7.7 Overall Picture 96
4.7.8 Experimental Evaluation 97
4.8 NAA Summary 99

5 RDMA-Accelerated TCP/IP Communication 101
5.1 Introduction to the Internet Protocol Suite 102
5.1.1 The Network Layer: IP 103
5.1.2 The Transport Layer 104
5.1.3 Data Transmission and Reception in Linux 106
5.1.4 Interrupt Coalescing and NIC Polling with NAPI 107
5.1.5 TCP/IP Protocol Overhead and Bottlenecks 109
5.2 Related Work 111
5.2.1 OpenFabrics Enterprise Distribution 111
5.2.2 Sockets-like Interfaces 114
5.3 Objectives and Strategy 115
5.4 Transmission of Ethernet Frames over Extoll 117
5.4.1 Link Frame Transmission and Reception 117
5.4.2 Message Matching and Resource Management 120
5.4.3 Maximum Transmission Unit 122
5.4.4 Address Mapping – Unicast 123
5.4.5 Multicast Routing 125
5.4.6 EXN: Extoll Network Interface 128
5.5 Direct Sockets over Extoll 131
5.5.1 Protocol Overview 131
5.5.2 Setup and Connection Management 132
5.5.3 Data Transfer Mechanisms 135
5.5.4 AF_EXTL: A Prototype Implementation of EXT-DS 139
5.6 Performance Analysis 145
5.6.1 TCP/IP Configuration Tuning in Linux Systems 145
5.6.2 Test System 147
5.6.3 Microbenchmark Evaluation 147
5.6.4 MPI Performance 151
5.7 TCP/IP Summary 153

6 Efficient Lustre Networking Protocol Support 155
6.1 Introduction to the Lustre File System . . . 156
6.1.1 File System Components . . . 156
6.1.2 Network Communication Protocol . . . 158
6.1.3 Client Services and File I/O . . . 161
6.2 Lustre Networking Semantics and Interfaces . . . 164
6.2.1 Naming Conventions and API Summary . . . 165
6.2.2 Memory-Oriented Communication Semantics . . . 166
6.2.3 Credit System . . . 170
6.2.4 Available Lustre Network Drivers . . . 171
6.3 Design Challenges and Strategy . . . 172
6.4 Efficient RDMA with Vectored I/O Operations . . . 173
6.4.1 Memory Management . . . 173
6.4.2 Infiniband Verbs and Scatter/Gather Elements . . . 174
6.4.3 Scatter/Gather DMA Operation Support for Extoll . . . 176
6.5 Support for LNET Protocol Semantics . . . 177
6.5.1 Data Transmission Protocols . . . 177
6.5.2 Message Matching and Descriptor Queues . . . 180
6.6 EXLND: Extoll Lustre Network Driver . . . 182
6.7 Preliminary Performance Results . . . 183
6.7.1 System Setup and Methodology . . . 183
6.7.2 LNET Self-Test Results . . . 184
6.8 EXLND Summary . . . 186

7 Resource Contention Mitigation at Scale 189
7.1 Spider II – A Leadership-Class File System . . . 190
7.2 The Need for Balanced Resource Usage . . . 191
7.3 Related Work . . . 193
7.4 Observations and Best Practices for File I/O . . . 195
7.5 End-to-End Performance Tuning . . . 196
7.5.1 Fine-Grained Routing Methodology . . . 196
7.5.2 Balanced Placement I/O Strategy . . . 199
7.6 Design Objectives and Strategy . . . 201
7.7 Aequilibro – An I/O Middleware Integration . . . 202
7.7.1 Transport Methods . . . 203
7.7.2 Software Design and Implementation . . . 204

7.8 TAPP-IO Framework . . . 206
7.8.1 Parallel I/O Support . . . 207
7.8.2 Runtime Environment . . . 210
7.9 Data Collection and Analysis . . . 212
7.9.1 Benchmarking Methodology . . . 212
7.9.2 Experimental Setup . . . 215
7.9.3 Synthetic Benchmark Evaluation . . . 216
7.9.4 HPC Workload Evaluation . . . 222
7.10 Summary . . . 223

8 Conclusion 225
8.1 Outlook . . . 227

List of Abbreviations 229

List of Figures 235

List of Tables 239

Listings 241

References 243

Chapter 1

Introduction

In recent years, the major driver behind technology inventions and science discoveries has been the computational science domain. Nowadays, *High Performance Computing* (HPC), in particular large-scale simulation codes [1], complements the two traditional pillars of science, *theory* and *experiment*, and enables researchers to build and test models of complex phenomena, including molecular dynamics, quantum mechanics, weather forecasting, climate research, and astrophysics.

For years, the performance of supercomputing systems has been scaled through increased clock frequencies, larger, faster memories, and high-bandwidth, low-latency interconnects. With the end of frequency and Dennard [2, 3] scaling in the early 2000's, parallel computing has become the new paradigm to increase the computational power of HPC deployments. In order to maintain the traditional growth rates, multi-core architectures have been developed and the overall system performance is scaled by constantly increasing components and node counts.

The success of this strategy until now is outlined in Figure 1.1 for the number one systems of the TOP500 [4] over the past two decades. A superficial reading of this graph provides no conclusive indication that future HPC systems might encounter any difficulties in increasing their performance at the same pace. This impression is mainly inflicted by the nature of the *High-Performance Linpack* (HPL) benchmark [5], which is utilized for the ranking. To obtain the HPL benchmarking results, HPC systems are typically operated in the *capability mode*, which aims at solving a single huge problem as quickly as possible by applying the system's entire compute power to that particular problem. The HPL solves a large dense system of linear equations and heavily relies on re-using cached data from local registers and caches.

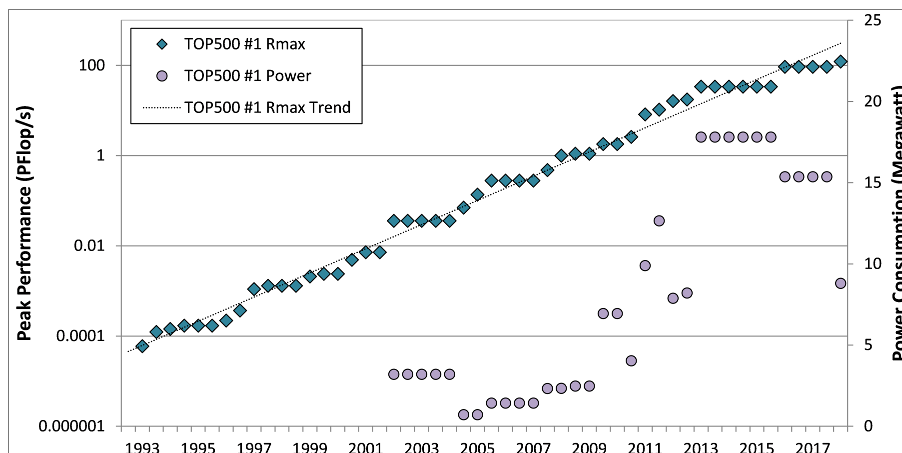


Figure 1.1: TOP1 system performance and power consumption development.

This means that the HPL is embarrassingly parallel and puts very little load on the network. In reality, however, large-scale simulations are increasingly data-intensive and put excessive pressure on the interconnection network to perform communication and I/O. Most HPC environments in the open science domain are operated in the *capacity mode*. Such systems aim to solve a plurality of problems simultaneously by partitioning their compute powers and applying one partition to each job exclusively.

Today's supercomputing facilities comprise two major building blocks: the compute cluster and the storage system. A cluster typically consists of up to thousands of compute nodes, with each node being equipped with four main components. The *central processing unit* (CPU) implements the aforementioned multi-core architecture, and runs the operating system and applications. The most predominant CPUs in the current TOP500 (June 2018) are Intel's Xeon processors with a share of 94%. The second system component is the *memory*. For example, the current number one system Summit provides 512 GB of DDR4 per node, which is complemented by 96 GB of high bandwidth memory (HBM2). Another major component is the *interconnect*, which provides the connectivity to the compute cluster and the storage system. Currently, the predominant interconnect technologies are 10 Gigabit Ethernet (34%) and Infiniband (34%). However, half of the TOP500's system performance is delivered through proprietary and custom interconnects such as Cray's Aries and Fujitsu's Tofu, which makes this work particularly relevant. The last component is represented by *accelerators*, whose number has been constantly increasing over the past years. Here, NVIDIA's *graphics processing units* (GPUs) dominate the system share. In recent years, the increase in CPU cores has led to a deterioration of power dissipation. This has enforced the popularity of accelerators, which comprise massively parallel architectures. Accelerators deliver remarkably high performance while providing

an unprecedented energy efficiency. However, with their introduction, the extreme concurrency and heterogeneity in HPC systems has been further intensified. This poses major challenges to application developers as the code needs to be carefully parallelized to exploit the benefits of such heterogeneous systems.

The second major building block of supercomputing facilities are storage and I/O systems, which typically consist of hundreds of storage servers and up to thousands of disks. Designed for capacity and capability, storage systems are inherently complex and shared among concurrently running jobs. With the emergence of data-intensive applications, storage systems are further contended for performance and scalability. Given that storage capabilities are typically much lower than those of a processor, the processing of large amounts of data has introduced a phenomenon referred to as the *I/O gap*, which is inflicted by the latency gap between processors and storage tiers. A central component for providing good storage connectivity is the interconnection network, which requires careful optimization through intermediate software layers to provide optimal bandwidth performance and mitigate network contention.

In traditional HPC systems, interprocess communication mostly relies on message passing in order to synchronize and move data, and involves the interconnection network for internode communication. As pointed out by a recent study [6], large-scale applications spend an average of about 34% of their runtime with point-to-point or collective operations, which indicates that communication severely impacts the performance of an application run. While computation is relatively fast, communication has been proven to be one of the major bottlenecks [7] for parallel computing systems, especially with regard to power consumption.

The next major challenge will be to provide exascale performance [8], i.e., to perform a quintillion (10^{18}) floating point operations per second, which will be impacted by the power consumption, resiliency, memory hierarchies and I/O, and concurrency. Modern HPC facilities comprise heterogeneous system architectures, which heavily rely on the interconnection network. With the paradigm shift to parallel computing and the upcoming exascale challenge, the system performance is dominated by the time spent in communication and I/O. In order to cope with the extreme concurrency and heterogeneity of future systems, the software ecosystem needs to be carefully tuned to excel in reliability, programmability, power consumption, and usability. With the interconnection network being the backbone for both message exchange and I/O, this work focuses on the gaps in today's intermediate software environments with regard to direct accelerator and internode communication, storage connectivity, and optimal resource utilization for both compute and storage components.

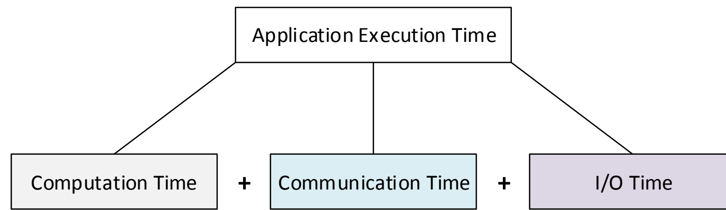


Figure 1.2: Key factors influencing the total application execution time.

1.1 Motivation and Challenges

The motivation for this work is based on the observation that the total application execution time is affected by the computation, communication, and I/O time, as displayed in Figure 1.2. In order to maximize the overall system performance, all components of the equation need to be carefully addressed. As previously described, a major contributing factor is the efficient network communication and I/O, which is heavily impacted by the following three observations:

I/O Gap Large-scale applications often do I/O for reading initial datasets or writing numerical data from simulations out to the distributed disk arrays, but also periodically store application-level checkpoints. Compared to other components of an HPC platform, I/O subsystems are typically slow. Different hardware components operate at different speeds, thus, resulting in the so called I/O gap. The anatomy of a disk access comprises of the latency and bandwidth inflicted by the CPU, the memory, the interconnect, and the actual hard drive access. For example, the memory access latency is 100 times slower than the CPU. Since a pioneering technological breakthrough is not expected any time soon, optimizing the I/O time is just as important as optimizing the computation and communication time. Thus, it is necessary to carefully optimize and implement a number of intermediate layers for the coordination of the data accesses. This includes the efficient mapping of a parallel file system's network protocol to the chosen interconnect technology, but also the allocation of storage resources in a balanced manner to maximize the aggregate file access bandwidth.

Communication Gap In the past, the conventional wisdom has been that computation is rather expensive while communication is cheap. With the paradigm shift to parallel computing, the system size is scaled by adding more and more compute nodes, which results in the problem that large-scale applications need to be distributed among these resources. This reduces the amount of time spent in computation in relation to the time spent in communication, and thus,

produces the communication gap. The workload of bulk data transfers mixed with the vast amount of small messages for communication and synchronization purposes requires the utilization of highly efficient network technologies. It is desirable to increase the performance of point-to-point and collective communication through extensive tuning of the intermediate software layers, particularly, the software environment of the chosen interconnect by leveraging its hardware capabilities efficiently.

Concurrency Gap Another gap is introduced through the extreme concurrency of today's heterogeneous system architectures. In order to increase the overall system performance while providing a reasonable energy efficiency, more and more accelerator devices are added to modern HPC systems. They offer an unprecedented computational power per watt, while being optimized for massively parallel computation. However, the efficient utilization of such devices is significantly limited by the static application-to-compute-resources mapping, but also through the steep learning curve posed to scientific application developers to exploit the hardware capabilities. Another challenge is introduced through the overhead associated with direct communication between distributed accelerators. Especially with the upcoming exascale era in mind, new communication architectures need to be explored to serve the needs of extreme-scale applications. By leveraging the innovative hardware features of modern interconnects such as Extoll, intermediate software layers can be designed that enable the dynamic workload distribution at run time in an N-to-M ratio, e.g., by providing device virtualization over the network while bypassing the host CPU. Such virtualization techniques also introduce novel concepts for direct accelerator communication.

1.2 Contributions

This work aims to resolve the described observations by closing the gaps through the design, implementation, and evaluation of intermediate software layers. The contributions can be divided in two parts.

The first part evolves around the Extoll interconnect technology, which has been designed to fit the needs of future large-scale simulation codes. The technology has mainly been chosen for two reasons. First, as previously presented, customized interconnects deliver 50% of the performance share in the TOP500, which makes them an important component for the design and delivery of exascale systems. Second,

Extoll has emerged from a research project conducted at the Heidelberg University. The second part of the contributions evolves around the Titan supercomputing system. The research complements this work by providing insights on the effects of the I/O gap at larger scales. This research has been enabled through two research stays at the Oak Ridge National Laboratory and a subsequent research collaboration.

The presented work makes the following contributions:

(1) *Introduction of the Network-Attached Accelerator approach as a new concept for direct accelerator communication and efficient resource utilization* – The network-attached accelerator approach introduces a novel communication architecture, which allows the disaggregation of PCI Express hosts from the end-points that they control. The key idea is to combine a cluster of multi-core processors with a tightly coupled cluster of many-core processors (e.g., GPUs or coprocessors) employing a highly scalable network. This unique architecture aims at the optimal application-to-compute-resource mapping, energy efficiency and extreme scalability, and therefore, targets the concurrency and communication gaps. This contribution comprises conference publications [9, 10], a poster paper [11], and a news article [12].

(2) *RDMA-Accelerated TCP/IP communication* – Modern high performance network technologies provide the hardware support for efficient communication models such as remote direct memory access (RDMA) and partitioned global address space (PGAS). However, they also need to support traditional communication semantics, e.g., the Sockets interface, to seamlessly support legacy applications. The challenge is to design a software layer that is capable of exploiting the innovative hardware features to a legacy code without the need for any source code modifications or recompilation. This work introduces two different communication protocols targeting the acceleration of traditional TCP/IP communication. They are implemented as transparent, intermediate software layers, and provide IP addressing and address resolution support while leveraging the RDMA capabilities of Extoll. This contribution has led to two workshop presentations [13, 14].

(3) *Efficient storage connectivity and I/O* – File I/O is an important part of large-scale applications. In order to maximize the performance, HPC systems are typically backed by a parallel file and storage system. One popular choice is the Lustre file system due to its support for data-intensive applications and POSIX-complaint namespace for large-scale deployments. Using a high-performance interconnect such as Extoll to speed up to I/O-bound part of simulation codes

can improve the overall application execution time, but requires an efficient software layer between Lustre and the network interface. In case of Lustre, these software layers comprise of the Lustre Networking Protocol (LNET) and Lustre Network Drivers (LNDs). This work presents the design, implementation, and evaluation of the Extoll LND, which efficiently maps the LNET protocol semantics onto the Extoll technology. The contribution has been presented in a scientific talk [15] and published at a workshop [16].

(4) *Transparent resource contention mitigation in large-scale HPC systems* – Besides the need to accelerate and optimize the network communication of a parallel file system through efficient protocol support, the careful management of data accesses is another major pillar of improving file I/O. In this work, two easy-to-use load balancing frameworks are designed, implemented, and evaluated, which transparently distributes data across all available storage system components. The solutions balance the I/O workload on an end-to-end and per job basis, and maximize the parallelization and throughput of file I/O. The frameworks are evaluated on the Titan supercomputing systems. This contribution has been published in two conference papers [17, 18] and one poster paper [19].

1.3 Outline

Chapter 2 provides the basic knowledge about communication and I/O in HPC systems, which is needed for the entire remainder of this work. In Chapter 3, an overview of the Extoll system environment is presented. Together with chapter 2, this Chapter provides the foundation for the presented work.

Chapter 4 presents the design and implementation of the network-attached accelerator (NAA) approach. NAA decouples accelerator devices from the host systems and enables optimal application-to-compute-resource mapping. NAA is implemented and evaluated with Intel Xeon Phi coprocessors and NVIDIA K20c GPUs.

Chapter 5 focuses on the support of the TCP/IP communication over Extoll. By efficiently mapping the TCP/IP protocol semantics onto the network technology, two protocols are specified: Ethernet over Extoll (EXT-Eth) and Direct Sockets over Extoll (EXT-DS). EXT-DS emulates the Ethernet protocol over Extoll, provides IP addressing means and accelerates traditional TCP/IP communication by providing asynchronous, two-sided RDMA for larger payload sizes. EXT-DS complements EXT-Eth by providing kernel bypass data transfers through RDMA operations for dedicated TCP point-to-point connections.

In Chapter 6, the Lustre network protocol is analyzed followed by the design of EXLND, which provides peer-to-peer connection semantics for Lustre over Extoll. The evaluation of a prototype implementation of EXLND is presented.

While the previous chapters focus on the design of communication protocols for the Extoll interconnect technology, Chapter 7 explores resource contention mitigation techniques for large-scale HPC deployments. The evaluation platform is the Titan supercomputing system, in particular, its parallel file system Spider II, which is based on Lustre. The chapter presents a transparent auto-tuning framework for file striping, which transparently distributes stripes of data among available storage system components in a fair manner.

The thesis concludes in chapter 8 by providing a summary of the contributions, but also explores potential improvements to the various contributions presented throughout this work.

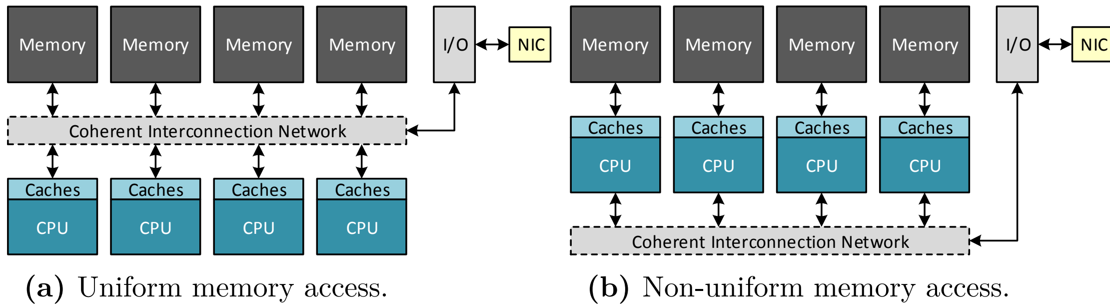
CHAPTER 2

Communication and I/O in HPC Systems

In general, parallel computing systems can be classified in two fundamental categories: *shared* and *distributed memory* systems. The distinctive feature for this categorization is how the memory can be accessed by processors, which also influences the communication model. Figure 2.1 provides an overview of shared memory system architectures. They provide a single address space, which can be accessed by all processors, and perform communication through shared variables. Depending on the arrangement of the memory, *uniform memory access* (UMA) and *non-uniform memory access* (NUMA) systems can be distinguished. In UMA systems, all processors have a uniform memory access latency. Due to the symmetrical design, it is also referred to as the *dance-hall architecture*. In NUMA systems on the other hand, memory modules are assigned to dedicated CPUs, which results in a higher access latency when accessing another processor's local memory. This characteristic also known as the *NUMA phenomenon*.

The second class of parallel computing architectures are distributed memory systems, as depicted in Figure 2.2. As indicated by the name, the memory modules are distributed across the system and physically assigned to dedicated CPUs. The communication is performed by exchanging messages. This requires address translation, which is performed by the network interface controller (NIC) at both endpoints.

Modern supercomputing architectures typically comprise of a combination of both distributed and shared memory architectures. Such systems consist of multiple *symmetric multi-processor* (SMP) nodes, which are connected to an interconnection network. While memory is shared by all processors of the same node, it is not shared between processors on different nodes. In the HPC domain, these systems are



(a) Uniform memory access. **(b)** Non-uniform memory access.

Figure 2.1: Shared memory system architectures.

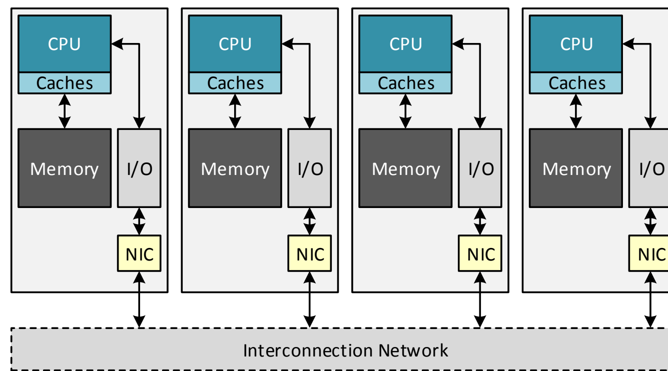


Figure 2.2: Distributed memory architecture.

called *massively parallel multi-processor* (MPP) systems. Applications running on such supercomputing environments typically rely on the message passing paradigm for communication and I/O, which means that processes exchange messages to synchronize and move data. In general, it can be said that large-scale HPC systems are multi-layered facilities comprising of several different entities, which makes communication and I/O a particularly complex task.

This chapter facilitates the basic knowledge for the remainder of this work and is organized as follows. The first section provides an introduction to a node's generic communication architecture. Afterwards, an overview of communication hardware is presented followed by a discussion about interprocess communication in distributed memory systems, which is of particular interest for this work. The chapter concludes with an overview of parallel I/O and storage environments.

2.1 Generic Communication Architecture Overview

A *communication architecture* defines the basic communication and synchronization operations within a system, and addresses the organizational structures that realize

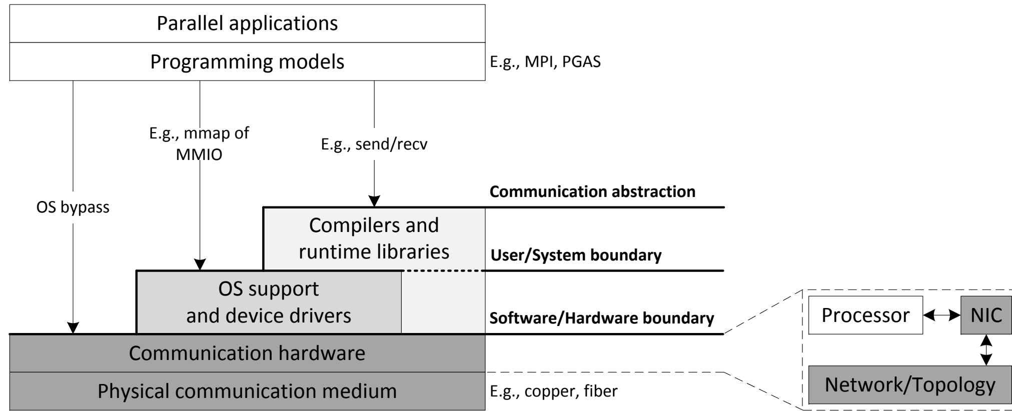


Figure 2.3: Communication architecture abstraction following Culler et al. [20].

these operations. As introduced by Culler et al. [20], a generic communication architecture can be divided in several different layers, which are implemented in both hardware and software as depicted in Figure 2.3.

Parallel applications and programming models compose the top layer. Typically, programming models are embedded in a parallel language or programming environment, and provide a *communication abstraction* to applications, e.g., for shared and distributed memory systems. A programming model specifies how the different parts of a parallel application exchange information between each other and what synchronization operations are available to coordinate their communication and activities. This is realized in terms of user-level communication primitives of the underlying system.

In between the programming models and the operating system, compilers and runtime libraries act as the *user/system boundary* by utilizing the primitives made available from the underlying operating system and hardware. Libraries depend on the used programming model and provide a hardware abstraction to the applications. The rule of thumb is that the more generalization is provided by a library, the less control for application and system-specific optimizations can be applied by the user. While less abstraction increases the variety of possible optimizations, it also burdens the user with system-specific characteristics when tuning an application's performance. Therefore, the level of abstraction is a trade-off between performance and user-friendliness.

The communication hardware and operating system (OS) comprise the bottom layer. Device drivers and the OS directly interface with the communication hardware, which typically consists of a network interface controller (NIC) and an interconnection network. Modern NICs provide hardware support for different communication

standards. Depending on the corresponding software stack, NICs can reduce the software overhead, e.g., by providing OS bypass techniques.

2.2 Network Communication Hardware

The following two sections provide an overview of two basic hardware building blocks for network communication: the *interconnection network* and the *network interface controller*.

2.2.1 Interconnection Networks

Following the definition by Dally and Towles [21], an interconnection network can be defined as a programmable system that transports data between terminals. In the context of interconnection networks, programmable means that different connections are established at different points in time. An interconnection network typically consists of several components, including buffers, channels, switches, and controls, that work together to deliver the data. This broad definition is applicable to many different layers within a computer systems, starting with networks-on-chip, which deliver data between memory arrays, registers, and arithmetic units, up to local area networks (LAN), which connect distributed memory systems within a data center or an enterprise.

Today, most high-performance interconnections are performed by point-to-point interconnection networks. This trend reflects the non-uniform performance scaling, which is imposed by the demand for better interconnection performance with the increasing processor performance and network bandwidth. Transfers should be completed with a latency as small as possible while a large number of such transfers should be allowed in parallel. Because the demand for interconnection has grown more rapidly than the capability of the underlying wires, the interconnection network has become a crucial bottleneck in most systems. The design of an interconnection network can be classified in the following categories [22, 23]:

Network Topology The topology describes the spatial arrangement of an interconnection network distinguished in regular and irregular structures. However, irregular interconnection networks are highly uncommon in high performance computing systems. Regular interconnection networks arrange processors as well as processes in regular patterns, which are aligned with common research problems. Depending on the use case, topologies can be further subdivided in two categories: static and

dynamic. Static topologies are fixed point-to-point connections between nodes, while dynamic topologies possess switching elements, which can be tuned for multiple different configurations. Static network topologies are mainly used for communication in large-scale computing systems. Also, static interconnection networks can be depicted as directed or undirected graphs in which nodes represent processors, compute nodes, switches or devices, and edges represent communication links. Example topologies are tree, mesh, cube, and hypercube. The chosen topology affects routing, reliability, throughput, latency, and scalability.

Routing Routing algorithms define how a packet is sent from the destination to the source node. They can be classified in two categories: *deterministic* and *adaptive routing* [24]. As implied by the name, deterministic routing algorithms always choose the same, predefined path between two nodes, independently from the current network state, for example the network traffic. On the contrary, adaptive routing algorithms choose a path based on the current state of the network, including historical channel information and the status of a node.

Switching Methodology and Flow Control Two major switching methodologies can be distinguished, *circuit switching* and *packet switching*. For circuit switching, a physical path is established between the source and the destination for the entire duration of the transmission. In case of packet switching, data is put in packets and routed through the interconnection network establishing a logical connection path between nodes without the need for a physical connection. In general, circuit switching is much more suitable for bulk data transmission, and packet switching is more efficient for short data messages. The switching methodology is tightly coupled with the chosen flow control. The flow control manages the allocation of resources along the path of a packet, including buffers and channels. Buffers are basically registers or memory, which provide a temporary storage to cache packets. Examples are circuit switching, virtual-cut-through switching (packet based), store-and-forward switching (packet based), and wormhole switching (flit based).

Operation Mode Two types of communication can be identified: *synchronous* and *asynchronous*. The asynchronous operation mode is mainly needed in multiprocessing environments where connection requests are issued dynamically. Synchronous communication is only allowed at certain points in time with communication paths established synchronously. One possible use case is the broadcast.

2.2.2 Network Interface Controllers

In computing environments, the network interface serves as the software and/or hardware interface between protocol layers or two pieces of equipment in a computer network [21]. Examples include network sockets, which provide a software interface to the network, and network interface controllers (NICs). The NIC connect clients to an interconnection network and serves as the network interface from a processor's point of view. Its key characteristics include low-overhead access and high-throughput message transfers. The guarantee that multiple processes can independently access the NIC while ensuring process security typically is provided through hardware virtualization mechanisms. The access for different processes is multiplexed by the operating system.

There are two different techniques for a NIC to indicate whether packets are available for transfer: *software-driven* and *interrupt-driven* I/O. In software-driven I/O, also known as *polling*, the processor actively samples the status of the peripheral under software control. When using interrupt-driven I/O, the peripheral alerts the processor when it is ready to send or receive data.

These two techniques are complemented by two data transmission methods [23]: *programmed I/O* (PIO) and *direct memory access* (DMA). In the context of a NIC, PIO requires the CPU to actively move data to or from the main memory to the NIC, while DMA describes an operation in which data is moved from one resource to another without the involvement of the processor. DMA removes load from the CPU, but requires more logic on the NIC.

One of the major design criteria is the placement of the NIC. In order the minimize the latency, it is desirable to place it as close to the processor as possible. In general, three different processor-network interfaces can be distinguished [21]: the *two-register*, the *register-mapped*, and the *descriptor-based* interfaces.

The simplest approach is the *two-register interface*. As indicated by the name, this concept utilizes two registers: the network input and the network output registers. Reading from or writing to these registers de- or enqueues the next word of a message. Longer messages are sent by splitting them in word-sized chunks and then writing them word by word to the output register. This adds a tremendous overhead to the communication since the processor is tied to serve as a DMA engine. Another problem is that misbehaving processors might occupy the network resources infinitely when they write the first part of a message, but fail to send the end.

The safety issue of the two-register interface can be resolved by utilizing the *register-mapped approach* where a message is sent atomically from a subset of the

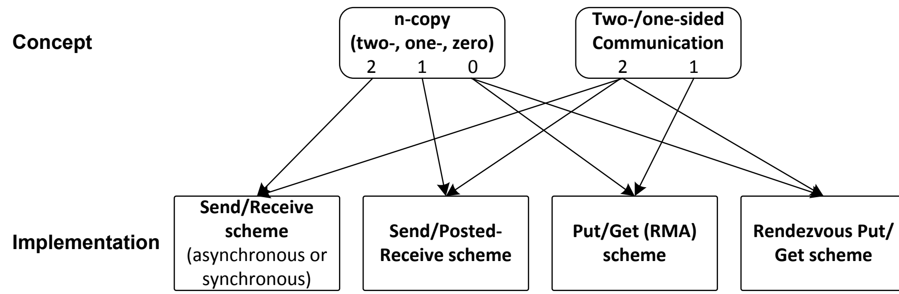


Figure 2.4: Communication models in distributed memory systems [23].

processor's general purpose registers. However, the limitation of the processor acting as the DMA engine remains and the data transfers are performed in a PIO-fashion.

The third concept introduces the use of *descriptors*. Instead of writing messages to a set of registers, the processor composes a descriptor, which can contain an immediate value of the message or the start address of a memory block and the corresponding size of the payload to be transferred. The descriptor is either written to a set of dedicated descriptor registers or a descriptor queue in main memory, and the NIC processes the messages from there. This approach minimizes the processor overhead by offloading the data transfer to the NIC.

2.3 Communication in Distributed Memory Systems

Today's HPC systems comprise of thousands of nodes, which are connected through an interconnection network. To serve the need for efficient data movement and communication in large-scale applications, different communication schemes and programming models are deployed.

In the following, an introduction to interprocess communication in distributed memory systems is presented. In addition, different communication schemes and interfaces are introduced, including the corresponding programming models, but also performance metrics and synchronization.

2.3.1 Communication Schemes

In distributed memory systems, several different interprocess communication schemes can be classified by two main criteria, as depicted in Figure 2.4. The first criterion is the number of copies done by the involved processes, which is indicated by *n-copy*. The second characteristic is the number of actively involved processes in the communication. Combinations of these criteria can be implemented in various schemes,

including the traditional *send/receive* scheme and *PUT/GET*. The understanding of these communication schemes is an important factor for the actual communication protocol design for a given network technology.

2.3.1.1 Two-sided Communication

The traditional way to implement interprocess communication in parallel applications targeting distributed memory systems is to exchange messages. This concept is also known as message passing or send/receive scheme. In message passing systems, messages are exchanged by using matching pairs of send and receive function calls on the sender and target processes, respectively. Both sides are actively involved in the communication, hence the classification as *two-sided communication*. The basic communication workflow for this paradigm is shown in Figure 2.5. Since the introduction of the MPI-1 [25] standard, message passing has become the de-facto standard in most parallel codes. An important concept of two-sided communication is the message matching mechanism. Matching ensures that the next message delivered from the transport layer matches certain criteria.

2.3.1.2 One-sided Communication

The *one-sided communication paradigm*, also called RDMA, RMA or put/get model, conceptually only involves one active process in the communication transaction. In the classical send/receive paradigm, both processes, the source or initiator and the target or destination process, participate in the transaction, while an RDMA transaction features an active and a passive partner. The put operation, also called RDMA write, writes the content of the initiator's local buffer to a specified buffer on the target process's side. The initiator needs to know the address of the remote buffer beforehand. The get or RDMA read operation is the opposite; the initiator requests the contents of a remote buffer to be copied into the specified local buffer. Figure 2.6 illustrates the put and get operations.

2.3.1.3 Remote Load/Store

In contrast to one-sided communication, the *remote load/store paradigm* enables a process to issue communication operations by simply performing a load or store operation to a special address, which in turn triggers a network request. In the event of a load operation, the response carries the requested memory cell which is then used by the NIC to complete the operation. A remote store operation works analogously, though no response is needed as depicted in Figure 2.7. An example of

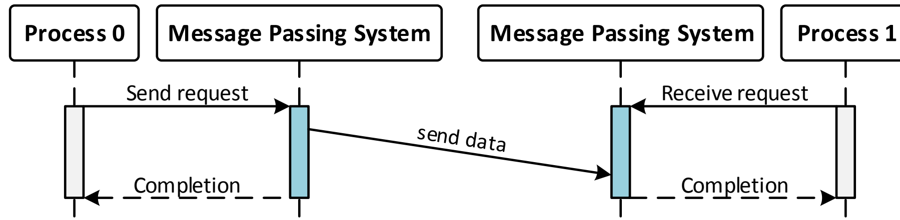


Figure 2.5: Basic send/receive sequence.

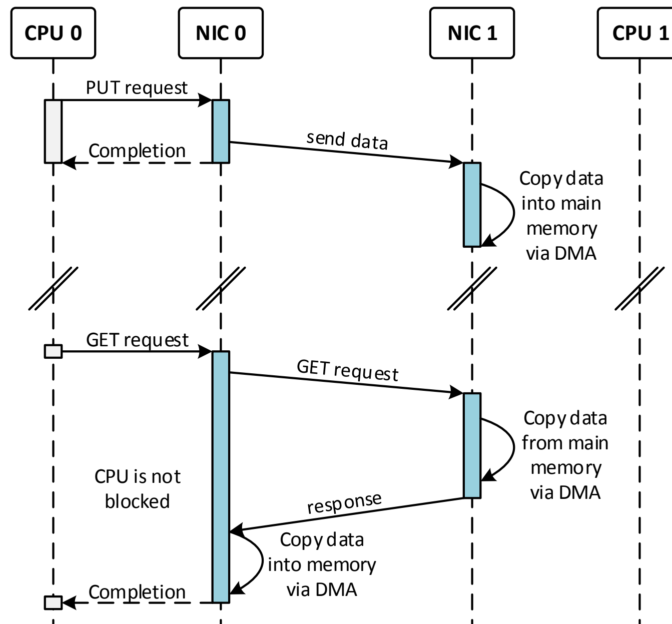


Figure 2.6: Remote Direct Memory Access operations.

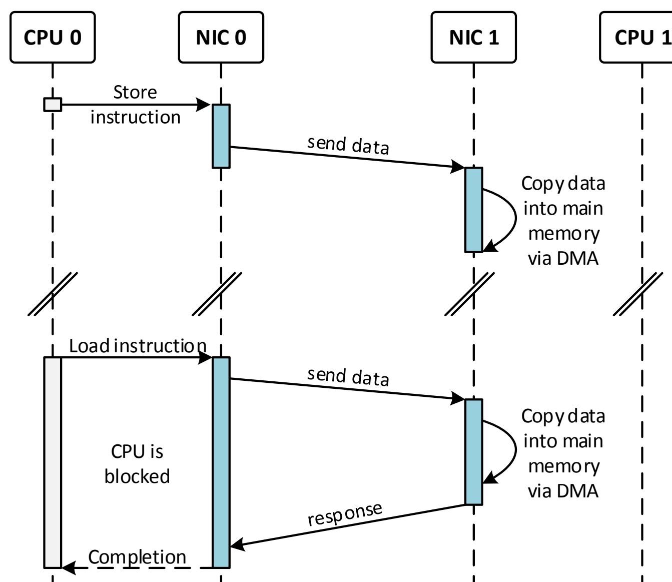


Figure 2.7: Remote load and store operations.

an interconnect standard using the remote load/store paradigm for remote memory access is the *Scalable Coherent Interface* (SCI) [26].

While the remote load/store method looks tempting to many programmers and architects, it suffers from fundamental drawbacks for parallel computing, which are intensified by the characteristics of today's system architectures. First, a CPU load/store operation typically supports only small data movement granularity. This results in relatively small transactions in the register size region (32-128 bits), which limits the network efficiency. Second, long load latencies stall CPU cores, since there is only a limited amount of outstanding memory transactions. This paradigm requires either hardware that supports the forwarding of remote read and write operations or a framework such as a compiler that translates remote memory requests to one- or two-sided communication requests.

2.3.2 Synchronization

An important aspect of communication is the synchronization between initiator and destination side. Synchronization is the enforcement of a defined logical order between events. This establishes a defined time-relation between distinct places, thus defining their behavior in time. Depending on the system architecture or chosen communication model, synchronization can either be *implicit* or *explicit*.

When using two-sided communication, the send and receive scheme ensures an implicit synchronization between remote processes, since both sides are involved in the communication process. When using one-sided communication or remote load/store operations, there is no implicit synchronization between the initiator and destination node. Therefore, explicit synchronization methods such as barriers or global locks are needed. Even though race conditions are much more likely when explicit synchronization is needed, the decoupling of synchronization and data transfer can improve the performance by overlapping communication and computation.

2.3.3 Performance Metrics

Communication or network performance refers to measures of service quality of as seen by the user/application. There are many different ways to evaluate the performance. The most important metrics are bandwidth, latency, message rate, and overhead [27].

Bandwidth The *bandwidth* refers to the maximum rate at which information can be transferred, or in other words, the amount of data that can be transmitted in a

given period of time. *Aggregate bandwidth* refers to the total data bandwidth supplied by the network, and *effective bandwidth* or *throughput* is the fraction of aggregate bandwidth delivered by the network to an application. Typically, bandwidth is measured in bytes per second.

Latency The *latency* or *network delay* describes the minimum time a packet needs to be transferred from one node to another. The total latency of a packet can be expressed by the following equation:

$$\text{Latency} = \text{Sending overhead} + \text{Time of flight} + \frac{\text{Packet size}}{\text{Bandwidth}} + \text{Receiving overhead}$$

Message Rate The *message rate* describes the number of messages that can be sent by a single process in a specified period of time and indicates how well the processing of independent processes can be overlapped. It varies for different message sizes, is limited by the bandwidth and highly depends on the sending and receiving overhead.

Overhead The *communication overhead* describes the time a node, i.e., a processor, needs to send or receive a packet [28], including both software and hardware components. The sending overhead is the time needed to prepare a packet, while the receiving overhead describes the time needed to process an incoming packet.

2.3.4 Interprocess Communication Interfaces

Several different application programming interfaces (APIs) for interprocess communication exist, which utilize the previously described communication schemes. Of particular interest for this work is interprocess communication in distributed memory systems. The following sections present an overview of the most widely used communication standards and corresponding example implementations.

2.3.4.1 Message Passing Interface

The *Message Passing Interface* (MPI) has become the de-facto standard for parallel communication in distributed memory systems. The most popular implementations are MPICH [29], MVAPICH [30], and OpenMPI [31]. In the MPI programming model, a computation comprises of one or more processes that communicate by calling library routines to send/receive messages to/from other processes. An MPI application typically follows the *Single Program Multiple Data* (SPMD) paradigm

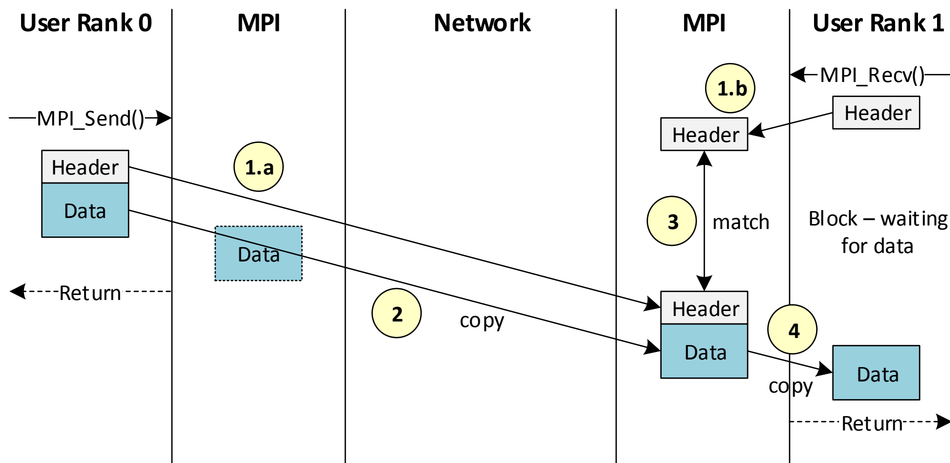


Figure 2.8: MPI eager protocol.

where each program launches multiple processes. A process can be identified by its *rank*, which is a unique identifier that can be used to address other processes in point-to-point communication. Also, multiple ranks can be grouped together in MPI communicators, which restrict communication to ranks within the same communicator. In MPI, a message consists of a header and data, which is basically the payload. The header contains a tag, a communicator, the length, the source rank, plus implementation specific private data. The data can be identified by *naming* and *ordering*. The naming defined by the source rank, communicator, and a tag, which is an arbitrary integer value chosen by the user. If multiple messages are identical in naming, the order in which the messages arrive determines which message is matched with the corresponding receive request.

There are three basic point-to-point communication protocols in MPI: *short*, *eager* and *rendezvous*. In the short protocol, the data is directly sent with its header and no buffering is needed on the sending side. Both eager and rendezvous protocols differ in their buffering scheme and the selection of the protocol that is applied is usually based on the message size. The following introduces the two protocols, but also explains the concept of message matching and collective operations.

Eager Protocol In the eager protocol, the message is sent assuming that the destination can store the data. This approach reduces synchronization delays and simplifies the programming, but may require active involvement of the receiving CPU to drain the network at the receiver's end and may introduce additional copies (intermediate buffer to final destination). Figure 2.8 illustrates the eager protocol. First, the header is written by the sender (1.a) and the receiver (1.b) to the target

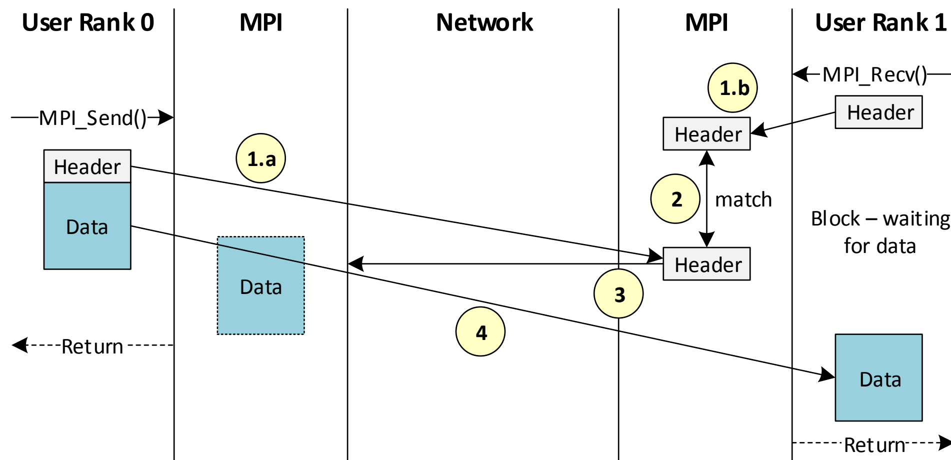


Figure 2.9: MPI rendezvous protocol.

process' mailbox, where it is later matched with the receive request (3). Depending on the payload size, the data is either buffered on the sender side (indicated by the dashed line) or directly sent with the header and buffered in the target's mailbox (2) from where it is copied to its final destination (4).

Rendezvous Protocol While the eager protocol targets smaller payloads, the rendezvous protocol is used for large messages, since the extra copy from the system to the user buffer at the receiver side can significantly add latency and negatively impact performance. When using rendezvous, the message is sent once the target notifies the sender that sufficient user buffer is available. This approach is robust and safe, but requires more complex programming. In addition, it may introduce synchronization delays. Figure 2.9 illustrates the rendezvous protocol. Similar to eager, the header is sent to the target ((1.a) and (1.b)), where it is matched with the receive request (2). Provided a matching receive can be found, MPI returns the target's user buffer address (3) to the sender, which in turn initiates the copy operation (4). Alternatively, the target can use an RDMA transfer, i.e., a get operation, to fetch the data.

Message Matching An important but also complex feature is *message matching*. Message matching guarantees that an MPI code is deterministic by matching message headers and receive requests. The matching algorithm utilizes the source rank, tag, and communicator including wildcards for tag and source. The first message complying with the MPI matching rules is chosen and the data is transferred to the target buffer. Messages that arrive but cannot be matched with already posted

receive requests are added to the so-called *Unexpected Message Queue* (UMQ). Unexpected messages can be avoided by using the rendezvous protocol, which will add an additional roundtrip, but perform true zero-copy.

Collective Operations Apart from the point-to-point communication support, MPI also introduces a set of collective operations, which provides a method of communication which involves participation of all processes in a given communicator. Examples are synchronization calls like `MPI_Barrier`, but also `MP_Broadcast`, `MPI_Alltoall`, `MPI_Gather` or `MPI_Scatter`. One important side effect is that collective communication implies a synchronization point among processes. With the latest release of the MPI standard, MPI-3, non-blocking collective operations were introduced, which aim to maximize the overlap of communication with computation.

2.3.4.2 Partitioned Global Address Space

The *Partitioned Global Address Space* (PGAS) [32] model is a shared memory programming paradigm that introduces the idea to create a global but logically partitioned address space. Parallel processes share one global address space, which consists of memory segments in the local memory of participating processes. For every process, the global address space is divided into a local, private segment and a global, shared segment. While the global memory can be used for communication, the local part cannot be accessed from other processes.

The term PGAS is used for both communication libraries and specialized programming languages. The most well-known PGAS-based programming languages are *Universal Parallel C* (UPC) [33] and *Co-Array Fortran* [34]. PGAS-languages are extensions to common programming models and require special compilers. The compiler is responsible for the data placement. Therefore, access to a local segment is always translated to load/store instructions, while remote accesses can be performed in several ways. Remote load/store operations are able to use local caches to speed up some operations, whereas other solutions are two-sided as well as one-sided communication using an underlying communication framework.

One communication interface often used in conjunction with UPC is the *Global Address Space Network* (GASNet) [35], which provides a high-performance, network-independent interface for PGAS languages. GASNet consists of two layers: the core and the extended API. The core API leverages the Active Message specification and provides atomic operations as well as utility functions. Active Messages provide a low level mechanism in which messages trigger the execution of code on the remote

target. The principle of active messages can be compared to remote procedure calls. The extended API of GASNet provides high-level abstractions to exchange active messages and data, or to ensure synchronization, including data transfer operations based on put/get semantics.

Besides the PGAS-based languages, there exist several communication libraries, which provide direct one-sided put and get operations to and from shared memory regions. In addition to GASNet, *openSHMEM* [36] and *Global Address Space Programming Interface* (GASPI) [37] are representatives of such PGAS-APIs.

2.3.4.3 Remote Procedure Call

Remote procedure call (RPC) packages [38] implement a request-response protocol and follow a simple target [39]: to make the process of executing code on a remote machine as simple and straightforward as calling a local function. An RPC is initiated by the client, which sends a request message to a known remote server to execute a specified procedure with supplied parameters. Once the remote server has processed the request, it sends a response to the client, and the application continues its process. Typically, the client is blocked until the server has finished processing and sent its response, unless an asynchronous request is sent. There are many different RPC implementations, resulting in a variety of different, incompatible RPC protocols. One of the main differences between local and remote procedure calls is that unpredictable network problems can result in a faulty behavior or failure of an RPC. In general, the client must handle such failures without knowing whether the RPC was actually invoked.

2.3.4.4 Portals

The *Portals Network Programming Interface* [40] is a low-level network API for high-performance networking developed by Sandia National Laboratories and the University of New Mexico. Portals provides an interface to support both the MPI standard as well as various PGAS models, such as Unified Parallel C, Co-Array Fortran, and SHMEM, and combines the characteristics of both one-sided and two-sided communication. In addition to the traditional put/get semantics, Portals defines *matching put* and *matching get* operations. The destination of a put is not an explicit address, but a list entry using the Portals addressing scheme that allows the receiver to determine where an incoming message should be placed. This flexibility allows Portals to support both traditional one-sided operations and two-sided send/receive operations with both bypass mechanism for the operating systems as well as the

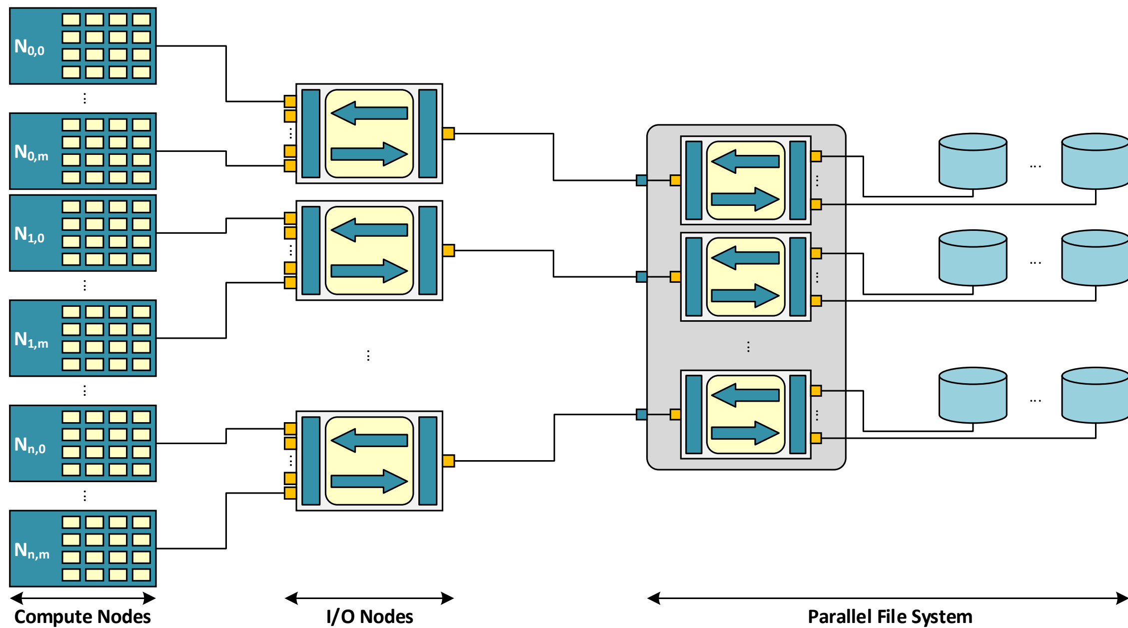


Figure 2.10: Abstract model of HPC system with external file system.

application. Portals does not require activity on the part of the application to ensure progress, which as denoted as *application bypass*.

2.3.4.5 Berkeley Sockets

The *Berkeley Sockets API* [41] is one of the most widely used networking APIs and has been adopted by several different operating systems. The socket concept was developed to provide a generic interface for accessing computer networking resources and processes. Nowadays, sockets are an important Internet API where they are used to implement the Layer 4 of the Internet Protocol, the ISO/OSI layer model. In terms of HPC, sockets are mostly used in data centers. While sockets are not restricted to a specific protocol, the most widely used ones are TCP/IP, UDP/IP and the UNIX domain. Protocols using sockets can either be connectionless (datagram sockets) or connection-oriented (stream sockets).

For connection oriented protocols like TCP, sockets follow the client-server paradigm. This means a server process opens a socket and listens for incoming connection requests issued by the client. Upon receiving a connection request, the server accepts and establishes the connection between two ports. The sockets performance is mainly limited by deficiencies and overhead introduced by lower layers such as the TCP/IP stack and the interaction with the operating system. For example, data needs to be passed through a deep protocol stack including several memory copies and the latency properties of the NIC. An alternative is offered by the stateless, connectionless UDP

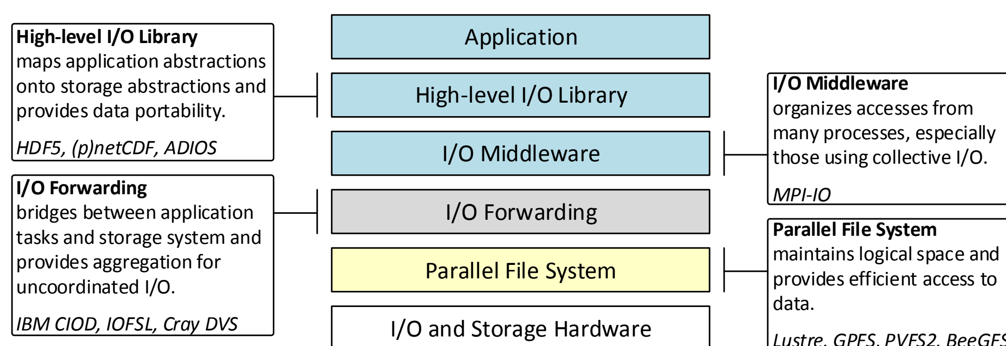


Figure 2.11: Typical parallel I/O architecture for data-intensive sciences [42].

protocol. UDP sockets are able to perform a write- or read-operation by using send or receive. Each packet sent or received on a datagram socket is individually addressed and routed. Order and reliability are not guaranteed with datagram sockets. Matching processes with sockets is much easier compared to the matching mechanism in MPI. Data within a socket reaches the target buffer by addressing a port, which is assigned to a process. There is no active matching needed, because data is always sent to a specified port. Ports are an abstraction of the logical connection between two endpoints.

2.4 Introduction to Parallel I/O

In the context of a HPC systems, parallel I/O [43] describes the ability to perform multiple input/output operations at the same time, for instance simultaneous outputs to storage devices and display devices. It is a fundamental feature of large-scale HPC environments. Modern high performance computing facilities deliver computational and data resources to an ever broadening set of scientific and engineering domains. Large-scale simulation codes stress the capability of file and storage systems by producing large amounts of data in a bursty pattern. With the advent of big data, it is expected that future large-scale applications will generate even more data. Parallel file systems distribute the workload over multiple I/O paths and components to satisfy the I/O requirements in terms of performance, capacity, and scalability.

To meet the required performance and capacity levels of modern HPC facilities, a large-scale parallel I/O environment comprises of multiple layers, as displayed in Figure 2.10, including block devices, the interconnect between the block devices and the file system servers, file system servers and the parallel file system itself, and the interconnect between the file system servers and client nodes. Some configurations will include I/O routers between the file system servers and the clients. These

components are then integrated into usable systems through complex software stacks, as presented in Figure 2.11.

2.4.1 Scientific I/O

Compared to traditional I/O, scientific I/O [44] is performed by large-scale applications from the scientific domain. Typically, scientists think about their data in terms of their science problems, e.g., molecules, atoms, grid cells, and particles. Ultimately, physical disks store bytes of data, which makes such workloads difficult to handle for the storage system. In scientific applications, I/O is commonly used to handle the following data types:

Checkpoint / Restart Files *Application checkpointing* is a technique that adds fault tolerance into a distributed computing system. In general, checkpoint/restart files consist of snapshots, which reflect an application's current state. These snapshots can be used to restart an application in case of system or node failures. The most basic way to implement checkpointing is to periodically stop an application, copy all the required data from the main memory to reliable, persistent storage (e.g., parallel file system), and then continue with the execution.

Two primary categories of techniques can be distinguished: *uncoordinated checkpointing* and *coordinated checkpointing*. Uncoordinated checkpointing employs the idea of saving snapshots of each process independently from each other. The problem imposed by this technique is that simply forcing processes to save checkpoint information at fixed time intervals does not ensure global consistency. Unlike uncoordinated checkpointing, coordinated checkpointing synchronizes the processes' states and dependencies at checkpointing time to ensure that the global state is saved consistently. This is achieved by applying a two-phase commit protocol (i.e., an atomic commit protocol), which results in periodic, bursty I/O phases.

Data Input / Data Output Data input and data output mainly happens at the beginning and end phases of an application run. Large amounts of data need to be read, e.g., to fill the main memory with an initial data set, or need to be written, e.g., to store numerical output from simulations for post-processing analysis.

Out-of-core Algorithms Out-of-core algorithms, also called external memory algorithms, are designed to deal with data sets that are too large to be stored in the available main memory. Such algorithms basically implement "demand paging"

under user application control and are optimized to efficiently fetch and access data that is stored on a slower storage system, e.g., the underlying distributed file system.

2.4.2 Parallel File Systems

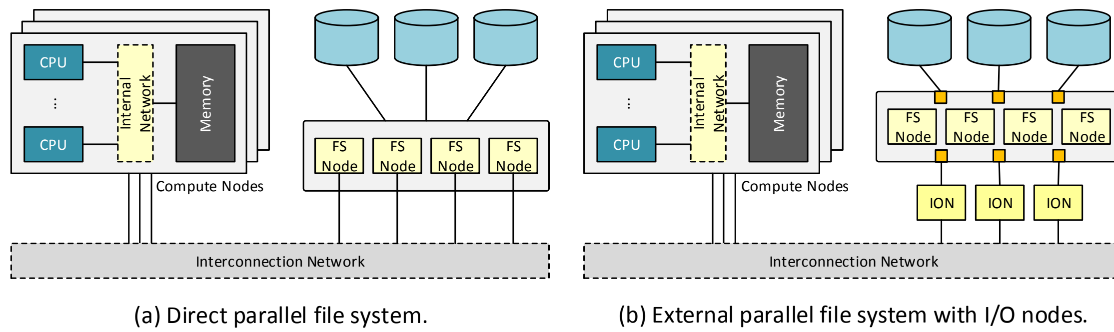
Conventional, local file systems assume that the disk or disks are directly connected to a compute node. In cluster systems consisting of hundreds or thousands of nodes, this would mean that there is no global name space and every node would have to organize its own separate data. To cope with this problem at scale, the file system is outsourced to the network. *Distributed file systems* utilize a network protocol to access the distributed storage. Files are shared between users in a *hierarchical and unified view* [45]: files are stored on different storage resources, but appear to users as they are put on a single location. *Parallel distributed file system architectures* spread individual files across multiple storage nodes, usually for performance and/or redundancy reasons, and allow concurrent read and write access to the files. This mechanism is called *file striping*. Key features [45, 46] of parallel file systems include transparency, fault tolerance, and scalability:

Transparancy The complexity of the underlying distributed storage system is hidden from the user. File access is provided via the same operations and interfaces as used for local file systems, e.g., mounting/unmounting, read/write at byte boundaries, listing directories, and the system's native permission model. Also, fault tolerance mechanisms hide storage system failures from the user.

Fault Tolerance As distributed file systems are deployed on large-scale systems, the possibility of failures are the norm rather than the exception. Therefore, distributed file systems provide fault tolerance mechanisms that keep the system alive in the case of a transient or partial failure. Considered faults are network and server failures, but also data integrity and consistency in case of concurrent file accesses.

Scalability The ability to efficiently scale the system to a large number of nodes and also to effectively leverage dynamically and continuously added system components.

Different architectural approaches can be used to achieve a scalable system architecture [47]. *Client-Server architectures* have several servers which are responsible to manage, store, and share metadata and data by providing a global name space to the clients. *Cluster-based Distributed File Systems* decouple metadata from the data.



(a) Direct parallel file system.

(b) External parallel file system with I/O nodes.

Figure 2.12: Schemes of storage systems attached to a HPC system.

The system comprises of one or more metadata servers and several object servers to store the data. If there is only one metadata server, the system is called *centralized*. *Symmetric architectures* have no distinguished master node, all nodes build the file system in a peer-to-peer like manner. *Asymmetric systems*, on the other hand, have multiple metadata and object data servers, both working on the requests from the clients. Distributed file systems typically use RPCs over the network to invoke read or write operations on a remote node independently from the operating system. Most distributed file systems use the TCP/IP or the UDP protocol to initiate RPCs.

In general, two different parallel file system configurations can be distinguished. Figure 2.12(a) presents the directly connected parallel file system configuration. The file system nodes are directly connected to the interconnection network of the HPC system. The second configuration is displayed in 2.12(b), which represents a center-wide file system. In this system setup, different supercomputing systems are sharing the access to the parallel file system. Each system may deploy a different network technology (e.g., 10GE, Infiniband). Therefore, so called *I/O forwarding nodes* (ION) provide connectivity to the external file system.

2.4.3 High-level I/O Libraries and Middleware

The IEEE *Portable Operating System Interface* (POSIX) [48] defines the primary application programming interface (API) for UNIX variants and other operating systems, along with command line interpreters and utility interfaces, The first standard was released in 1988, when a single computer owned its own file system. The POSIX API presents a useful, ubiquitous interface for basic I/O, but lacks useful constructs for parallel I/O. When using POSIX I/O, a cluster application is basically one program running on N nodes, but it looks likes N programs to the file system. In addition, POSIX has no support for noncontiguous I/O and no hinting

or prefetching mechanism. Together with its rules such as *atomic writes* and *read-after-write consistency*, POSIX I/O has a negative impact on parallel applications. To overcome the limitations of POSIX I/O, high-level I/O libraries and middleware solutions have been developed to provide data abstraction and to organize accesses from many processes, especially those using collective I/O. The following sections introduce MPI-IO and selected libraries.

2.4.3.1 MPI-IO

in distributed memory architectures, parallel I/O systems needs a mechanism to specify collective operations, user-defined data types to describe noncontiguous data layouts in both memory and file, communicators to separate application-level message passing from I/O-related message passing, and non-blocking operations. MPI-IO, which was released in 1997 as part of MPI-2, is an I/O interface specification for use in MPI applications and provides a low-level interface for carrying out parallel I/O. Internally, it uses the same data model as POSIX, i.e., streams of bytes in a file. MPI-IO's features include collective I/O, noncontiguous I/O with MPI data types and file views, non-blocking I/O, and Fortran and additional language bindings. Collective I/O is a critical optimization strategy for reading from, and writing to, a parallel file system. The collective read and write calls force all processes in a communicator to read/write data simultaneously and to wait for each other. The MPI implementation optimizes the read/write request based on the combined requests of all processes and can merge the requests of different processes for efficiently servicing the requests. Given N processes, each process participates in reading or writing a portion of a common file.

There are several different MPI-IO implementations. One of the widely used ones is *ROMIO* [49] from the Argonne National Laboratory. ROMIO leverages MPI-1 communication and supports local file systems, network file systems, and parallel file systems such as GPFS and Lustre. The implementation includes data sieving techniques and two-phase optimizations.

2.4.3.2 HDF5 and Parallel HDF5

The *Hierarchical Data Format v5* (HDF5) [50] consists of three components: a data model, an open source software, and an open file format. The HDF5 file format is defined by the HDF5 file format specification [51] and defines the bit-level organization of an HDF5 file on storage media. HDF5 is designed to support high volume and/or complex data, every size and type of system (portability), and

flexible, efficient storage and I/O. An HDF5 file is basically a container consisting of data objects, which can either be datasets or groups. Datasets organize and contain multidimensional arrays of a homogeneous type, while groups are container structures consisting of datasets or other groups.

Altogether, HDF5 provides a simple way to store scientific data in a database-like organization. The data model complements the file format by providing an abstract data model independent of the storage medium or programming environment. It specifies the logical organization and access of HDF5 data from an application perspective, and enables scientists to focus on high-level concepts of relationships between data objects rather than descending into the details of the specific layout. The HDF5 software provides a portable I/O library, language interfaces, and tools for managing, manipulating, viewing, and analyzing data in the HDF5 format. The I/O library is written in C, and includes optional C++, Fortran 90, and high-level APIs. For flexibility, the API is extensive with over 300 functions. *Parallel HDF5* is a configuration of the HDF5 library, which can be used to share open files across multiple parallel processes. It uses the MPI standard for interprocess communication.

2.4.3.3 ADIOS

The *Adaptable I/O System* (ADIOS) [52, 53] is an extendable framework which allows scientists to plug-in several different I/O methods, data management services, file formats, and other services such as analytic and visualization tools. It provides a simple I/O application programming interface (API) and allows the usage of different computational technologies to achieve good, predictable, performance. ADIOS provides a mechanism to externally describe an application's I/O requirements using an XML-based configuration file. One of its salient features is that I/O methods can be exchanged between runs by modifying the configuration file without the need to modify or recompile the application code. ADIOS has demonstrated impressive I/O performance results on leadership class machines and clusters, and has been deployed on several supercomputing sites including the Argonne Leadership Computing Facility (ALCF), the Oak Ridge Leadership Computing Facility (OLCF), the National Energy Research Scientific Computing Center (NERSC), the Swiss National Supercomputing Centre (CSCS), Tianhe-1 and 2, and the Pawsey Supercomputing Centre.

ADIOS is implemented as a user library. It can be used like any other I/O library, except that it has a declarative approach for I/O. The user defines in the application source code the "what" and "when" while the framework takes care of the "how". There are two key ideas. First, users do not need to be aware of the low-level layout

and organization of data. Second, application developers should not be burdened with optimizations for each platform they use. It is capable of I/O aggregation on behalf of the application to increase the I/O performance and scalability.

2.4.3.4 SIONlib

The *Scalable I/O library* (SIONlib) [54] implements the idea of writing and reading binary data to or from several thousands of processors into one or a few physical file(s). In other words, SIONlib maps the task-local I/O paradigm onto shared I/O. SIONlib is implemented as an additional software layer in between the parallel file system and a scientific application, and provides an extension to the traditional POSIX or standard C file I/O API. To enable parallel access to files, SIONlib provides collective open and close functions, while writing and reading files can be done asynchronously. Its main application area are internal file formats such as scratch files and checkpointing files. SIONlib requires only minimal changes to the application source code, mainly to the open and close function calls. Since each task writes its own data segment to the same file, SIONlib assigns different regions of the file to a task. This minimizes the file lock contention and I/O serialization for collective I/O. In order to do so, SIONlib needs to know the estimated (or known) data size to place the data accordingly. If a write call exceeds the data size specified in the open call, SIONlib moves forward to next chunk. SIONlib provides support for different languages including C, C++, and Fortran, and supports MPI, OpenMP, and MPI+OpenMP.

2.4.4 Access Patterns

Typically, scientific I/O comprises of large amounts of data written to files in a structured or even sequential *append-only* way. Most HPC storage systems employ a parallel file system such as Lustre or GPFS to hide the complex nature of the underlying storage infrastructure, e.g., spinning disks and RAID arrays, and provide a single address space for reading and writing to files. There are three common I/O patterns [44] used by applications to interact with the parallel file system.

2.4.4.1 Single Writer I/O

In the *single writer I/O* pattern, also known as sequential I/O, one process aggregates data from all other processes, and then, performs I/O operations to one or more files. The ratio of writers to running processes is 1 to N, as depicted in Figure 2.13.

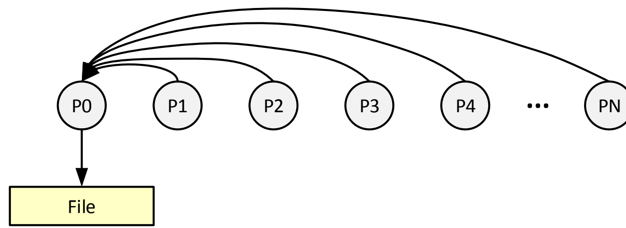


Figure 2.13: Single writer I/O pattern.

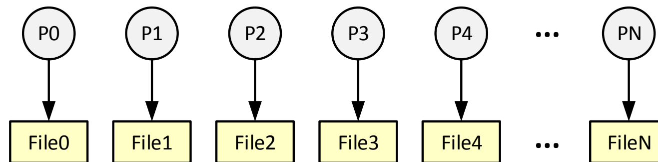


Figure 2.14: File-per-process I/O pattern.

This pattern is very simple to implement and easy to manage, and can provide a good performance for very small I/O sizes. But, single writer I/O does not scale for large-scale application runs since it is limited by a single I/O process. It performs inefficiently for any large number of processes or data sizes leading to a linear increase in time spent in writing.

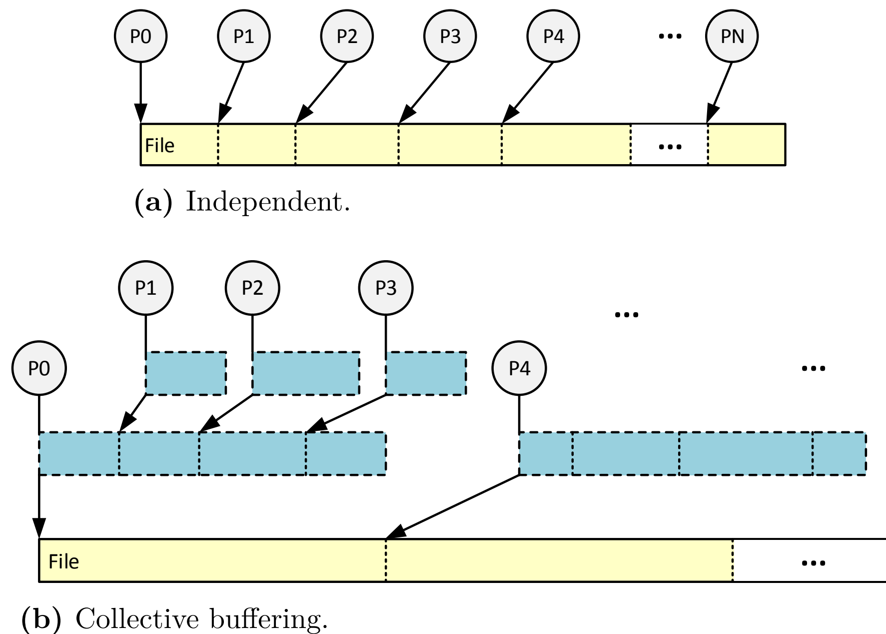
2.4.4.2 File-Per-Process I/O

In the *file-per-process I/O* pattern, each process performs I/O operations on to individual files as shown in Figure 2.14. If an application runs with N processes, N or more files are created and accessed (N:M ratio with $N \leq M$). Up to a certain point, this pattern can perform very fast, but is limited by each individual process which performs I/O. It is the simplest implementation of parallel I/O enabling the possibility to take advantage of an underlying parallel file system.

On the downside, file-per-process I/O can quickly accumulate many files. Parallel file systems often perform well with this type of I/O access up to several thousands of files, but synchronizing metadata for a large collection of files introduces a potential bottleneck, e.g., even a simple `ls` can break when being performed on a folder containing thousands of files. Also, an increasing number of simultaneous disk accesses creates contention on file system resources.

2.4.4.3 Single Shared File I/O

The *single shared file I/O* pattern allows many processes to share a common file handle but write to exclusive regions of a file. Figure 2.15a displays the *independent*



(a) Independent.

(b) Collective buffering.

Figure 2.15: Single shared file I/O patterns.

variant of this I/O pattern, where all processes of an application write to the same file. The data layout within the file is very important to prevent concurrent accesses to the same region. Contesting processes can introduce a significant overhead, since the file system uses a *lock manager* to serialize the access and guarantee file consistency. The advantage of the single shared file I/O pattern lies in the data management and portability, e.g., when using a high-level I/O library such as HDF5.

Figure 2.15b displays the *collective buffering* variant of this pattern. This technique improves the performance of shared file access by offloading some of the coordination work from the file system to the application. Subsets of processes are grouped in so called *aggregators* performing parallel I/O to shared files. This increases the number of shared files and improves the file system utilization. Also, it decreases the number of processes which access a shared file, and therefore, mitigates file system contention.

Chapter 3

Extoll System Environment

Interconnection networks are a key component in large-scale HPC deployments and play a vital role in the overall system performance. They must offer high communication bandwidth and low latency to cope with the nature of the mixed system workloads, which include data-intensive and communication-intensive applications. Essentially, networks are the backbone for both communication and I/O, and connect compute nodes, I/O devices, and storage devices. The design of future HPC interconnect technologies faces multiple challenges, including scalability, efficiency, heterogeneity, resiliency, virtualization, cost and power consumption.

Extoll is such a high-performance interconnect technology [55, 56], which started off as a research project at the Heidelberg University in 2009. Its key objective is to fit the needs of future HPC applications in terms of latency, bandwidth, message rate [57], and scalability. Released in 2016, the first *Application Specific Integrated Circuit* (ASIC) version of Extoll is the Tourmalet chip. Extoll has been chosen as the fundamental technology for the research work presented in Chapters 4 through 6 and is described in the following sections.

3.1 Technology Overview

The top-level diagram of the Extoll technology is displayed in Figure 3.1. The design is divided into three parts: the host interface, the network interface, and the network.

The network part of the NIC provides six links to build a 3D torus network topology. The Extoll NIC implements a direct network approach by integrating a switching logic on-chip, which removes the need for external switches. An additional

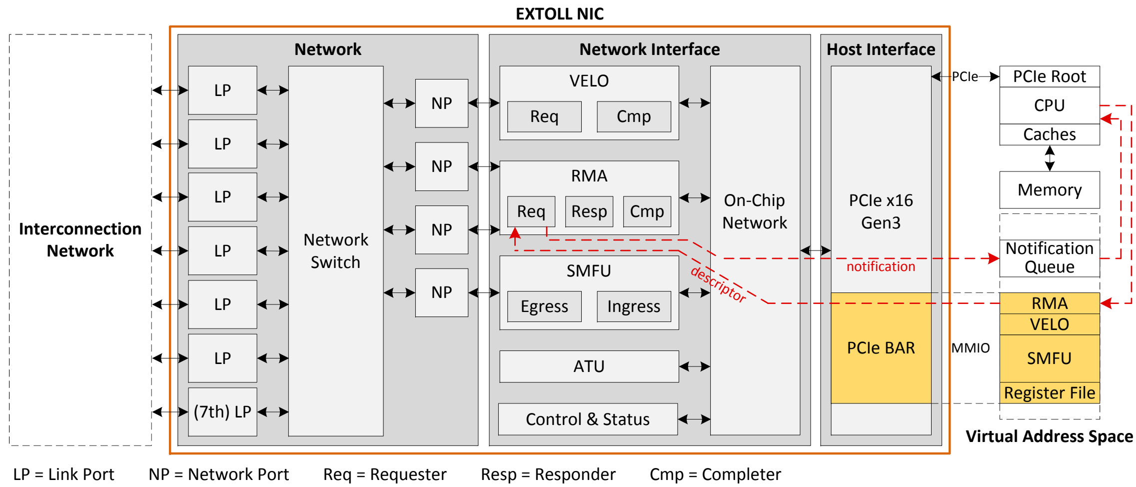


Figure 3.1: Overview of the Extoll NIC top-level diagram.

7th link can be used to connect network-attached memory [58], accelerators or special purpose cards to the network, or to provide network connectivity to the outside world without breaking up the torus topology. The link ports are connected to a crossbar (network switch) that either routes messages between the link ports, or from the network to the functional units. The crossbar provides features such as virtual channels, packet retransmission, multi-cast groups, and table-based routing. Each crossbar port has its own local routing table. Routing entries specify whether to forward the incoming packets to the NIC's network interface units or to reroute the packet to another crossbar port, which forwards the packet to a node that is closer to its destination. Along with deterministic routing for each link port, adaptive routing is supported to avoid network congestion and to improve network utilization.

The Tourmalet chip comes with a PCI Express Gen3 x16 host interface. An on-chip network with high throughput and low latency connects the functional units of the network interface with the host interface. Another feature of the Extoll NIC is that it can be configured to act as a PCIe root port. With this functionality, the NIC can configure other PCIe endpoint devices connected to its host interface.

The network interface provides the hardware support for different communication models, including *Remote Direct Memory Access* (RDMA), *Message Passing Interace* (MPI), and *Partitioned Global Address Space* (PGAS).

3.2 Functional Units

The hardware support for different communication models is realized through a number of different functional units, which reside on the network interface.

Table 3.1: Overview of possible RMA notification type combinations.

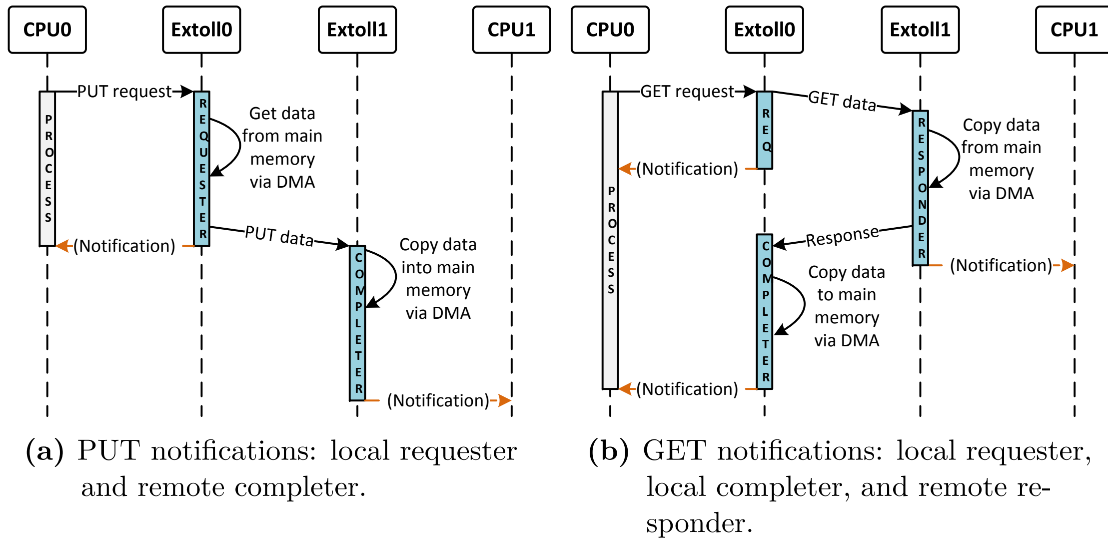
Command	Requester	Completer	Responder	
PUT	0	0	0	No notifications
	1	0	0	One-sided PUT
	1	1	0	Two-sided PUT
GET	0	0	0	No notifications
	0	1	0	One-sided GET
	0	1	1	Two-sided GET

3.2.1 Remote Memory Access Unit

The *Remote Memory Access* (RMA) [59] unit is one of the main functional units in the Extoll network protocol and provides a throughput oriented design for middle to large message sizes. The unit allows remote direct memory access based on either virtual (network logical) or physical addresses with PUT and GET operations. For this purpose, the unit implements a DMA engine for CPU offloading. Using network logical addresses (NLAs) ensures that the memory region is pinned and cannot be swapped out of the main memory, and also safeguards interprocess security. The work requests queue resides on the NIC and is mapped into the processors virtual address space. When a descriptor is written to the work requests queue, the NIC begins with the execution of a request as soon as the last word of the descriptor is received. After completion, a notification is written to the system notification queue, which resides in the main memory and implements a double-linked list structure. In Figure 3.1, the red, dashed line highlights the path of an RMA operation, including the generation of a requester notification.

3.2.1.1 Notification Mechanism

Notifications consist of two 64-bit words and inform whether data has been sent or received. They can also be used to inform remote processes that a PUT or GET operation has completed. The processor can query the queue for new notifications. To consume a notification, the processor has to explicitly remove it in a First In First Out (FIFO) manner. Depending on the targeted communication model, the RMA unit is capable of triggering different types of notifications. For example, a notification can be written upon completion of a work request or when new data has been received by the NIC. This way, notifications can be generated on both origin and target side. Table 3.1 displays possible notification combinations for PUT and GET operations. With the RMA unit's capability to generate different



(a) PUT notifications: local requester and remote completer.

(b) GET notifications: local requester, local completer, and remote responder.

Figure 3.2: Overview of notification mechanism for Extoll PUT and GET.

types of notifications, different communication models can be mimicked by the RMA. While one-sided communication only requires notifications on one side, a message passing scheme can be emulated by generating notifications on both sides. Figure 3.2 provides a sequence diagram for both RMA PUT and RMA GET operations, including possible notifications.

3.2.1.2 From Software to Network Transactions

The RMA functional unit is split into three units: the *requester*, the *responder* and the *completer*. Work requests issued by the processor are received by the requester. The requester translates work requests into network packets, which are sent to the target node. On the target side, the responder creates responses. For instance, a get operation triggers the requester to send a data request to the target node's responder, which in turn responds by initiating the data transfer. On both origin and target side, the completer performs the DMA transfer. In a put operation, the completer is only involved on the target side where it writes the data to the main memory.

Every RMA transaction is initiated by a software, which assembles a software descriptor describing the desired transactions and writes it to one of the descriptor queues of the Extoll device. Figure 3.3 presents an example software descriptor layout for byte *sized PUT/GET requests*. Byte sized requests only write as many bytes to memory as defined in the header. This is done by creating a byte mask, however the byte region must be consecutive, so no stride or other read/write patterns are supported. For GET requests, the payload will be read from the destination node

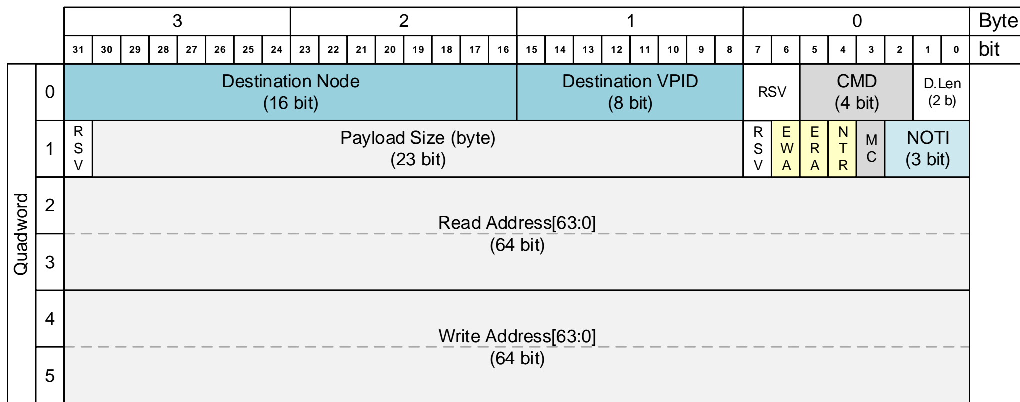


Figure 3.3: Sized PUT/GET software descriptor format [60].

and the GET response will write the data into the memory of the issuing node. For PUT requests, the payload is read from the issuing node and then written to the destination node's memory. Another command supported by the RMA unit is the *Immediate PUT request*, which embeds the payload in the software descriptor and provides small data transfers (72 bit) without involving the local DMA engine. This mechanism can be useful, e.g., to access a remote Extoll register file.

An RMA transaction initiated by a software descriptor can move up to 8 MB. For larger data transmissions, multiple RMA transactions must be triggered by writing additional software descriptors. Internally, an RMA transaction is split in 512 byte packets, 16 bytes are reserved for the network descriptor, which leaves 496 bytes for the payload. The payload size is also referred to as *RMA MTU*.

3.2.2 Virtualized Engine for Low Overhead Unit

The *Virtualized Engine for Low Overhead* (VELO) [61] is another of the main functional units and is optimized to provide efficient transmission for small messages in a send/receive style using one DMA ring buffer per *virtual process identifier* (VPID) on the receiving side. Small messages require a very low latency compared to larger payloads, where a longer delay is more acceptable since it can be compensated by exploiting the benefit of high bandwidth. Thus, VELO complements RMA in offering minimized latency for applications relying on heavy use of smaller messages.

A typical VELO message transfer can be described as follows. First, a software descriptor, as depicted in Figure 3.4, is written to a network request queue. It carries several pieces of information such as the target VPID, target node, message length and message type tag (MTT), which can be used to differentiate software dependent message types. If the multicast (MC) bit is set, the target node is

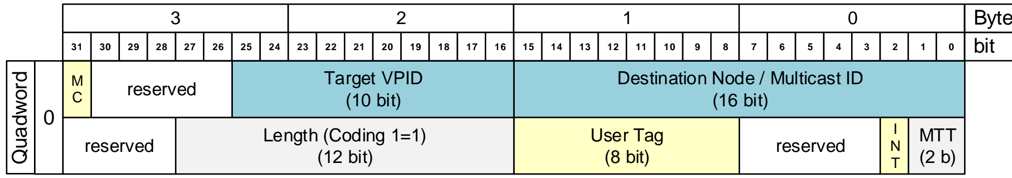


Figure 3.4: VELO message software descriptor format.

interpreted as a multicast group ID and the crossbar can match the multicast ID to the corresponding entry in the routing table to forward the message accordingly. After writing the software descriptor, messages are posted automatically and performed as PIO operations by the CPU. The sender copies the data from the original buffer to a requester address, which decodes to the VELO requester. On the receiving side, the DMA ring buffer corresponding to the target VPID is divided into slots of 128 bytes. When a receiving a new message, the VELO completer writes a status word (8 bytes in size) to the first quadword of the slot. The rest of the slot is filled with the payload. The maximum VELO packet size is 128 bytes with 8 bytes reserved for the status word (network descriptor).

The VELO unit combines several different techniques including stateless work processing, secure and atomic PIO message triggering from user space, integrated end-to-end flow control, and virtual cut through in all pipeline stages.

3.2.3 Virtual Process ID

In the Extoll system environment, user and kernel level processes initiate RMA and VELO transmissions by writing a software descriptor either to the work request queue residing on the Extoll NIC or to a dedicated area in the main memory respectively. In order to ensure process security, Extoll utilizes a resource virtualization mechanism called *virtual process identifiers* (VPID), which provides process separation.

Each VPID identifies a process or a group of processes, which can communicate with each other. A process can only access memory registered for its VPID. In addition, an in-memory mailbox is allocated for each registered VPID. The Extoll hardware writes hardware-specific notifications to these memory areas. The VPID is assigned by the Extoll device driver and stored in hardware, and must be requested by a process before any communication can be performed. Extoll is designed for systems with up to 256 processors. Therefore for both RMA and VELO, 256 VPIDs are available. For each VPID, one 4 KB page is mapped into the main memory.

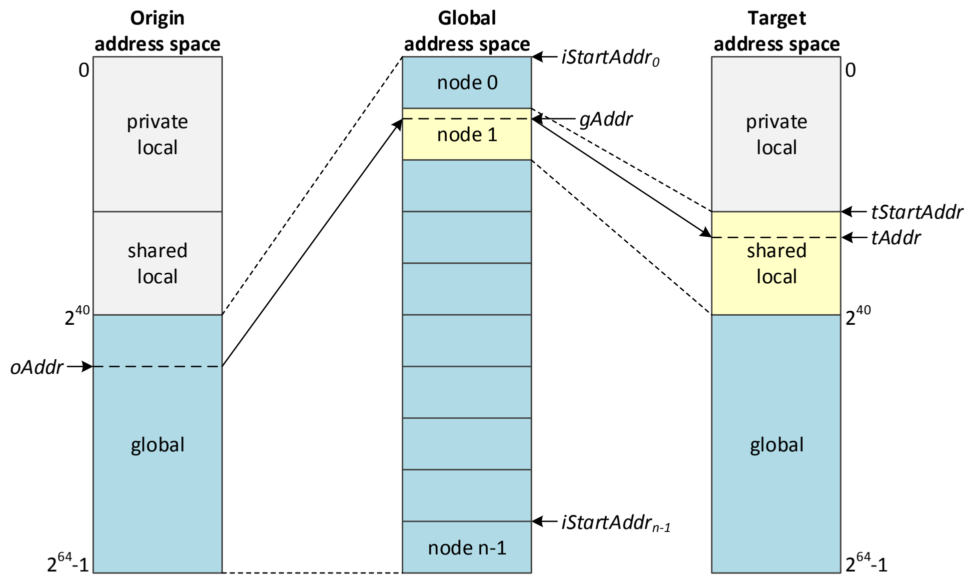


Figure 3.5: SMFU's address space layout for the interval configuration [62].

3.2.4 Shared Memory Functional Unit

While the RMA and VELO units provide hardware support for one- and two-sided communication, the *Shared Memory Functional Unit* (SMFU) [62] allows to span a virtual shared address space across physically distributed system nodes and provides the hardware support for distributed global address spaces. Local load/store operations to the shared address space are transparently encapsulated in network packets and forwarded over the Extoll network to the corresponding remote node. An example use case is the *Partitioned Global Address Space* (PGAS) model. The SMFU aggregates exported memory segments from different nodes into a single, non-coherent, distributed shared memory space. Figure 3.5 illustrates an example SMFU address space layout. A node's address space is divided in a local and a global memory partition. The left side displays the address space layout of the origin node. The target node's address space is shown on the right. In this example, all addresses between 2^{40} and $2^{64} - 1$ span the global address space. The memory itself is divided into intervals; every memory interval corresponds to one remote node.

The SMFU uses physically contiguous and pinned memory regions for the shared local partitions on the target side to avoid logical to physical address translations. The CPU triggers remote memory access by a load or store operation, which is translated into a programmed I/O (PIO) read or write operation on the global address space. Ultimately, the load/store operation is encapsulated in a network packet and sent over the network to the target node.

Originally, *HyperTransport* (HT) [63] was chosen as the interface between the SMFU's communication engine and the host system, but has been replaced with PCI Express. The main reason for this is that HT is not available anymore, since AMD changed their processor design. The communication engine passes transactions (e.g., non-posted reads and posted writes) from the origin to the target node. A transaction can be divided in several sub-tasks, including target node determination and address translation. There are three different address spaces in the SMFU architecture: one global address space and two local address spaces (target and origin). All three address spaces are traversed by a remote memory access. To avoid unnecessary latency overhead, a simple yet efficient translation scheme is used.

The simplest way to utilize the SMFU is the *interval* configuration. For each interval, the start (*iStartAddr*) and end (*iEndAddr*) addresses, and the respective nodes are stored in the Extoll register file. In addition, the target address (*tStartAddr*) is configured, which points to the start of the local shared memory segment and incoming requests are forwarded to this address. As depicted in Figure 3.1, the SMFU has two sub-units, the *egress* and the *ingress unit*. Load and store requests to an address (*oAddr*) residing in the SMFU BAR are forwarded to the egress unit. For example if this address hits interval n, the global address (*gAddr*) can be calculated by subtracting the start address of interval n ($iStartAddr_n$) from the origin start address (*oAddr*), as shown in Equation 3.1.

$$gAddr = oAddr - iStartAddr_n \tag{3.1}$$

After matching *gAddr* with the node the interval is assigned to, the load/store operation is transparently encapsulated in a network packet. On the target node, the ingress unit calculates the target destination address (*tAddr*) in the following way:

$$tAddr = (gAddr + tStartAddr) \tag{3.2}$$

The interval configuration only supports a limited number of intervals, currently 64 intervals. To cope with this limitation, it is possible to use an address mask, which encodes the target node in the global address (*gAddr*). The target node identifier is then determined by (1) applying a bit mask to the global address, and then, (2) performing a bitwise right shift operation (refer to Equation 3.3). The resulting value represents the target node identification (*tNodeID*).

$$tNodeID = (gAddr \;\&\; mask) \;>>\; shift_count \tag{3.3}$$

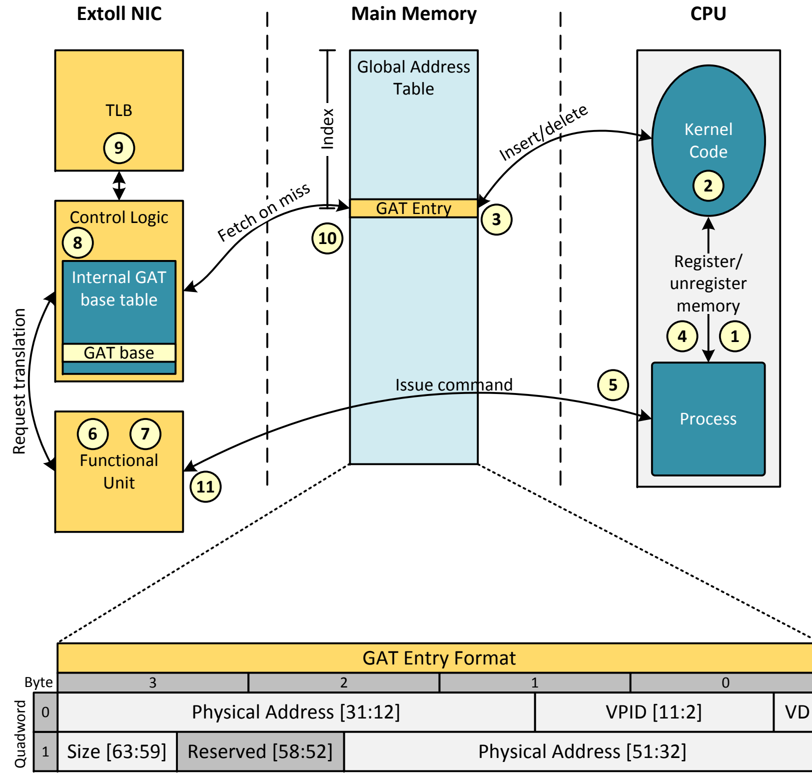


Figure 3.6: ATU address translation overview [55].

On the target side, the target address (*tAddr*) can be computed by (1) applying the inverted mask to the global address, and then, (2) adding the start address of the target local shared memory region (*tStartAddr*):

$$tAddr = (gAddr \ \& \ \sim mask) + tStartAddr \tag{3.4}$$

3.2.5 Address Translation Unit

The *Address Translation Unit* (ATU) [55, 64] translates physical addresses residing in the main memory into *Network Logical Addresses* (NLAs), which are mapped into the address space of a functional unit. An NLA can be compared to a virtual address in the context of the operating system. The ATU basically acts as a memory management unit for the Extoll NIC.

Before an NLA can be used, it has to be decoded, because the translations are based on *Network Logical Page Addresses* (NLPA). An NLPA logically addresses a memory area called *Network Logical Page* (NLP), which by default has the size of a single 4 KB page, but also supports 8 KB, 64 KB, 2 MB, and 1 GB page sizes. NLAs are mainly used for bulk data transfers through the RMA unit.

The ATU features two levels of translation. The first level is performed on chip in the *internal global address (GAT) table*, which is implemented as an SRAM and can hold up to 1024 entries. Each entry can point to an in main memory GAT, which is 2 MB in size and can map up to 2^{18} NLPs. Figure 3.6 displays the layout of an NLA. In addition, it outlines the address translation process. The following steps describe how a process registers and uses NLAs [55]:

(1) A process invokes the memory registration function of the kernel code.

(2) The kernel code allocates an empty GAT entry (GATe) and writes it into the main memory. An entry consists of the physical page address, the VPID, access restriction bits (denoted as VD), and the actual page size. For example if the size field contains a value of nine, the NLP size is $2^9 * 4$ KB = 2 MB.

(3) The index of the allocated GATe in the GAT is returned to the process.

(4) The requesting process can now use the GATe index to communicate the memory location to the Extoll device.

(5) In order to issue a request, the process sets up a software descriptor, which references the mapped memory by passing the GATe index, also referred to as *software NLA*. The VPID is passed indirectly through the functional unit by means of the physical address space used to issue the command.

(6) The request arrives at a functional unit.

(7) The functional unit forwards the GATe index and the VPID to the ATU.

(8) The ATU performs a translation look-aside buffer (TLB) lookup. In case of an TLB miss, the base address of the GAT to be used can be calculated by using the high-order bits of the NLA as index into the GAT base table.

(9) The TLB performs a lookup using the NLA and verifies the VPID and access permissions. If the TLB lookup returns successfully, the entry can be used.

(10) Otherwise, a single 64-bit read operation from the main memory is sufficient to return the corresponding NLA to the TLB.

(11) Upon successfully completing all checks, the translation is returned to the requesting functional unit, which continues with processing the request.

Two types of physical buffers can be pinned over the ATU: physically contiguous memory windows and pinned page lists. Page lists must consist of filled pages. The data can start at an offset into the first page (first byte offset) and can end on a non-page boundary, i.e., the last page may be partially filled. The ATU can be used to mimic scatter/gather I/O.

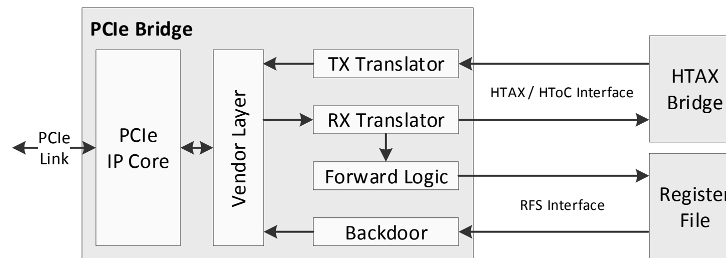


Figure 3.7: Overview of the PCIe Bridge unit [65].

3.2.6 Register File

The Extoll *Register File* [64], also known as *Control and Status* unit, serves as the central component and storage for control and status functions such as configuration settings, module debug information, and performance counters. Complex hardware designs such as Extoll have many different configuration settings and control options, which need to be adjusted in order to use the hardware. The register file can be used to configure the Extoll NIC, but also the retrieve debug and status information.

3.2.7 PCIe Bridge

The PCIe Bridge [65] is located in the on-chip network of Extoll's network interface. The functional unit is responsible for translating PCIe packets into *HyperTransport On-Chip* (HToC) [66] packets and vice versa. An overview of the unit is shown in Figure 3.7. The module consists of several submodules. Of interest for this work are the *Register File*, the *PCIe Backdoor interface*, and the *Forward Logic*.

PCIe Bridge Register File The PCIe Bridge is connected to the register file, which among other things stores the buffer space for packets that are either received by the forward logic or sent by the PCIe backdoor. The PCIe Bridge register file is implemented as a random access memory (RAM). Table 3.2 presents an overview of the registers that are needed to configure and use the PCIe backdoor.

PCIe Backdoor The *PCIe Backdoor* is used to inject software controlled packets into the outgoing PCIe traffic for configuration or debug purposes. The PCIe Backdoor has an internal 64-bit wide RAM to store application controlled packets. The RAM is managed by the PCIe Backdoor, but also has a connection to the register file. The PCIe Backdoor reads packets from the internal RAM and the register file writes packets to it. The internal RAM has only one port for read and write access so overlapping requests from both sources have to be prioritized. A finite

Table 3.2: Overview of PCIe backdoor registers needed for configuration.

Register Name	Description
`pcie_to_htoc_cfg`	This register stores the control information about which packets should be extracted from the PCIe receive traffic and stored to the `pcie_to_htoc_data` register.
`htoc_to_pcie_backdoor_control`	To initiate a PCIe backdoor transaction, an application needs to write the length of the packet to this register followed by setting the send bit to one.
`htoc_to_pcie_backdoor_data`	This register serves as a data buffer for PCIe packets, which will be injected by the PCIe backdoor. It can hold a four double word header and up to 32 double words of payload.
`pcie_to_htoc_backdoor_info`	To inform an application that a PCIe packet has been forwarded to and stored in the register file, this register holds a valid bit that must be polled by the application. It also stores the number of read accesses to the `pcie_to_htoc_backdoor_data` register, which are necessary to retrieve the whole packet. The register is implemented as a first in, first out (FIFO) buffer. Each read access to the register removes one entry from the FIFO (destructive read).
`pcie_to_htoc_backdoor_data`	This register allows access to the PCIe to HTOC backdoor receive interface, which is implemented as a FIFO buffer. A read returns the first value of the FIFO and removes the entry from the buffer.

state machine prioritizes read requests from the PCIe Backdoor over write requests from the register file. After the PCIe Backdoor has injected the packet into the outgoing traffic, the outstanding write requests from the register file are processed.

Forward Logic The *Forward Logic* is located in the RX translator. According to the options specified in the `pcie_to_htoc_config` register, PCIe packets from the received PCIe traffic are either stored in the `pcie_to_htoc_backdoor_data` register, forwarded to the HTAX Bridge or dropped.

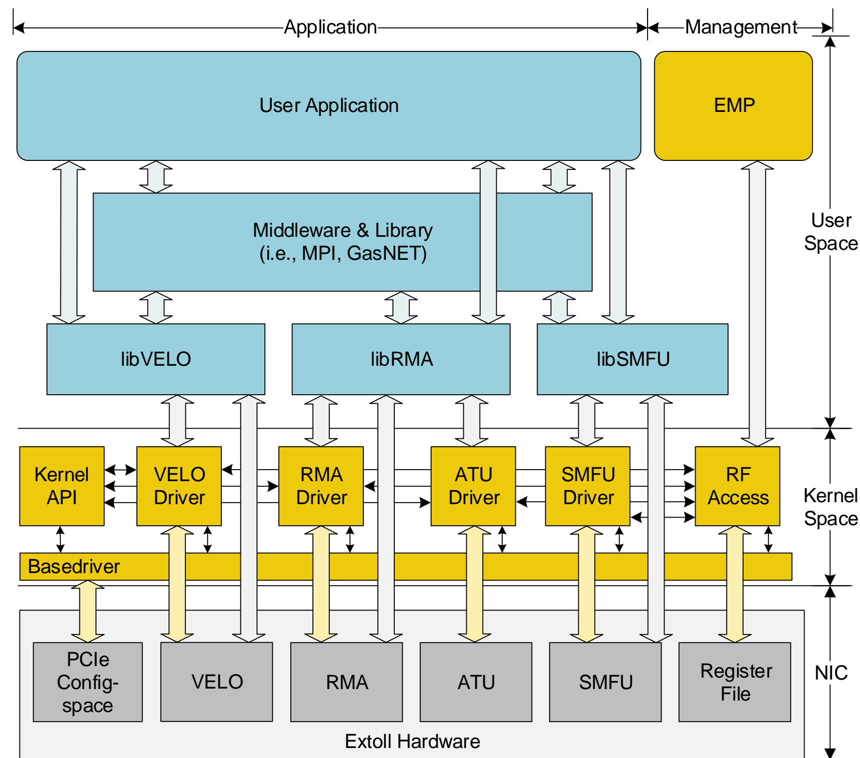


Figure 3.8: Overview of the Extoll software environment.

3.3 Software Environment

The Extoll software environment comprises of several different layers, as depicted in Figure 3.8. The following sections describe the kernel and user space software components, but also introduce the Extoll management program.

3.3.1 Kernel Space

The kernel space side of the software stack consists of several kernel modules. *extolldrv* is the base driver, which controls the hardware and initializes the device. The other modules are built on top of it to provide different functionality and access to the functional units.

velodrv implements the kernel interfaces used by libVELO and the kernel API. *rmadrv* implements the kernel interfaces used by libRMA and the kernel API. *extoll_rf* provides the sysfs interface to the register file. *smfudrv* provides access to the SMFU functionality and resources. The *kernel API* module, as indicated by the name, provides a kernel API, which exports functions for RMA and VELO to be accessed by other kernel code and modules.

To react to the different interrupt sources, Extoll's interrupt handler can register callback functions for the different trigger event groups. This allows not only to react to internal events causing the interrupt, it can also be used to install callback functions to handle interrupts received from a device connected to the remote NIC. Registering and de-registering these callbacks is part of the kernel API too.

3.3.2 User Space

To expose Extoll functionalities to user space applications, Extoll provides three libraries, libVELO, libRMA and libSMFU, which provide an API to Extoll's respective functional units. They allow to establish connections to a process on the remote node, send and receive messages and finally de-register the connection.

Extoll exposes its register file to the user via the pseudo file system *sysfs* provided by the Linux kernel, creating one file for every register in the register file.

The user space software stack also provides a native Extoll OpenMPI implementation by providing a *Matching Transport Layer* (MTL) for Extoll, which is implemented on top of libRMA and libVELO. An OpenMPI MTL typically provides device-layer support for transfer of MPI point-to-point messages over devices that support hardware or library message matching.

3.3.3 EMP: Network Discovery and Setup

The *Extoll Management Program* (EMP) serves to fulfill tasks such as routing setup, network surveillance, and management of network resources like multicast groups. Extoll can be used to build commonly used topologies such as mesh and torus or individual non-standard ones. Either way, each NIC's routing table must initially be configured for network operation.

EMP supports two different types of network configuration modes: discovery based and topology file based. The discovery based mode provides an auto-detection mechanism of connected Extoll devices and creates the topology automatically. In addition, the topology file based mode can be used to verify that all devices are connected in the expected way or to setup customized network topologies. For both modes, EMP assigns unique node identifiers to every NIC and initializes the routing table entries according to the desired routing scheme.

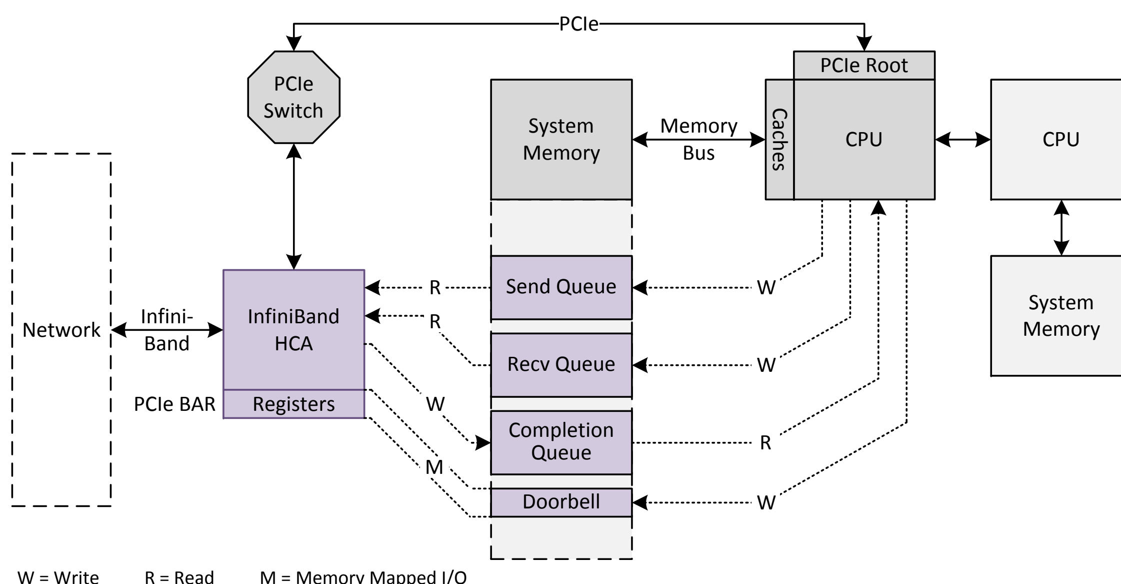


Figure 3.9: Overview of Infiniband HCA system integration.

3.4 Related Interconnection Standards

While the Extoll technology forms the basis for this work, the Infiniband interconnect and PCI Express are two fundamental interconnection standards in today's HPC systems. They key concepts are introduced in the following sections.

3.4.1 Infiniband

The Infiniband architecture [67] is an industry standard developed by the InfiniBand Trade Association (IBTA) and was originally designed to replace the Peripheral Component Interconnect (PCI) and Ethernet in HPC systems. In recent years, it has become a popular interconnect technology choice with currently being deployed in 28% of the systems listed in the TOP500. The vast majority of Infiniband implementations come as indirect, switch-based networks. The most common network topologies are both fat tree and dragonfly. Figure 3.9 displays an overview of the Infiniband system integration and processor interface. Similar to the Extoll NIC, the Infiniband NIC, also commonly known as the *Host Channel Adapter* (HCA), implements the descriptor-based approach to interface the processor. Typically, an HCA is connected through PCI Express. When the Infiniband HCA is initialized, a *queue pair* (QP) is allocated in the main memory with one queue serving send and another queue serving receive work requests. For instance, when the processor wants to transmit data to another node, it creates a descriptor for the work request and enqueues it

Table 3.4: Infiniband versus Ethernet performance comparison.

Interconnect	Latency [us]	Bandwidth [GB/s]
Ethernet 1G [68]	47	0.112
Ethernet 10G [68]	12	0.875
Infiniband QDR [68]	1.6	3.23
Infiniband EDR [69]	0.6	12.5

into the send queue. To signal the availability of a new work item to the HCA, the processor needs to write to a doorbell register, which in turn notifies the HCA. The HCA then reads the descriptor and processes it from the send queue through *direct memory access* (DMA). When the transaction is complete, the HCA generates a notification and writes it into the completion queue, which is frequently queried by the processor. The receiving data path has an analogous workflow. A descriptor is enqueued into the receive queue. Then, the processor triggers the transmission by writing to a doorbell register.

Table 3.4 provides a performance comparison of different Infiniband and Ethernet generations. As can be seen, Infiniband is superior to 1G and 10G Ethernet in terms of bandwidth and latency. Even though, both Ethernet generations are still deployed as their cost is relatively low and typical cluster deployments need thousands of NICs and hundreds of switches.

Applications designed for Infiniband clusters typically rely on Verbs [70] to describe RDMA functionality. Verbs is not an actual API, but a low level description for RDMA programming. It is close to the “bear-metal” and can be used as a building block for many applications such as sockets, storage and parallel computing. Verbs can be divided in two groups: control path and data path. The control path manages the resources and requires context switches, while the data path uses the resources to send/receive data. *libibverbs* [71] is an open-source Verbs implementation and has become the de-facto standard for *nix operating systems.

3.4.2 PCI Express

The *Peripheral Component Interconnect Express* (PCI Express or PCIe) [72, 73] standard describes a high performance, general purpose I/O interconnect standard, which is designed to connect peripheral devices and processors within a node. It was designed as a replacement for the older Peripheral Component Interconnect (PCI) bus standard and its extension PCI-X (PCI eXtended). The PCI Express standard maintains several PCI key attributes including the usage model, the load/store

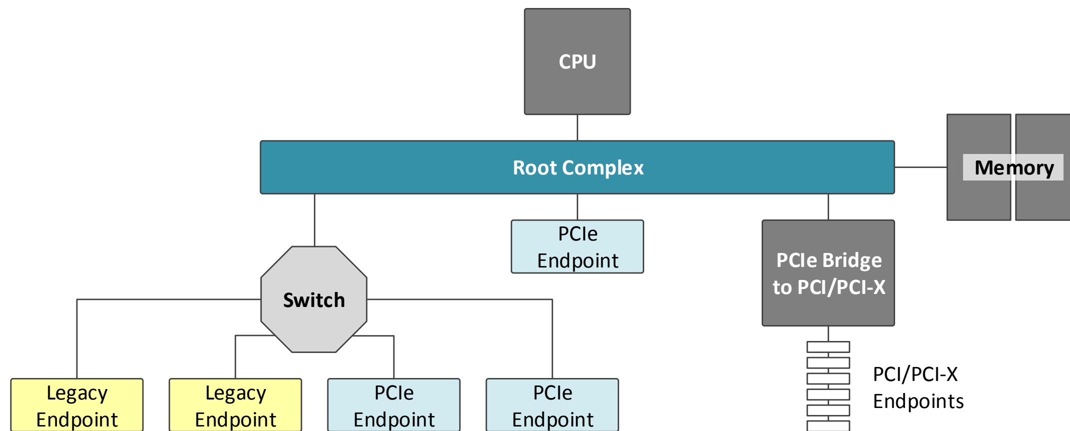


Figure 3.10: PCIe topology.

architecture, and the software interfaces, while the parallel bus implementation is replaced with a high-speed serial point-to-point interface. Nowadays, PCIe is the de-facto standard to connect add-in cards, such as graphics cards, to a computer, which makes it the fundamental technology for the NAA approach. Unlike older PCI standards which described an actual bus standard with several devices attached to it, PCIe devices are directly connected by point-to-point links. Depending on the bandwidth requirements, these links can consist of 1, 2, 4, 8, 12, 16 or 32 lanes, which in turn consist of one differential signal pair for each direction. An important feature of the PCIe design is the software backward compatibility with older standards. Operating systems and device drivers written for PCI devices are capable of operating PCIe devices without noticing any difference, as long as none of the new PCIe features are required. One aspect of this compatibility is that the PCIe device hierarchy is logically organized as a tree, just like PCI. Each PCI or PCIe device in a system is identified by a bus number, a device number and a function number. An example hierarchy is depicted in Figure 3.10. Typically, a PCIe hierarchy consists of several different components, including a root complex, switches, and endpoint devices.

Link A link is the physical connection between two devices. As described before, links can consist of varying number of lanes to suit different requirements.

Port A device's interface to the link is called port. It can either be an upstream port pointing towards the root complex or downstream port.

Switch Switches are the building blocks of the PCIe topology. They have one upstream port and several downstream ports. Internally, a switch consists of

one or more virtual PCI-to-PCI bridges, each connected to one of its ports. This way, despite offering only point-to-point connections to their neighboring devices, switches can build a tree structure and appear to the operating system as one or more PCI-to-PCI bridges to assure software-compatibility with PCI.

Root Complex The root complex is the root of the PCIe hierarchy and connects the hierarchy to the CPU and system memory. It is responsible for creating PCIe transactions when the CPU wants to access a device. It can contain several downstream ports. Similarly to a switch, the ports of a root complex are internally connected by virtual bridges on an internal bus.

Endpoint Devices Endpoint devices are leaves in the PCIe device tree. Examples are peripheral devices like graphics cards, network devices or storage devices.

Root Port A root port is a port of the root complex.

3.5 Performance Overview

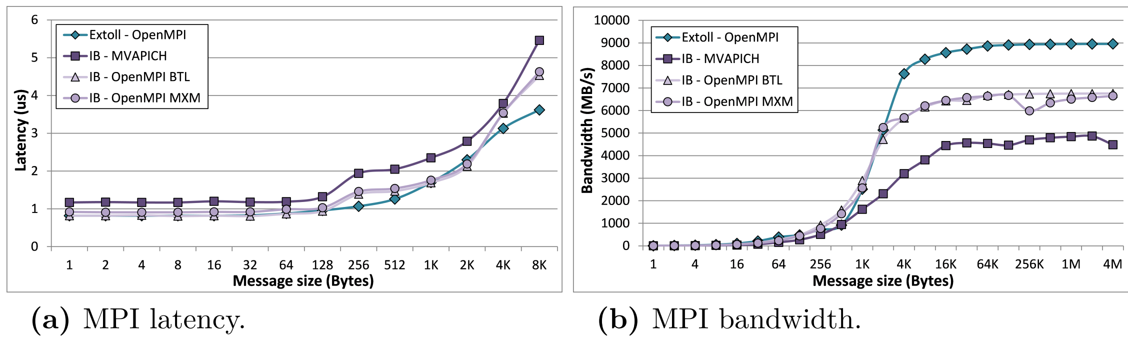
This section provides a summary of the performance comparison between the Extoll technology and Infiniband FDR cards [13, 14]. The latency and bandwidth results of the Extoll interconnect will serve as a reference point throughout this work.

3.5.1 Test Setup

The system comprises of two DL380 HP servers each equipped with an Intel Xeon CPU E5-2620 v3 (12 cores per socket) running at 2.4 GHz and 64 GB of main memory. Both systems host an HP ConnectIB FDR card and an Extoll Tourmalet 100G card.

The servers run CentOS 7.1 as their operating system. For the Infiniband cards, the Mellanox OpenFabris Enterprise Distribution version 3.0-1.01 provides the software environment while Extoll utilizes its own software stack release version 1.3.1.

The network cards are evaluated with the OSU Micro-Benchmarks (OMB) suite [74]. The benchmarks of interest are the point-to-point MPI benchmarks, which provide a measure for latency, multi-threaded latency, multi-pair latency, multiple bandwidth and message rate, bandwidth and bidirectional bandwidth. The ConnectIB FDR cards are evaluated with the MPI implementations MVAPICH 2-2.1 and OpenMPI 1.8.5, while Extoll is tested with its OpenMPI 1.6.1 MTL implementation.



(a) MPI latency.

(b) MPI bandwidth.

Figure 3.11: MPI performance overview.

3.5.2 Performance Results

Figure 3.11a displays the latency results obtained with the OMB *osu_latency* benchmark. It is notable that for small messages Extoll's latency is 0.81us while ConnectIB with OpenMPI MXM achieves 0.93us. But, Infiniband values are back-to-back. Realistically, a switch has to be included in the latency calculation which would result in 1.13us. For large messages, Extoll outperforms Infiniband in terms of latency. It is 191us faster than the fastest Infiniband MPI implementation.

Figure 3.11b presents the bandwidth results collected with the OMB *osu_bw* benchmark. The ConnectIB FDR cards reach about 6.7 GB/s with OpenMPI MXM and OpenMPI on Verbs (BTL), while MVAPICH only provides about 4.5 GB/s. Extoll is about 20% faster than ConnectIB FDR and reaches 8.9 GB/s bandwidth with its OpenMPI MTL implementation.

Chapter 4

Network-Attached Accelerators

The trend in supercomputing development shows that every decade the computing power of HPC systems increases by a factor of ten. With the upcoming Exascale challenge, the design of such systems poses multiple challenges, including power demands and consumption, the gap between compute capabilities and I/O bandwidth (also known as the *memory wall*), and higher hardware failure rates as the systems grow in scale. Another challenge is imposed by the extreme concurrency and efficient utilization of the heterogeneous computing resources. By adding accelerator devices such as many-core processors (coprocessors) or general purpose graphics processing units (GPGPUs) to a cluster, the overall energy and cost efficiency can be improved. They offer an unprecedented computational power per watt (Flop/s per Watt). Yet, the current generation of accelerator devices requires a host CPU to configure and operate them, which limits the number of accelerators per host and results in a static arrangement of hardware resources.

This chapter introduces the *Network-Attached Accelerator* (NAA) approach, which moves accelerators into a "stand-alone" cluster connected through the Extoll interconnect. The novel communication architecture enables the direct communication between accelerators through a high-speed network without any host interactions and an optimal application-to-compute-resources mapping, e.g., through the extension of the batch system capabilities [75]. The architectural idea is derived from the *Dynamical Exascale Entry Platform* (DEEP) project [76]. This chapter summarizes and extents various contributions to international conferences [9, 11, 10].

The remainder of the chapter is structured as follows. First, some background information about the DEEP project series and the PCI Express subsystem are

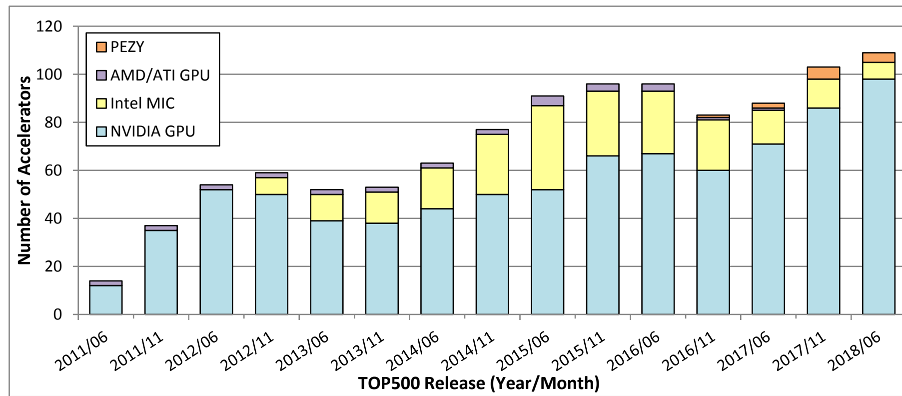


Figure 4.1: Accelerator development trend in the TOP500 since 2011.

presented followed by a summary of related work. Afterwards, the design objectives and strategy of the NAA software environments are presented and implemented for two different system setups: the DEEP booster and the VPCI project. For both implementations, an initial performance evaluation is presented.

4.1 Motivation

To satisfy the ever growing computational demands, many technologies and concepts have been developed and explored. The common scheme is to maximize the exploitable parallelism, which requires more computational power and high-speed internode communication in a large-scale system. Another important metric for efficient parallelization is scalability. A scalable system adapts well to increasing levels of parallelization without introducing significant bottlenecks. Considering this development cycle, it is questionable whether CPUs will be competitive in the future. Accelerator devices such as GPGPUs or coprocessors offer an unprecedented computational power per watt, while being optimized for massively parallel computation. The use of accelerator devices can significantly increase the system performance while providing energy efficiency. Therefore, it comes as no surprise that accelerators have been widely adopted in HPC systems and are subject to ongoing research efforts.

Many systems in the TOP500 [4] comprise of *heterogeneous architectures*, which combine multi-core CPU nodes with a small number of accelerator devices directly plugged into the node. A total of 110 systems in the current TOP500 list (June 2018) are using either coprocessor or GPU technology, including Summit, Tianhe-2A, Titan, and Piz Daint in the TOP10. Figure 4.1 displays the development trends in the TOP500 since 2011. However for accelerator devices to operate efficiently, sufficient utilization is important, which in turn requires fast network communication.

Typically, the network efficiency is measured in terms of latency and bandwidth. High throughput is required to provide an accelerator with sufficient data to work on, whereas latency is particularly important for synchronization.

Network communication is one of the main bottlenecks, which can introduce significant limitations on the efficiency of a computing cluster. When using accelerators, this is particularly important for peer-to-peer communication between local, but also remote, accelerator devices. Typically, remote accelerator communication must pass through the host's switch, interconnect, or memory and requires direct interaction with the host CPU. Also, accelerators are not designed to run autonomously and do not come with their own integrated network interconnect. Therefore, they are incapable of sourcing or sinking network traffic, which drastically limits the workload distribution and communication. Taking all of this into account, the scalability of host-centric accelerators is significantly limited. In addition, Amdahl's law [77] states that the scalability of a parallel code is limited by its sequential part. In reality, the problem size scales with the size of the system. E.g., if a scientific application is run on a system twice as capable, the system is not used to execute the application in half the time but to address a problem twice as big.

Future HPC systems have to be able to run applications with a varying degree of scalability and complicated communication patterns. The NAA approach proposes such a novel architectural idea, which is designed to provide a dynamic compute resource mapping at runtime satisfying the needs of large-scale applications.

4.2 DEEP Project Series

The DEEP project series [78] addresses the research of Exascale computing challenges following a stringent co-design approach. The series comprises of the three EC-funded projects *DEEP*, *DEEP – Extended Reach* (DEEP-ER), and *DEEP – Extreme Scale Technologies* (DEEP-EST). DEEP, the first project of the series, introduced a new heterogeneous computer architecture called the *Cluster-Booster concept* [79], [80], [81], as depicted in Figure 4.2. DEEP-ER [82] extended the Cluster-Booster architecture by implementing a multi-level memory hierarchy introducing *non-volatile memory devices* for efficient I/O buffering and checkpointing, and *network-attached memory* (NAM) [58]. DEEP-EST, the final edition of the DEEP project series, combines previous work by introducing the *Modular Supercomputing Architecture* [83].

Of particular interest for this work is the Cluster-Booster architecture. The basic idea is to split the architecture into a standard *cluster* and a *booster*. A cluster

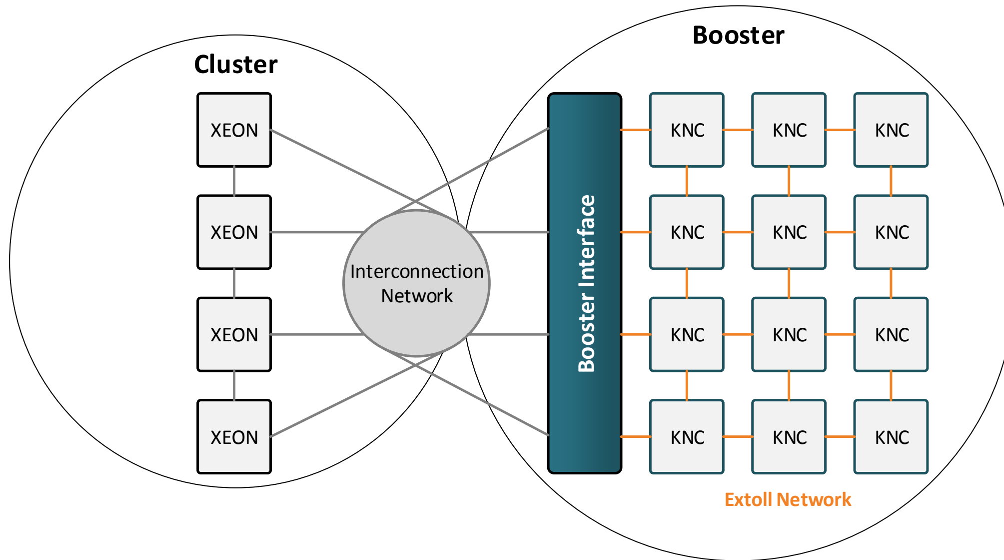


Figure 4.2: The DEEP Cluster-Booster concept.

consists of Compute Nodes (CNs), which are built from low-cost commodity of the shelf (COTS) components. The CNs are connected through a highly flexible switched network to a booster with an independent number of Booster Nodes (BN). A BN consists of an accelerator device and an Extoll NIC, and requires no direct host connection. The Booster Interface (BI) nodes are nodes that act as bridges between the two different interconnection network types. The *Cluster-Booster* concept is designed to overcome the limitations of current accelerator-based architectures. There are several advantages resulting from the innovative approach. BNs can directly communicate with other BNs and CNs. The distribution of workload is not fixed. The assignment between BNs and CNs can be decided during the runtime of an application in an N to M fashion. The CNs contain multi-core, out-of-order CPUs for executing scalar code while the booster needs less energy when running highly scalable code. There are different possible use cases for the proposed architecture. Depending on the code, the booster can run highly scalable applications without any host system interaction, but can also be assigned to specific CNs.

4.3 Introduction to the PCI Express Subsystem

The following sections provide a brief overview of the PCI Express address spaces, the Linux enumeration process, and two accelerator device architectures. The knowledge is needed to understand the requirements and design strategy of the Network-Attached Accelerator approach.

4.3.1 PCI Express Address Spaces

The PCI Express standard describes three address spaces: *I/O* space, *memory-mapped I/O* (MMIO) space and *configuration* space. All three address spaces are accessible by the CPU. The I/O and MMIO address spaces are used by the device drivers and can also be accessed by other devices while the configuration space is used by the PCI initialization code within the Linux kernel. The I/O space serves the same purpose as the MMIO space, but is only of interest when using legacy devices. In addition, PCIe introduces message transactions, which are intended to provide in-band support for features like legacy interrupts and error handling, making the need of extra lines unnecessary and reducing the number of required pins compared to previous PCI bus standards. The I/O space and messages are only required for legacy devices.

4.3.1.1 Memory-Mapped I/O Space

Memory-mapped I/O uses the same address space to address both memory and I/O devices. The memory and registers of I/O devices are associated with address values. When an address is accessed by the CPU, it may refer to a portion of physical RAM, or it can instead refer to the memory of an I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices. When the CPU wants to access one of these addresses, the root complex generates a transaction request and sends it to the device. The content and organization of MMIO space is device-specific. PCIe only provides the specification of the available MMIO space types and how a device requests memory from the system. For this purpose, each PCIe device has a set of *Base Address Registers* (BAR). These registers are located in the configuration space.

4.3.1.2 Configuration Space

The configuration space of a PCI Express device consists of a set of registers. These registers are used by configuration software to retrieve the status information, but also to configure its operation. To generate configuration transactions, the PCIe specification describes two mechanisms that allow configuration software running on the CPU to stimulate the root complex:

- the *PCI compatible Configuration Access Mechanism* (CAM), and
- the *PCI Express Enhanced Configuration Access Mechanism* (ECAM).

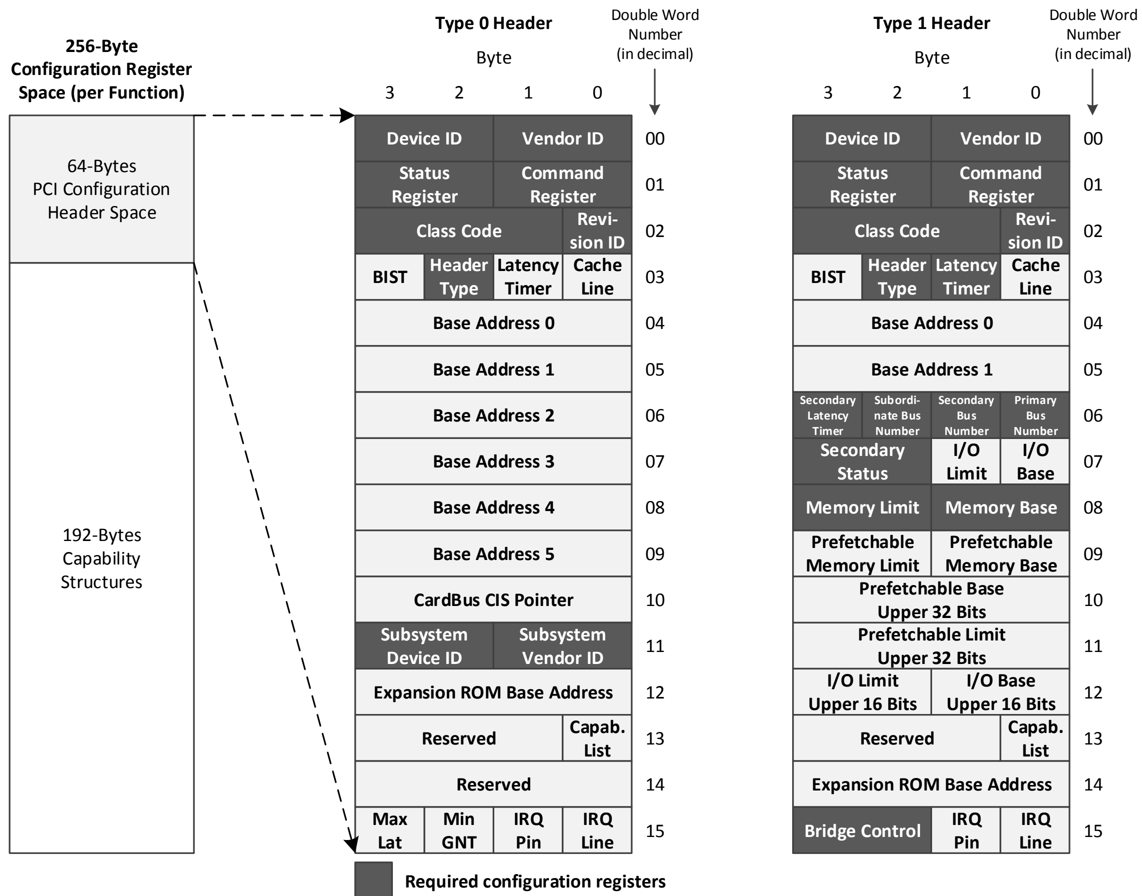


Figure 4.3: PCI Express configuration space header types [73].

Historically, the PCI configuration space was limited to 256 bytes, but has been extended to 4 KB for PCIe. Therefore, the first 256 bytes are referred to as *PCI-compatible space*, the rest builds the *PCI Express extended configuration space*. The first 64 bytes of the PCI-compatible space contain the PCI configuration header, which has a standardized set of registers, whereas the rest contains device-specific information and the capabilities list.

Header Types PCIe devices can be distinguished in bridge devices and endpoint devices. To serve their different purposes, they have different configuration space headers: header type 0 for endpoint devices and header type 1 for bridges. Figure 4.3 provides on overview of the header types.

Base Address Registers BARs contain the start addresses of the device's MMIO windows. Every endpoint device contains six 32-bit wide BARs, which can either be MMIO or I/O regions. Alternatively, two registers can be combined to address

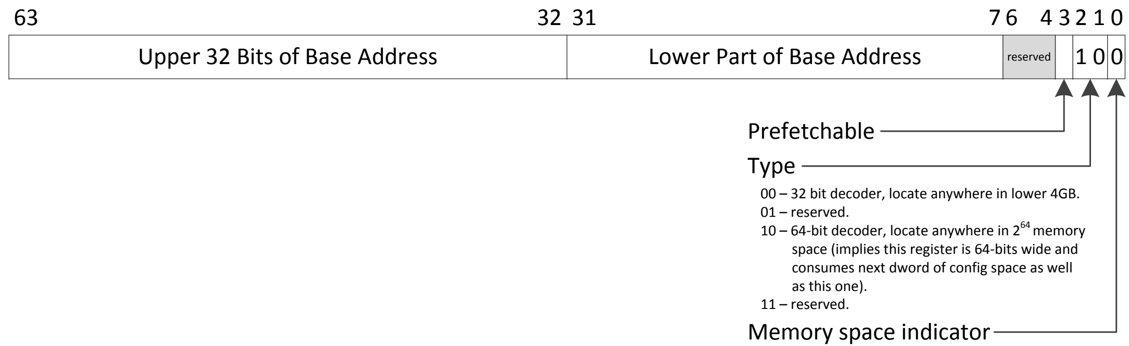


Figure 4.4: Base address register.

a 64-bit memory region. The least significant bits indicate the type of a BAR, as shown in Figure 4.4. If a BAR is 64-bit wide, its upper bits form the least significant 32 bits of the 64 bit address, whereas the next BAR contains the most significant 32 bits. If a BAR is prefetchable, a bridge is allowed to prefetch memory for improving read request performance. For PCIe endpoint devices, only 64-bit memory BARs can be configured as prefetchable. The address stored in a BAR register is assigned by the system configuration software. First, the software needs to query the size. This is done by writing `1`s into the register, and then, reading back the result. The lowest writable bit indicates the size and alignment of the address range.

Expansion ROM Base Address Register The PCIe specification defines a mechanism where devices can provide expansion ROM code that can be executed for device-specific initialization and, possibly, a system boot function before any operating system or device drivers are loaded. The *Expansion ROM* BAR stores the address of such an expansion ROM. Similar to the BAR, device independent software can determine the size of the address space by writing a value of all 1's to the address portion of the register, and then, reading the value back.

MSI Capability Register Set PCIe devices are required to support the *Message Signaled Interrupts* (MSI) mechanism. MSI is an in-band method for signaling an interrupt and is designed to replace traditional out-of-band assertions of dedicated interrupt lines. They eliminate the need to implement extra pins for signaling interrupts. To trigger an interrupt, a device simply sends a memory write request to the root complex, which in turn signals an interrupt to the CPU. On the transaction layer, an MSI cannot be distinguished from a memory write request, but its content can be differentiated from other transactions. The differentiation can be made based on the address the MSI is written to on the root complex. To enable a device's MSI

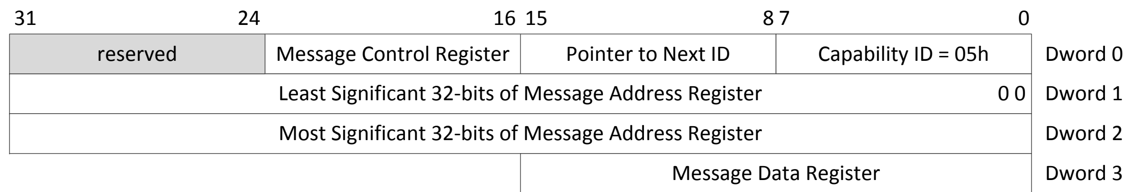


Figure 4.5: MSI Capability register set for 64-bit address size.

capability, the *target address*, the *payload* written to the target address, and the *number of messages* needs to be configured . The configuration is stored in the MSI capability register set, which is located in the PCIe configuration space. Figure 4.5 shows such a register set. There are two formats defined by the PCIe specification, one for 64-bit and one for 32-bit addresses. The description focuses on the 64-bit format.

The *Message Address Register* comprises of two 32-bit values, which specifies the DWORD-aligned address for the MSI memory write transaction. The *Message Data Register* configures the payload of the MSI memory write. The *Message Control Register* indicates the function's capabilities and provides system software control over MSI. If the MSI enable bit in the message control register is set to 1 by the system configuration software, the device is permitted to use MSI.

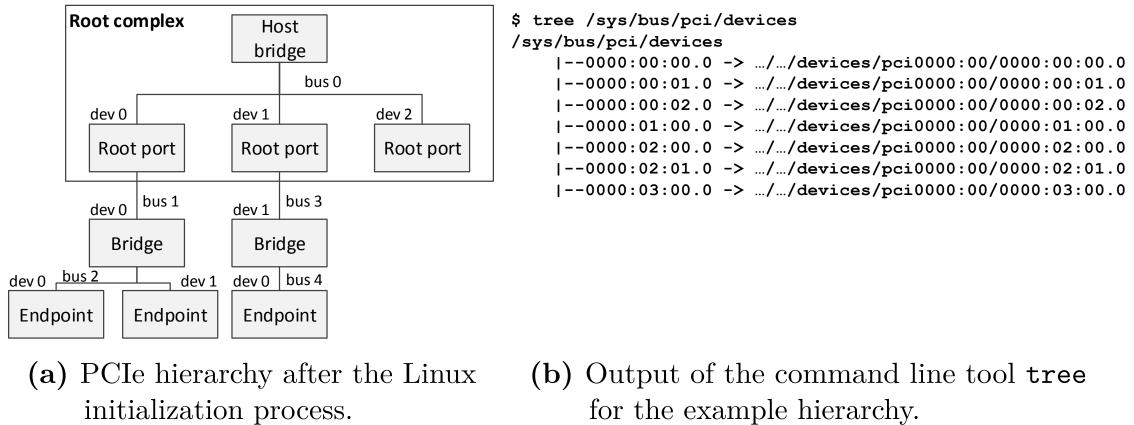
4.3.2 Linux PCI Express Enumeration

At boot time, the topology of the PCIe device hierarchy is unknown to the operating system. In order to find and configure all connected PCIe devices, the configuration code, typically the *Basic Input/Output System* (BIOS) or its successor the *Unified Extensible Firmware Interface* (UEFI), performs a depth-first search until all devices are discovered. This process is called *enumeration*. The following steps outline the generic steps performed by the enumeration process for a single root complex:

(1) The only bus known at boot time is the primary PCIe bus 0, which is the internal bus inside of the root complex. Each newly discovered bus is consecutively numbered by the configuration code.

(2) If the current bus has any devices attached to it, they are initialized and numbered incrementally starting with 0.

(3) For any device, it is required that the first function number is set to 0. If it is a multi-function device, the remaining functions can have any number between 1 and 7, and are not required to be numbered consecutively.



(a) PCIe hierarchy after the Linux initialization process.

(b) Output of the command line tool `tree` for the example hierarchy.

Figure 4.6: Linux PCI Express enumeration example.

(4) When a bridge device is discovered, the search continues recursively on the bus connected to its downstream port and proceeds with step 2.

Figure 4.6a displays an example hierarchy configured by the described enumeration process, while Figure 4.6b presents the output of the command line tool *tree* for the example hierarchy. A detailed enumeration process example can be found in the book *PCI Express System Architecture* [73]. As the Linux kernel initializes the PCIe subsystem, it allocates data structures mirroring the real topology of the system. Each PCIe device (also including the bridges) is described by a data structure of type `struct pci_dev` and each bus is described by a data structure of type `struct pci_bus`. The result is a tree structure of buses each of which have a number of child PCIe devices attached to it. As a bus can only be reached using a bridge (except the primary PCIe bus, bus 0), each PCI bus structure contains a pointer to the parent device that it is accessed through. That PCIe device is a child of the PCIe bus's parent bus. With kernel version 2.6, the Linux kernel developers have introduced a unified new device model to cope with increasing requirements and capabilities of modern hardware, which is commonly referred to as the *Linux Device Model* [84, chapter 14]. The Linux kernel utilizes the tree-like data structure describing the PCI topology and exposes it to the user through the Linux device model, e.g., through the sysfs virtual file system.

As described by Rusling [85], the Linux PCI (Express) initialization code is broken into three logical parts: the *PCI device driver*, the *PCI BIOS*, and the *PCI fixup*.

PCI Device Driver The *PCI device driver*, not to be confused with a real device driver, is a function of the operating system called at system initialization time.

The PCI initialization code performs the actual enumeration process and locates all devices and bridges in the system. It utilizes the PCI BIOS code to determine whether a slot of the currently scanned bus is occupied. If a PCI slot is occupied, it allocates the corresponding data structure and links it into a double-linked list, which describes the topology of the PCI Express system.

PCI BIOS The *PCI BIOS* provides a software layer for the services described in the PCI BIOS specification. The functions are a series of standard routines which are common across all platforms. They allow the CPU controlled access to all of the PCIe address spaces. Only Linux kernel code and device drivers may use them.

PCI Fixup The system-specific PCI fix-up code tidies up loose ends of the initialization. For Intel-based systems, the system BIOS, which ran at boot time, has already fully configured the system. This leaves Linux with little to do other than map that configuration. For non-Intel based systems, further configuration is needed.

4.3.3 PCI Express Expansion Cards

The PCI Express standard can be employed on different interconnect levels in the system, e.g., as a motherboard-level interconnect, a passive backplane interconnect, or as an expansion card interface for add-in boards. The primary purpose of an expansion card is to provide or expand the features of a computing system. Accelerator devices are typically implemented as expansion cards. In the following, two popular accelerator device technologies are introduced, which rely on the PCI Express standard.

4.3.3.1 Intel Many Integrated Cores Architecture

The Intel *Many Integrated Cores* (MIC) architecture [86] is a coprocessor technology developed by Intel and commonly known as Intel Xeon Phi. Besides GPUs, it is one of the most popular accelerator choices in today's HPC systems. Figure 4.7 presents a simplified architecture diagram of an Intel Xeon Phi coprocessor of the Knights Corner generation. The Intel Xeon Phi provides up to 61 multi-threaded cores, each with its own private L2 cache. The global distributed tag directory *TD* is used to facilitate cache coherency. The GDDR5 device memory can be accessed through four memory controllers. The host interface is PCI Express. All components are connected through a ring interconnect.

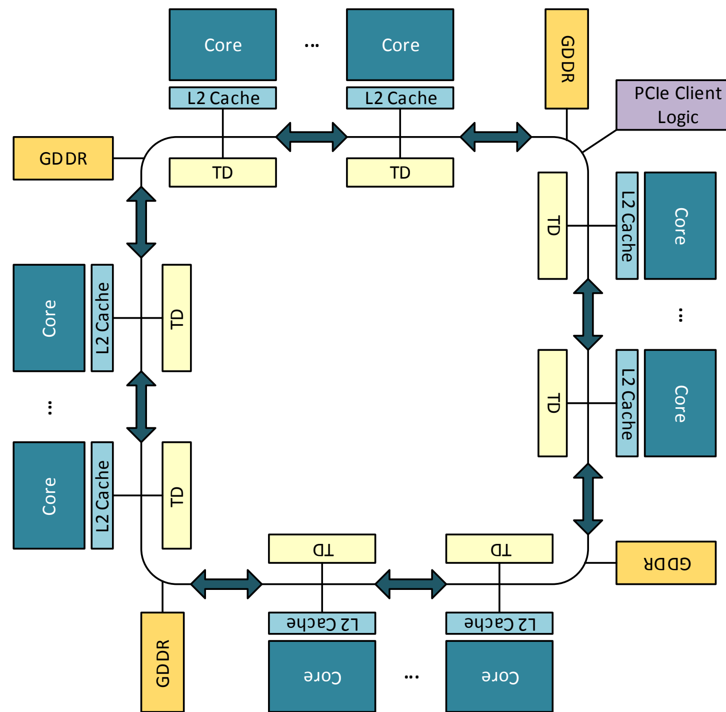


Figure 4.7: Simplified architectural overview of an Intel Xeon Phi Coprocessor.

The Intel MIC architecture is based on x86 and provides its own embedded Linux operating system, which allows the compilation of kernel modules. The Intel *Manycore Platform Software Stack* (MPSS) is used on a host system to control and operate connected MICs. It consists of an embedded Linux, a minimally modified GCC and driver software for the Intel MICs. The Intel MIC architecture can be used with different programming models for the MPI mode [87].

4.3.3.2 Graphics Processing Units

The NVIDIA GK110 Kepler GPU architecture [88] is a modern *graphics processing unit* (GPU) architecture representative. The compute cores of a GPU are grouped in *Streaming Multiprocessor Units* (SMXs). Depending on the implementation, a graphics card relying on the Kepler architecture can comprise of up to 15 SMXs.

One SMX comes with 192 CUDA-cores with each of those cores running at a maximum frequency of 0.71 GHz. They are complemented by double precision units – special cores that can handle 64 bit operations –, load/store units (LD/ST), and 32 special function units. The latter provide fast approximations for complex mathematical operations, such as sinus, and trade speed for precision – they need to be used explicitly by the programmer. Each SMX has 64 KB of on-chip memory with 16 KB of L1 cache. SMXs share the same L2 cache with a size of 1536 KiB. Six

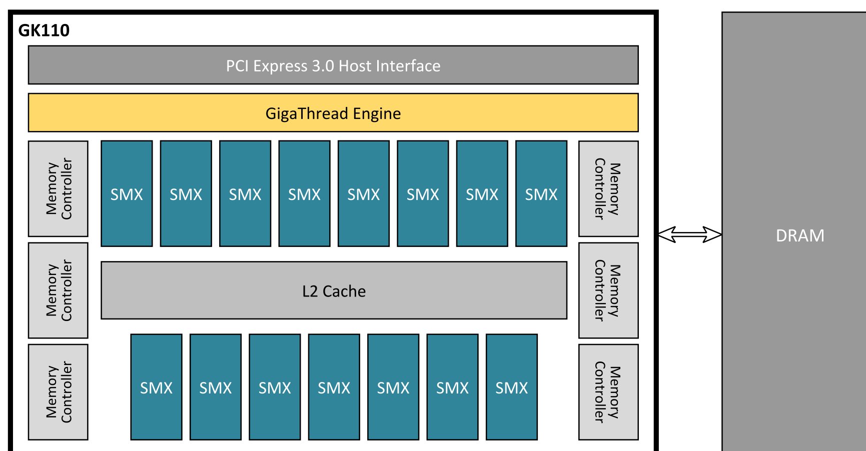


Figure 4.8: Nvidia's Kepler GK110 architecture + DRAM.

64 bit memory controllers manage the communication between the L2 cache and the on-board DRAM of the graphics card. The *GigaThread Engine* is NVIDIA's global scheduling unit that distributes work to the SMXs. A PCI Express 3.0 port enables communication with other devices.

4.4 Related Work

The improvement and optimization of direct communication between accelerator devices is one of the main research topics of current research. The following sections provide a selection of Intel Xeon Phi and GPU-related research efforts in the communication domain followed by an overview of hardware-related research.

4.4.1 Intel Xeon Phi Coprocessor-based Communication Models

DCFA The *direct communication facility for many-core based accelerators* (DCFA) [89] targets the implementation of direct data transfers for many-core architectures and utilizes the InfiniBand technology. The internal structures of an Infiniband HCA are mapped to memory areas of the host and the MIC. The MIC reads or writes data by directly accessing the memory areas of a remote host or remote MIC. MPI communication primitives executed on the MIC transfer data by issuing commands to the HCA. The implementation is based on the Mellanox InfiniBand (IB) driver. DCFA-MPI [90] provides an MPI implementation based on the DCFA framework for direct peer-to-peer communication between remote Intel MICs.

MVAPICH2-MIC MVAPICH2-MIC [91, 92] is a proxy-based communication framework using InfiniBand and the symmetric communication interface (SCIF) [87]. Its goal is to optimize the collection of communication paths possible in the symmetric mode. The framework provides three different communication designs. *DirectIB* provides direct MPI communication through the Coprocessor Communication Link (CCL) driver of the MPSS software stack. The *passive proxy* handles host-staged communication by setting up staging buffers, but is not directly involved in communication between a MIC and a remote MIC. It utilizes the RDMA capability of SCIF. The *active proxy* utilizes a dedicated processor core on the host. The *one-hop* variant initiates and progresses communication that is staged through the host. SCIF transfers are initiated on the host. The *two-hop* variant tries to utilize the high bandwidth channels of the MIC. A MIC-to-remote-MIC transfer is staged to the local host then to the remote host and finally to the remote MIC.

HAM and HAM-Offload HAM [93] is implemented as a C++ template library to create type-safe heterogeneous active messages (HAM). Active messages can contain or reference to code that should be run upon receipt. The main question is how to translate between handler addresses of heterogeneous binaries with minimal cost. HAM solves the problem by adding a level of indirection that is implemented in pure C++ without any language extensions. The message handler registry acts like a map between keys and handler addresses. Each process has the same set of keys, but the handler addresses differ between the individually compiled binaries for different instruction set architectures. The HAM-offload API [94] completes the functionality of HAM with a unified intra- and internode offload API. It provides similar primitives as other offload programming models. The framework is compiler independent and provides communication backends for MPI and SCIF.

4.4.2 GPU Virtualization and Communication Techniques

rCUDA rCUDA [95] or remote CUDA is a framework for remote GPU virtualization in cluster environments. It is fully compatible with the CUDA runtime and transparently allocates one or more (local or remote) CUDA-enabled GPUs to a single application. rCUDA implements a client-server architecture. On the client side, the rCUDA wrapper library intercepts calls to the CUDA runtime and forwards them to the server side. The rCUDA server daemon, running on each node, offers acceleration services, receives the forwarded requests and runs the CUDA kernel. The software overhead reduces the performance in comparison to pass-through technologies.

VirtualCL VirtualCL (VCL) [96] is very similar to rCUDA in its basic concepts. It provides transparent access to accelerator devices on remote nodes and allows applications to utilize accelerators on different nodes without requiring the application to explicitly split its computations between these different nodes. The application itself only runs on a single node and VCL executes OpenCL kernels on other nodes when necessary. VCL consists of three components: the VCL library, the broker and a back-end daemon. The VCL library implements OpenCL and transparently accesses OpenCL devices on the cluster. The broker is a daemon running on each host system. It is responsible for monitoring the availability of OpenCL devices and allocates available devices for a client. It also manages communication between applications and back-ends. The back-end daemon runs on every node that contains usable accelerator devices. It uses vendor-specific OpenCL libraries to run OpenCL kernels on its devices, when requested by a client.

NVIDIA Grid Front-End virtualization, as it is implemented by frameworks like rCUDA or VCL, offers great flexibility and reduces the required hardware while increasing the utilization of available devices, but comes with two major disadvantages. First, there is an overhead due to the software involved on client and server side. Second, these frameworks are completely tied to a specific API, like CUDA or OpenCL. To avoid these problems, the NVIDIA Grid technology [97] offers back-end virtualization for virtual machines. The NVIDIA Kepler GPU design implements a memory management unit (MMU) and dedicated input buffers for each virtual machine. This way, different virtual machines can simultaneously use a single GPU without interfering with each other and without the software overhead of API interception. NVIDIA Grid offers high performance without penalizing multiplexing. However, special hardware support is required, restricting this technique to a limited number of devices, which can be configured to support such a functionality.

GPUDirect RDMA GPUDirect RDMA [98] is a feature of the CUDA runtime environment and was first introduced in Kepler-class GPUs with CUDA 5.0. This technique provides a direct peer-to-peer data path between two GPUs by mapping the GPU memory to one of the GPU's BARs. Other peripheral devices can use this physical address to communicate directly with the GPU. Remote RDMA communication is improved by removing unnecessary memory copies between GPU memory and host memory. But, the ratio between CPUs and GPUs is still fixed.

Global GPU Address Space The Global GPU Address Spaces (GGAS) [99] concept facilitates direct communication between distributed GPUs while bypassing the CPUs for all communication and computational tasks. GGAS relies on thread-collective communication, and therefore, maintains the GPUs bulk-synchronous, massively parallel programming model. Furthermore, GGAS utilizes a zero-copy technique for data movement between distributed GPU memories. This technique relies on overlapping shared GPU memory segments with SMFU address space of the Extoll NIC building a distributed shared, and therefore global, GPU address space. The major limitation of this approach is the requirement that a GPU device needs to support GPUDirect RDMA in order to span the global address space.

4.4.3 Hardware-related Research

The discussion about a cluster of accelerators was first introduced with the QPACE supercomputer [100] prototype. QPACE was a massively parallel quantum chromodynamics prototype enhanced by Cell BE processors. In 2010, QPACE was ranked #1 on the Green500 list [101]. Other approaches tend to use PCIe (Peripheral Component Interconnect Express) as the interconnection network between accelerators. Non-Transparent Bridges (NTB) [102], [103] connect independent PCIe hierarchies of different nodes. Communication between accelerators relies on address translation and table based addressing schemes inside the NTB. This introduces additional management overhead and does not scale very well. In addition, PCIe lacks of some interconnection network features. Advanced Switching Interconnect (ASI) [104] tries to extend the PCIe protocol to support features such as protocol tunneling, routing, and congestion management. The goal of independent scalability of hosts and accelerators is not achieved with these approaches. Recently, the NVIDIA NVSwitch technology [105] together with the NVIDIA DGX-2 system [106] have been introduced. NVSwitch is an on-node switch with 18 NVLink ports per switch. Internally, it is a fully connected crossbar. The NVIDIA DGX-2 utilizes the NVSwitch technology to connect 16 NVIDIA Tesla V100 GPUs to one host system.

4.5 NAA Software Design

The NAA software design is tightly coupled with the architecture of the underlying hardware. The following sections describe the general hardware architecture of NAA and software requirements followed by an introduction to design and implementation strategy of the NAA software environment.

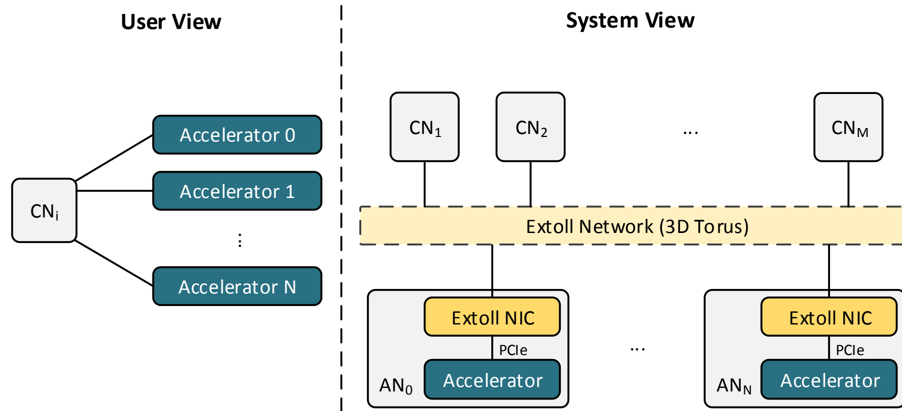


Figure 4.9: NAA architecture diagram: system versus user view.

4.5.1 System Architecture and Problem Statement

The network-attached accelerators approach proposes a new architectural idea for scaling the number of accelerators and host CPUs independently. The idea is to decouple accelerator devices from their host systems and place them on so called *accelerator nodes* (ANs) anywhere in the network. Figure 4.9 presents the architectural and user view of an NAA system. An accelerator node consists of a passive PCIe backplane equipped with an accelerator device and an Extoll NIC, which enables an accelerator to directly connect to a network. The main purpose of the backplane is simply to connect the two cards so that the PCIe traffic can be forwarded, since a PCIe add-in card comes without a PCIe slot. The Extoll card in the cluster node (CN) is configured as an endpoint device, while the Extoll NIC on the accelerator node acts as a root port. This modular way of integrating heterogeneous system components enables applications to freely choose at run time the kind of computing resources on which they can run in the most efficient way in an N:M ratio.

In order to provide applications with a transparent access to the computing resources, a virtualizing software layer is needed, which is able to configure and operate the remote accelerator devices, but also to make them visible in a cluster node's PCIe hierarchy. Figure 4.10 presents the schematic view of the PCIe tree visible to the Linux operating system after a network-attached accelerator has been mapped over Extoll as a virtual device in the local PCI Express hierarchy. In order to facilitate a good understanding about the system setup and software design requirements, several observations about Figure 4.10 have to be understood:

(1) The physical accelerator device is not directly connected to a cluster node. It is located on a network-attached accelerator node and connected to the host system via Extoll. This means that the local root complex and other local

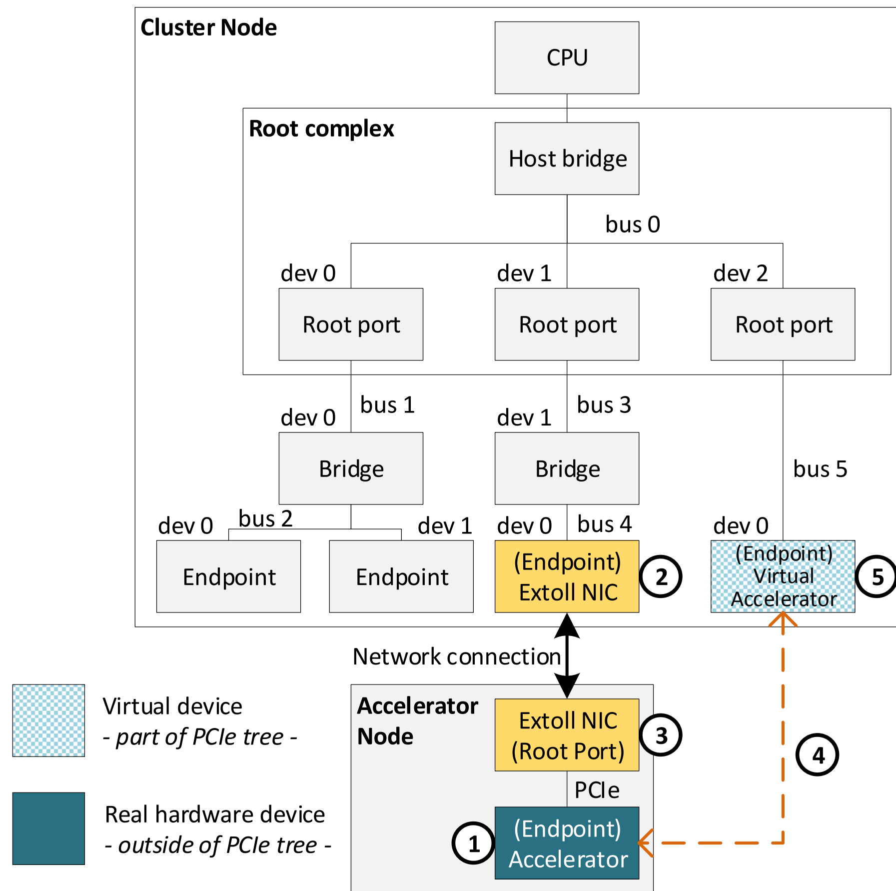


Figure 4.10: PCI Express tree as seen by the Linux operating system with one network-attached accelerator device.

PCIe devices cannot send any transaction requests to the remote device. In fact, the physical device is not part of the system's PCIe hierarchy.

(2) In terms of PCI Express, the Extoll NIC connected to the host system is an ordinary endpoint device, which only has an upstream port but no downstream ports. Therefore, the local NIC cannot act as a PCI bridge device, which could simply forward transactions to the remote accelerator.

(3) While the local NIC cannot be configured to serve as a bridge device, the remote NIC can be configured to act as a root port with the accelerator device being connected to its downstream port. But, it has no upstream port, and therefore, cannot receive any PCIe transactions from a cluster node. Also, only a root complex can initiate configuration requests, but the remote NIC, despite being configured to act as a root port, is not part of a root complex.

(4) In order to access the accelerator device, the network-attached accelerator approach must map both the virtual and physical device in a way such that

all transactions that involve the remote accelerator – either as a requester or completer – are forwarded between the two nodes. In Figure 4.10, the dashed line between both devices illustrates this mapping.

(5) In this example, the logical device is placed behind a root port, just like an actual device would by physically connected to that port. This means that even though no physical device is attached to the root port, the logical device appears to the CPU as if it is locally connected.

4.5.2 Objectives and Strategy

The main objective of the NAA software architecture is to provide a transparent mapping between the network-attached accelerator and cluster nodes by emulating a PCIe device. The implementation of the software stack needs to be transparent to upper software layers, including the accelerator device driver and runtime libraries, while maintaining the commodity aspect of the accelerators. Recapitulating the findings from section 4.3 and subsection 4.5.1, the NAA software environment needs to fulfill the following tasks:

Configuration Request Forwarding PCIe configuration read and write requests need to be forwarded to and from the remote accelerator. The NAA software stack needs to be able to recognize and forward such requests accordingly.

Device Enumeration The software environment should be able to emulate the PCIe device enumeration. After powering an accelerator device, it needs to be configured and mapped onto system resources to be ready for operation.

Memory-Mapped I/O The cluster nodes should be able to map and forward access requests the accelerators' BAR windows, which are typically made visible to the system as MMIO regions in the main memory. This is needed for the configuration and communication with the accelerator devices, but also to provide the compatibility with PCI-specific system calls and functions.

MSI Configuration and Interrupt Delivery During the accelerator device configuration, the software has to modify the MSI packet (destination address and payload) by writing the corresponding values to the MSI capability register set. The layer has to be able to register the accelerator's interrupt handler with the Extoll interrupt management. The software needs to distinguish between Extoll-based interrupts and interrupts issued by the accelerator.

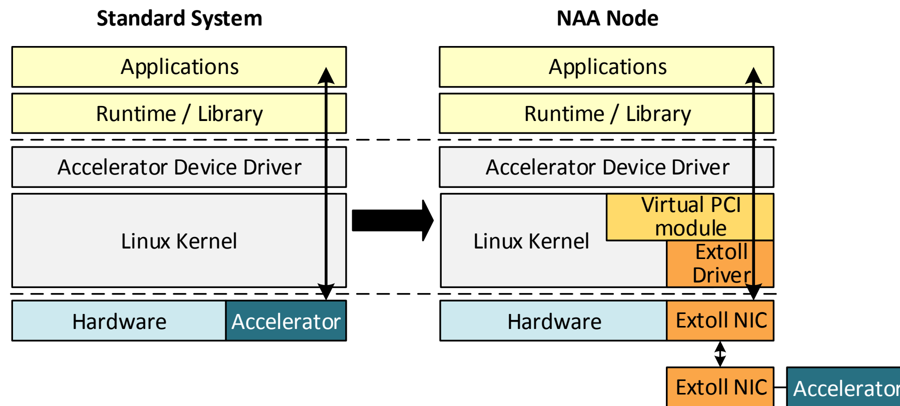


Figure 4.11: Abstract software stack view.

Figure 4.11 shows a generic overview of the NAA software environment in comparison to the default Linux software stack. On the left-hand side, the typical layers of a software stack supporting an accelerator device are displayed. The goal of the NAA software environment is to remove the locally installed accelerator device and integrate it into the Extoll network, and if possible, without any modification of the original application code. In principle, achieving this goal is possible by manipulating any of the software layers on the left side. But, it is desirable to introduce as little modifications as possible.

The general idea of the NAA software approach is presented on the right-hand side of Figure 4.11. The idea is to redirect PCI support library calls to a “virtual” PCI layer, which is able to distinguish between local PCI requests and communication targeting the accelerator nodes, but also provides the means to fulfill the previously described tasks. The following sections explain the concepts needed to implement these tasks and illustrate how they can be mapped onto the Extoll technology.

4.5.2.1 PCIe Configuration Space Access

In the absence of a CPU and a root complex on the accelerator node, it is the task of a remote host system to configure the accelerator card’s PCIe host interface over the network. The Extoll NIC’s interface is designed to send and receive only PCIe memory request packets as an endpoint. To configure a PCIe hierarchy, the host interface must be able to send and receive PCIe configuration request packets.

For this purpose, the Extoll NIC features two functional units that enable the configuration and operation of remote PCIe buses: the RMA unit and the PCIe Bridge unit. In addition, the remote Extoll NIC needs to be configured to act as a root port. First, a specialized functional unit is needed to inject PCIe configuration

packets into the Extoll NIC's outgoing host interface traffic. The PCIe bridge unit, introduced in section 3.2.7, resides in the on-chip network of Extoll's network interface and can be configured by writing to the corresponding registers in the register file. The unit is accessible from every device over the network with *Remote Register file Access* (RRA) transactions, which are basically immediate PUT and GET operations to the remote registers. With this technique, a host can configure the remote PCIe bridge unit to forward incoming PCIe configuration packets to the accelerator device by writing to the PCIe backdoor register presented in Table 3.2. By writing PCIe configuration packets into the `htoc_to_pcie_backdoor_data` register in the remote register file via RRA transactions, they are inserted into the outgoing PCIe traffic stream from the root port to the accelerator.

4.5.2.2 Device Enumeration

A simplified enumeration process can be used, since only one device resides on the root port of a remote accelerator node. The important values that need to be configured are the *bus, device and function* number, BARs, and the MSI capability registers. The *bus, device and function* number is used to identify the accelerator inside the PCIe hierarchy, whereas the MSI capability registers define the target address and the data that is sent when an interrupt is issued from the accelerator. The BARs define a memory window which is required to enable the internal address translation for incoming request packets targeting the accelerator. This defines the way a host can access and communicate with the accelerator.

Therefore, with the remote device's configuration space being accessible transparently, all that needs to be done is to rescan the bus, on which it is to be placed.

4.5.2.3 Memory-Mapped I/O Regions

In general, peripheral components are accessed by using load and store operations to reserved address ranges in the PCIe configuration space and the memory-mapped I/O regions. The operations are mapped to an add-in card and translated into read and write requests. The easiest way to give a host access to the accelerator is to use these loads and stores to an MMIO region assigned to the accelerator. This has the additional benefit that the upper software and hardware layers can remain unchanged, but leads to the question of how to map physical addresses of the host memory to the accelerator's PCIe address space.

The Extoll NIC's SMFU can export segments of local memory to remote nodes to build a distributed shared memory system. Loads and stores from the CPU to

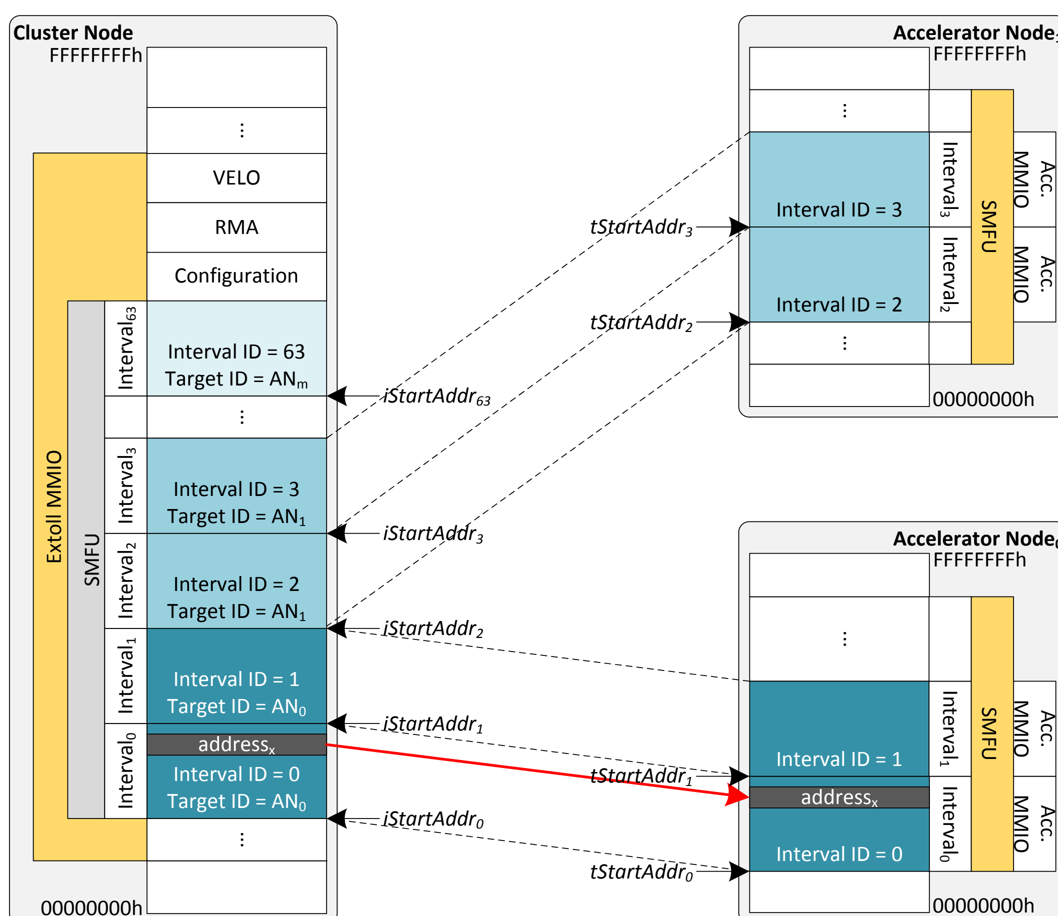


Figure 4.12: Memory mapping between a cluster node and two accelerators.

these exported memory regions are encapsulated into network transactions to the remote node. At the remote node, the packets are translated back into host interface requests. The PCIe packet ordering is ensured along the path from the CPU to the target accelerator and vice versa by keeping the order of the packets received at the SMFU and forwarding them to the host or the network interface with the same order in a FIFO-like manner. With this memory mapping technique and ensured packet ordering, the MMIO ranges appear to be locally mapped to the local host's main memory, but in reality this address range can be located anywhere in the network. The NAA software environment configures this memory mapping for every accelerator node in the system by writing the SMFU configuration to the node's register file through remote register file accesses.

Figure 4.12 illustrates the memory mapping between a cluster node address space and two remote accelerator PCIe address spaces. The MMIO region assigned to the NIC is divided into several intervals. Each of these intervals is exclusively assigned to a functional unit. The range assigned to the SMFU is further subdivided into different

intervals. Each of these intervals corresponds to a region of exported memory (BAR) with the size and location defined by a start ($iStartAddr_n$) and an end address. In addition, each interval has an *interval ID* and a *Target ID*. The *Target ID* specifies the Extoll node ID of the accelerator node the loads and stores are forwarded to, while the interval ID is used to match the source interval ID of these loads and stores. In this example, two BAR windows are mapped per accelerator node. $address_x$ hits SMFU interval 0, which is translated to an address on AN_0 in SMFU interval 0.

The SMFU on the accelerator node side adds an offset to incoming network-encapsulated request packets. The offset defines where the exported region is located in the accelerator's memory space. If the calculated address matches the BAR assigned to the accelerator, a load or store to that address is sent to the accelerator. Note, the offset to an address in the SMFU interval has the same offset to an address in the accelerator's BAR region. This technique allows the host CPU to directly access any accelerator connected over the Extoll network. Once the Extoll network is configured, each accelerator node has a unique *node ID* in the Extoll fabric. This *node ID* is used to address other accelerators within the system. As a consequence, two accelerator nodes belonging to different hosts are able to communicate with each other independently from the host systems.

4.5.2.4 MSI Configuration and Interrupt Delivery

Most PCIe devices require interrupts to fulfill their function and to communicate events between the device and the driver. PCIe devices implement the Message Signaled Interrupt (MSI) mechanism, which sends a posted write packet towards an *Advanced Programmable Interrupt Controller* (APIC). A write to this controller triggers an interrupt and the operating system forwards the interrupt to the corresponding device interrupt handler. In the network-attached accelerator architecture, there is no APIC register on the accelerator node and no direct connection between the accelerator's PCIe bus and the host systems APICs. To provide interrupt functionality, some special adjustments have to be made.

The interrupt management for accelerator devices is implemented by extending the Extoll interrupt handling on the host system to handle accelerator interrupts as well. The address of an MSI packet is stored in a PCIe configuration space register, which means that it can be modified from the remote host through configuration packets. The host sends configuration packets to the remote PCIe bus, which modifies the MSI capability registers on the accelerator card. The accelerator's MSI address is manipulated in a way that the address is forwarded to the host, and there, it hits a

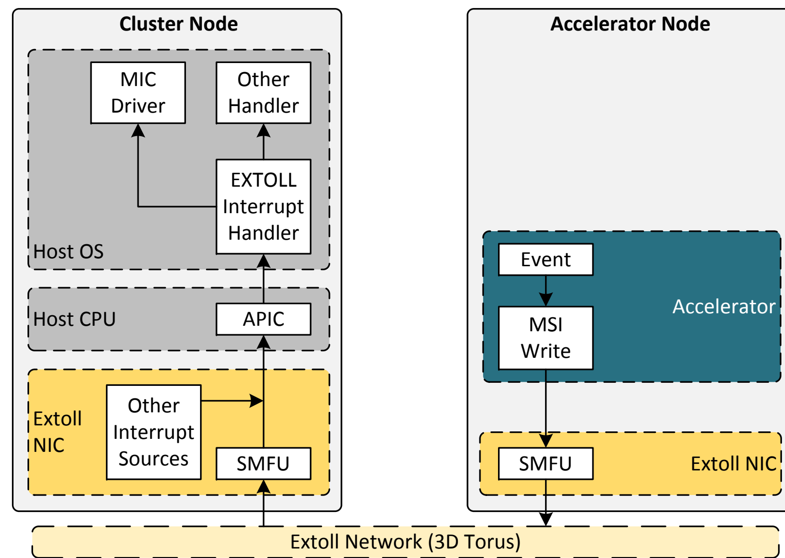


Figure 4.13: Interrupt handling within the booster.

special address region. This region is mapped by Extoll to a host system's MSI packet, which targets a valid APIC with a registered interrupt handler. Based on the payload data carried by the MSI packet, the interrupt handler can distinguish between Extoll and accelerator interrupts. Figure 4.13 displays the flow of an interrupt triggered by an accelerator node. The interrupt is forwarded over the Extoll network to the NIC of the target host, which in turn issues an Extoll interrupt.

4.5.3 Design Space Analysis

The NAA software implementation should be as transparent as possible to upper software layers. By providing a "virtualizing" software layer which is close to the hardware involved, all user space software and most kernel space code should be completely unaware of the existence of the intermediate NICs between CPUs and accelerators. Taking this into account, possible candidates for the shim software layer implementation are the Linux kernel, the accelerator driver itself, and the Extoll driver modules, which are explored in the following sections

4.5.3.1 Approach I: Linux Kernel Patch

Since the kernel is in full control of the system, it should be possible to add or modify code, to make the graphics driver work as expected. Furthermore, there would be no licensing issues for distribution. However, the kernel development is rather difficult and tricky, especially for the inexperienced. Furthermore, if there are any changes necessary in parts of the kernel, that can not be compiled as loadable module,

development would be even more tedious. Another big disadvantage is, that the kernel is constantly under heavy development and every change to any interface used, would require adjusting and retesting the patch set. Even worse than this would be the necessity to debug again, when the patch stops working due to more subtle or unrelated changes, that result in different behavior instead of compiler errors. Finding and debugging such issues would be very difficult and cumbersome. Overall this approach, while certainly possible, does not seem very attractive, especially in regard of long term maintenance.

4.5.3.2 Approach II: Accelerator Driver Modification

Another way of to enable the Network-attached accelerator approach from a software perspective is to modify the accelerator driver. Unlike the Linux kernel, most accelerator drivers are closed source and available in a binary form, and only provide source code for the parts that need to be linked against the targeted kernel. On the other hand, the only aspects that should need to be changed are function calls related to either the PCI support library or to interrupts, which are mostly done from the available source code. It seems feasible to re-implement the required PCI functions and compile the driver against these replacements. In comparison to using a customized Linux kernel, a driver patch seems like a much easier approach, however one would still need to maintain a patch set against a third party software, which might still be actively developed and break these patches with a future release. Furthermore, the source code is not licensed as GPL, so there might be problems with publishing these patches. Most importantly however, such an implementation can only work with the patched driver, but it is desirable to have a generic solution, that can be transparently used with other devices, drivers, and software environments.

4.5.3.3 Approach III: PCI Express Device Virtualization

The third option is to leave all software unmodified and instead virtualize a PCI device in kernel space. Once such a virtual device would be inserted in the kernel's list of available devices, any software, including accelerator drivers, should see no difference when trying to access a device. While at first it may seem more complicated than the other approaches, its main advantage is that no part of the existing software stack needs to be modified. Interface changes in the kernel can still break the compatibility, but it should be much easier to fix these changes than patching the complete kernel.

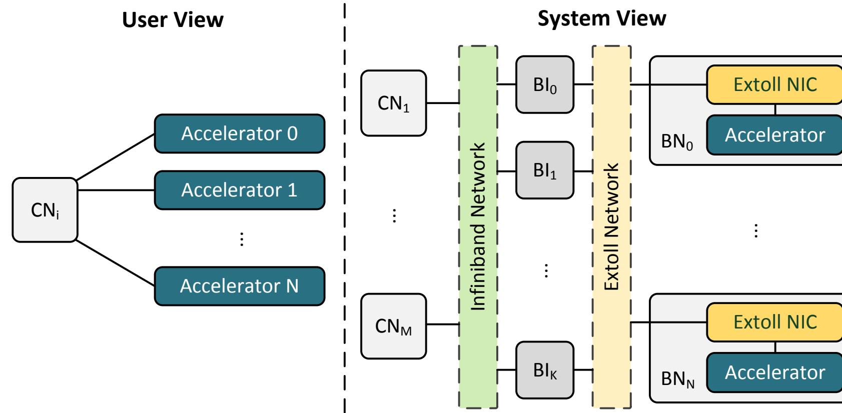


Figure 4.14: System and user view.

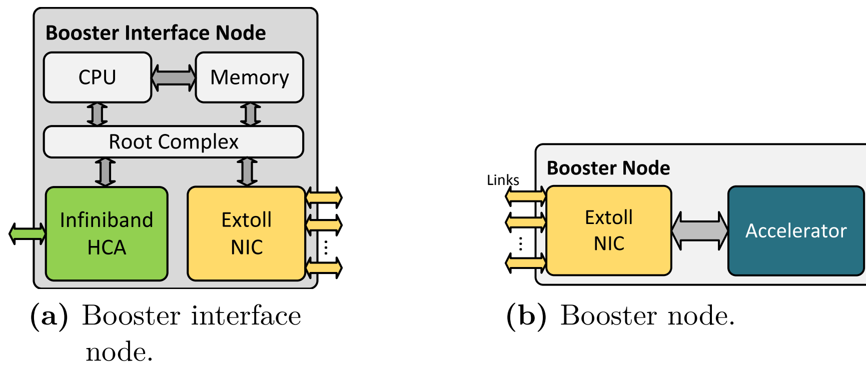
4.6 DEEP Booster Architecture

The DEEP Booster is the first system that deploys the network-attached accelerators approach. The software implementation follows approach II. The MIC device driver is modified in a way that PCI function calls are redirected to an Extoll-based kernel module. Figure 4.14 shows the system and user view of the DEEP booster system. The right part depicts the structure of an example booster cluster. The system consists of multiple compute nodes CN$_i$, $i \in \{0,\ldots,M\}$, booster interface nodes BI$_j$, $j \in \{0,\ldots,K\}$, and booster nodes BN$_k$, $k \in \{0,\ldots,N\}$, which represent the accelerator nodes. The CNs and BNs are connected with high performance interconnects that are not necessarily the same. The BIs configure the BNs and act as bridges between the different network protocols. In an ideal system, the same interconnection type is used system-wide and any CN can be used to configure a BN. The left part of the figure depicts how the booster communication architecture transforms the system view into a simplified user view. Each CN has access to all accelerators in the same way like to a locally connected accelerator device.

4.6.1 Hardware Components

Even though this work focuses on the software design for the network-attached accelerator approach, the hardware components of the DEEP system are briefly described to facilitate a better understanding. As mentioned before, the system architecture consists of three different entities.

A *compute node* (CN) is a standard cluster node, typically equipped with a super-scalar, multi-core CPU which provides a high single-thread performance. The CNs are connected over a high-speed network, in this context Infiniband.



(a) Booster interface node.

(b) Booster node.

Figure 4.15: Hardware components.

The *booster nodes* (BNs), see figure 4.15b, provide energy-efficient multi-threaded computing capabilities. The Extoll interconnect is used to connect the BNs in a 3D torus topology.

In the DEEP setup, a third node type is needed for the bring up and management of the remote accelerator devices, but also to bridge the communication between the CNs and BNs. The *booster interface* (BI) node, shown in Figure 4.15a, is equipped with two different NICs, one for the CN and one for the BN interconnection network respectively. The BIs enable software-bridged Cluster-Booster communication by translating between the two network protocols.

4.6.2 Prototype Implementation

The DEEP booster is the first prototype implementation of the NAA communication model. The booster architecture consists of booster node cards (BNC) that are connected by the Extoll interconnect, and booster interface cards (BIC). The following sections briefly described the implementation of the BIC and BNC, followed by an introduction to the booster low-level software stack. The DEEP Project has chosen the Many Integrated Core architecture (MIC) as the target accelerator technology. The innovative features of the Extoll NIC, e.g., the PCIe root port and the SMFU, allow to operate the MIC connected to the NIC without a host. One of the main advantages of using the Intel MIC technology is that the existing Extoll kernel modules can be utilized on the accelerator cards to source and sink network traffic.

4.6.2.1 Hardware Implementation

The counterpart of the BI is the BIC. The Extoll NIC has a memory-mapped region of $16 \cdot 8$GB and a region of $16 \cdot 128$KB to manage and address up to 16 accelerator cards. The SMFU maintains two intervals per accelerator to map the complete

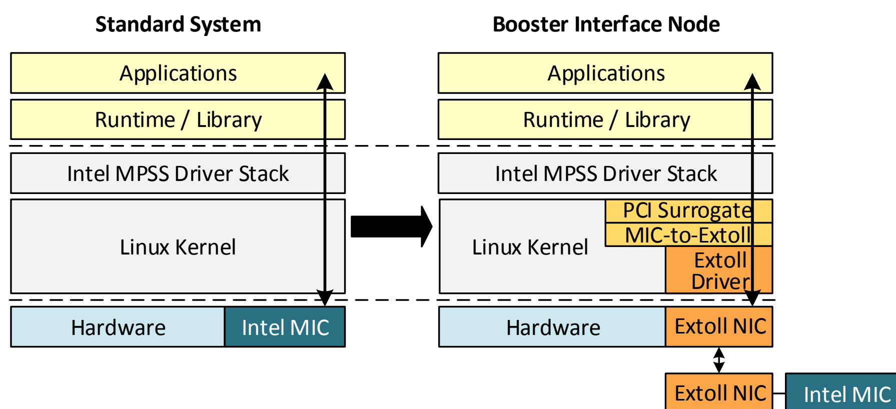


Figure 4.16: BIC software stack.

MMIO configuration register and the MemBAR GDDR5 memory of the BN's PCIe bus into the BICs address space. Four link ports from the NIC are used to connect the BIC with the 3D Torus of the booster. The BIC is responsible for booting and controlling the MICs over the Extoll interconnect. The BNC is a high-density implementation with two independent BNs, which are part of the 3D torus. A BN consists of a NIC and an accelerator, the Intel MIC. Eight BNCs are connected to a single backplane that provides connections to neighboring backplanes and their BNCs. As described in figure 4.12, the SMFU can be used to map regions of BI memory to BNs. On the BIC, two intervals are exported to the same BN. One exported region points to the BAR of the MMIO configuration register and the other region points to the BAR of the MemBAR GDDR5 memory. The MMIO registers of the Intel MIC are placed below 4GB, and the MemBAR is placed above 256GB to not interfere with the BIC's operating system memory map. This configuration is maintained for all BNCs. The communication between a BNC and CN does not require any CPU interaction on the BIC.

4.6.2.2 Software Implementation

The NAA Booster software stack implements all of the aforementioned requirements and consists of a collection of configuration scripts and a kernel module, which maps PCI support library calls and device structures onto the Extoll software stack resources. The main advantage of this approach is the transparency to upper software layers. Once the Intel MIC driver is running on top of the virtual PCI software stack, all existing applications can be used without any modification.

The configuration scripts are used to setup the SMFU interval mapping and to configure the PCIe Bridge unit to forward PCIe configuration packets to the remote

accelerators and vice versa. The scripts perform RRA reads and writes to the remote Extoll NICs by utilizing the user library libRMA. The kernel module intercepts the Intel MIC driver when it initializes the PCIe device and is responsible for maintaining and mapping necessary device structures onto the Extoll SMFU memory-mapped I/O regions. Figure 4.16 displays the software stack loaded on a BIC. The virtual PCI software layer comprises of two components:

PCI surrogate layer The PCI surrogate layer replaces the PCI support library. All PCI function calls are redirected to this layer.

MIC-to-EXTOLL Layer PCI structures needed by the Intel MIC driver are mapped to the Extoll data structures for device initialization.

To be able to use the BIC software stack, the Intel MPSS is recompiled against the PCI surrogate layer header file, which is built on top of the Extoll kernel API.

Device Configuration And Resource Mapping When the Intel MIC driver is loaded on a BIC, the driver initializes the connected MIC devices. Typically at module startup, the driver is registered with the PCI subsystem. The registration call is intercepted by redirecting it to the PCI surrogate layer, where a customized hardware initialization function is called. This is done by replacing the PCI header file include with `#include "pci_surrogate.h"`. Depending on the number of connected MICs, an array of MIC descriptor structures is initialized with the start and end addresses of the MemBAR and MMIO regions. These values are needed for booting the accelerators, since the PCIe client logic registers, referred to as SBOX registers, are accessed through the MMIO regions and the Linux image is copied into the remapped MemBAR region. Instead of allocating and mapping memory regions for each MIC, the memory regions are overlapped with the SMFU's MMIO space, which is subdivided into several intervals.

MSI Configuration and Interrupt Forwarding The second stage of the device initialization process is the MSI configuration of the MICs. This is done by writing directly to the corresponding SBOX registers, which reside in the mapped SMFU memory regions in the address space of the BIC. The vector is composed of a predefined address, a message, and the vector control. The predefined address enables the interrupt redirection to the BIC. The last step of the initialization process is the registration of the MIC interrupt handler with the EXTOLL interrupt handling subsystem. This is done by keeping a function pointer to the corresponding

interrupt handler within the Extoll interrupt management structure. The Extoll NIC has several functional units that are able to trigger an interrupt. The possible interrupts are divided into different *trigger event groups*. The Extoll driver manages possible interrupt sources in an array of function pointers, which are identified by unique tags. To handle interrupts issued by the MIC, the Extoll interrupt mechanism is extended by an additional event group and a flag indicates if MICs are present in the running system. When a hardware interrupt occurs on the Extoll card, the interrupt handler is called. The driver is able to identify the MIC's interrupt by its event group and redirects the interrupt by calling the corresponding function pointer.

4.6.2.3 Communication Paths

One of the most important advantages is the accelerator-to-accelerator direct communication between BNs. All accelerators in the communication architecture are directly connected to the network. As a consequence, the number of accelerators scales independently from the number of hosts. Another key feature is that an Intel MIC runs autonomously with its own Linux operating system after device setup. All Extoll low-level kernel modules, as well as the user space libraries, have been ported to the embedded operating system, which provides full access to all functional units of the Extoll NIC. With these hardware components, one-sided communication between accelerators is supported utilizing RMA PUT and GET operations to transfer large chunks of data. For small messages, the VELO unit supports MPI send and receive operations with very low-latency two-sided communication. The SMFU can be used to distribute parts of an accelerator's local memory to multiple different accelerators over the network and supports loads and stores to these memory regions.

The Extoll NIC provides the features necessary to build a scalable interconnection network. Figure 4.17 shows the possible communication paths within the DEEP architecture. Path (A) shows the accelerator-to-accelerator direct communication between two Intel MICs. During the boot and configuration process, path (B) is used for the OS image download, configuration, and status information. After the completion of this process, the Intel MICs can directly communicate with any other MIC in the system over path (A), receive workloads from the cluster or send results back over path (C). The coprocessor-only model for MPI applications strongly benefits from the communication path (A). All accelerators within the booster can be used to run parallel applications independently from any host.

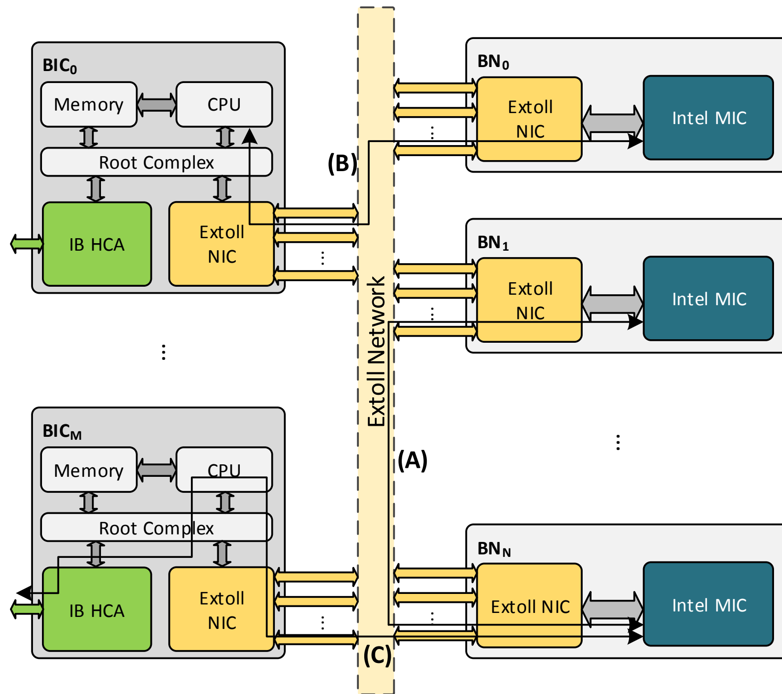


Figure 4.17: Communication paths.

4.6.3 Prototype Performance Evaluation

In this section, the test environment and performance evaluation of internode MIC-to-MIC communication using micro-benchmarks of the Extoll software stack, the OSU Micro-Benchmarks, and the LAMMPS molecular dynamics simulator, are described.

4.6.3.1 Hardware Environment

The BIC prototype node is a standard server machine with two Intel Xeon E5-2630 processors running at 2.30 GHz and 128 GB of memory. An Extoll NIC of the Galibier [107] generation is used, which utilizes a Xilinx Virtex6 FPGA design with a 128 bit-wide data path, running at 156.25 MHz, and one x4 16 Gbits/s Extoll link. The Galibier card has a memory mapped region of 16 GB, which can be used to manage two Intel Xeon Phi coprocessors (MICs) with 8GB GDDR RAM each. The network link of the Galibier card is used to connect to the seventh link of the BNC to get access to the 3D torus. The test environment contains one BNC with two Altera StratixV FPGAs. The FPGAs implement a Galibier-compatible 128 bit-wide data path running at 100 MHz and seven x4 16 Gbits/s links. Each StratixV Extoll NIC is connected to one MIC with a x8 PCIe Gen2 PCIe host interface. The StratixV FPGAs are connected to each other via an Extoll x4 16Gbits/s link.

4.6.3.2 Software Environment

The BIC runs CentOS 6.3 with kernel version 2.6.32-279.19.1.el6.x86_64 as the operating system, and has Intel MPSS 2.1.6720-16 and Intel Composer_xe_2013_sp1.2.144 installed. The micro-benchmarks of the Extoll software stack are used to evaluate the latency and bandwidth of MIC-to-MIC communication over Extoll. In addition, the MPI performance is evaluated between two MICs connected over Extoll (using OpenMPI 1.6.1) and directly connected to the BIC utilizing SCIF and OFED/SCIF (using Intel MPI Library 4.1.3.049 and OFED-1.5.4.1). OpenFabrics Enterprise Distribution(OFED) [108] provides an open-source software solution for RDMA and kernel bypass applications. The *Symmetric Communications InterFace* (SCIF) [87] is used for internode communication within a single system. Four different system configurations are used:

Booster This setup is the DEEP prototype booster system connecting two BNs with one BNC.

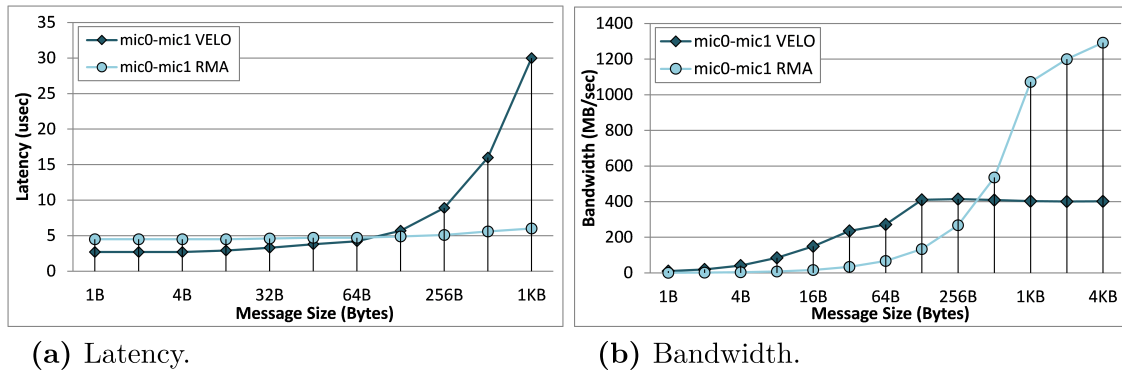
TCP/SCIF In this setup, two MICs are directly connected to the BIC over PCIe. The BIC acts as the host and runs the Intel MPSS without any OFED support. All communication is tunneled over SCIF.

OFED/SCIF The setup is an optimized version of the *mic0-mic1 TCP/SCIF* setup, where the Intel MPSS is run on top of the OFED stack. The communication is virtualized over the OFED/SCIF software stack, which implements RDMA by virtualizing direct access to a hardware InfiniBand Host Channel Adapter (HCA) between two MICs.

rEXTOLL Two hosts, equivalent to the BIC, are connected over Extoll. Each host has one MIC attached over PCIe.

4.6.3.3 Micro-benchmark Evaluation

For the micro-benchmark experiments, two prototype BNs are used, denoted as *mic0* and *mic1*. The micro-benchmarks are launched on mic0. The communication is set up over the low-level user-space library of the Extoll NIC. Figure 4.18a displays the results of the latency benchmark for the FPGA-based booster implementation. For messages smaller than 64B between mic0 and mic1, the latency performance of VELO outperforms the RMA unit by about 50%. Figure 4.18b presents the bandwidth results. For small messages, VELO provides a better bandwidth than RMA. The peak bandwidth provided by RMA is about 1.2 GB/s. These performance results



(a) Latency. **(b)** Bandwidth.

Figure 4.18: Micro-benchmarks performance of internode MIC-to-MIC communication using the Extoll interconnect.

have a direct impact on the direct accelerator-to-accelerator communication results. All communication traffic between the BNs is tunneled over Extoll.

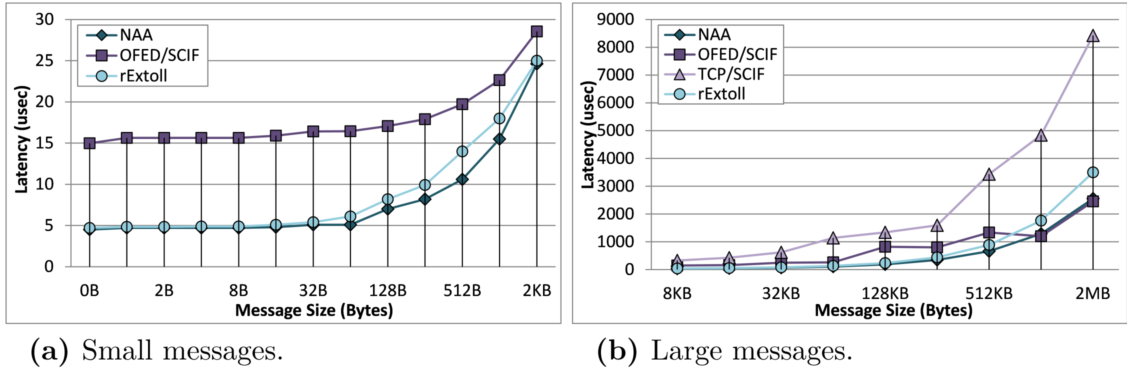
4.6.3.4 MPI Performance Evaluation

The point-to-point MPI benchmarks of the OSU Micro-Benchmarks 4.3 (OMB) [74] are used for the evaluation. Each benchmark is run 100 times. The results in the graphs are calculated as the arithmetical average of the runs. All benchmark results are verified by the Intel MPI Benchmark 3.2.3 (IMB) [109].

Latency Figure 4.19a displays the half round-trip latency results for small messages (<2 KB). The results for *mic0-mic1 TCP/SCIF* are not displayed, because the half round-trip latency is too large (>300 usec). Even though the prototype only uses an FPGA implementation of the NIC, the half round-trip latency using the booster architecture is improved compared to the latency measured when using the OFED/SCIF software stack. Furthermore, figure 4.19b shows that the half round-trip latency of large messages is also competitive compared to OFED/SCIF, although the bandwidth of the underlying hardware (PCIe Gen2 x16) is much higher (>6 GB/s).

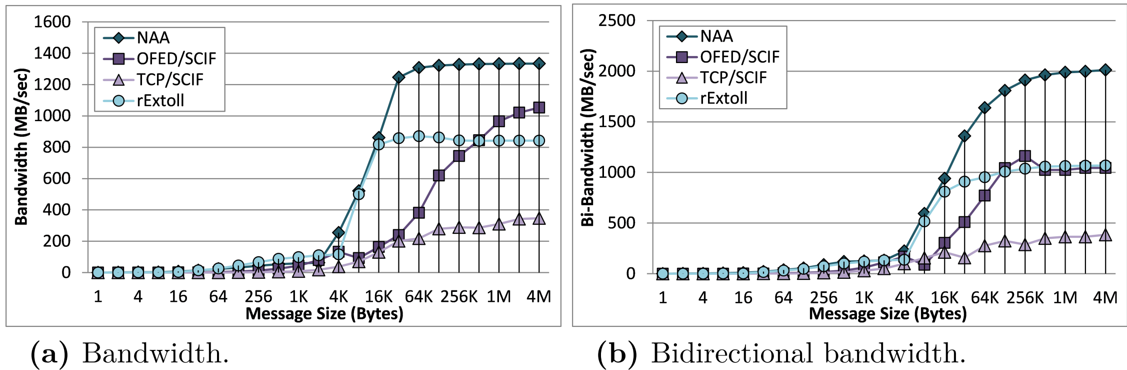
With the ASIC implementation of Extoll, the latency will be even smaller. The *mic0-mic1 TCP/SCIF* bandwidth is only displayed as a reference since it has a very poor performance. This is probably because of the need to perform a standard kernel-level TCP/IP communication on the MIC.

Bandwidth Figure 4.20 displays the performance results of the bandwidth and bidirectional bandwidth tests. The FPGA implementation is competitive with the OFED/SCIF solution, the peak bandwidth corresponds with the measured low-level



(a) Small messages. **(b)** Large messages.

Figure 4.19: Half round-trip latency performance of internode MIC-to-MIC communication using MPI.



(a) Bandwidth. **(b)** Bidirectional bandwidth.

Figure 4.20: Bandwidth and bidirectional bandwidth performance of internode MIC-to-MIC communication using MPI.

performance of the RMA unit. It is noteworthy that the peak MPI bandwidth using OFED/SCIF over PCIe is unable to utilize the bandwidth of the underlying PCI Express fabric.

4.6.3.5 Application Level Evaluation

In addition to the micro-benchmark and MPI performance evaluation, the communication architecture for network-attached accelerators is evaluated using a life science application. The MPI version of the LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) molecular dynamics simulator is used. LAMMPS is a classical molecular dynamics code [110]. It is written in C++ and MPI. The benchmark considered for the evaluation performs a bead-spring polymer melt of 100-mer chains, with finite extensible nonlinear elastic (FENE) bonds, Lennard-Jones interactions with a $2^{(1/6)}\sigma$ cutoff (5 neighbors per atom), and micro-canonical (NVE) integration. The problem has 32,000 atoms and runs for 100 time steps.

Table 4.1: Description of LAMMPS timings output.

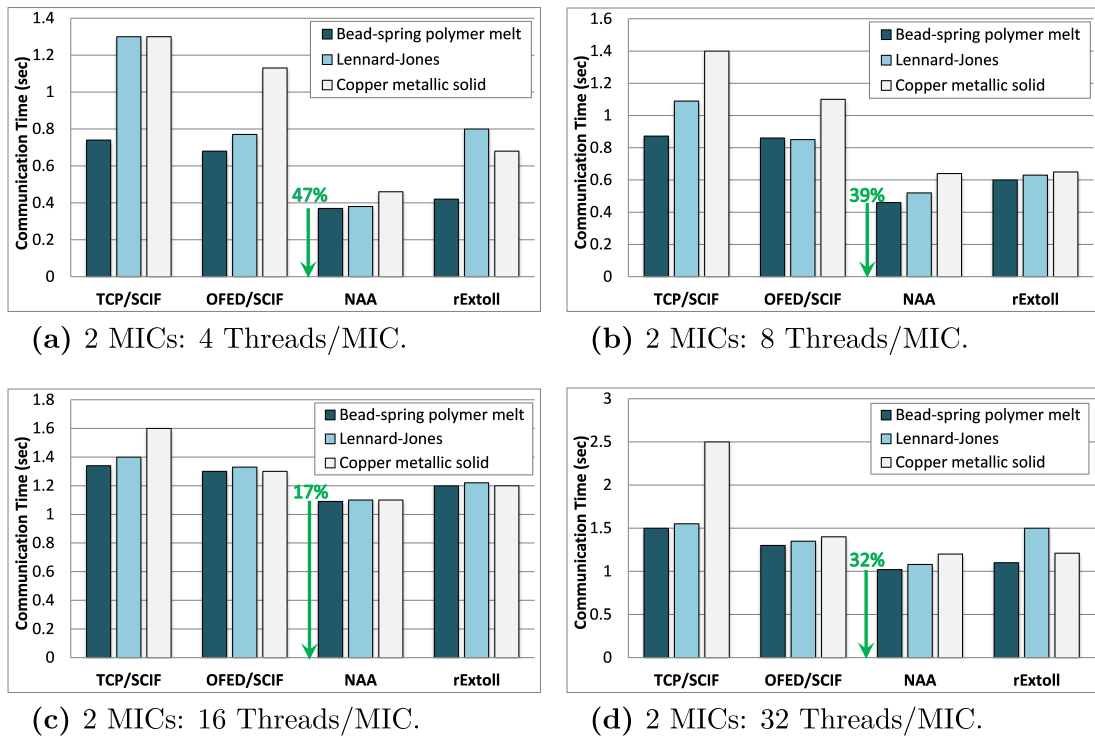
Name	Description
Loop	Total time spent in benchmark.
Comm	Time spent in communications.
Bond	Time spent computing forces due to covalent bonds.
Pair	Time spent computing pairwise interactions.
Neigh	Time spent computing new neighbor lists.
Outpt	Time to output restart, atom position, velocity and force files.
Other	Difference between loop time and all other times listed.

Figure 4.21 shows the impact of the communication architecture on the communication time for the LAMMPS Bead-spring polymer melt benchmark. The benchmark is run for 8, 16, 32, and 64 threads whereby the threads are equally distributed to the two MICs. It can be observed that the communication time for 32 threads/MIC is improved by 32%, while smaller runs with up to 4threads/MIC provide an improvement of the communication time up to 47%. Furthermore, running the LAMMPS simulator with a Lennard-Jones (LJ) benchmark (atomic fluid, LJ potential with 2.5σ cutoff (55 neighbors per atom), NVE integration) and the embedded atom model (EAM) metallic solid benchmark (metallic solid, copper EAM potential with 4.95 Angstrom cutoff (45 neighbors per atom), NVE integration) results in a similar improvement of communication time.

Figures 4.22a–4.22c display the overall application time for the bead-spring polymer melt, Lennard-Jones, and copper metallic solid benchmarks run with 32 threads/MIC. Table 4.1 provides a summary of timings used by figure 4.22. It can be seen that most of the application time is spent in communication. Therefore, the optimization of internode MIC-to-MIC communication time plays a crucial part in optimizing the overall application execution time.

4.6.3.6 Comments

Compared to the FPGA, the ASIC version of Extoll offers vastly improved network performance with its 128 bit-wide data path running at 750 MHz. As a result, the network link will provide an approximate bandwidth of 100 Gbits/s. As mentioned before, MVAPICH2-MIC is a proxy-based implementation of the MVAPICH2 MPI library. It reports a unidirectional bandwidth of up to 5.2 GB/s for internode MIC-MIC communication with InfiniBand HCAs. The usage of Tourmalet is expected to provide the sevenfold peak bandwidth.



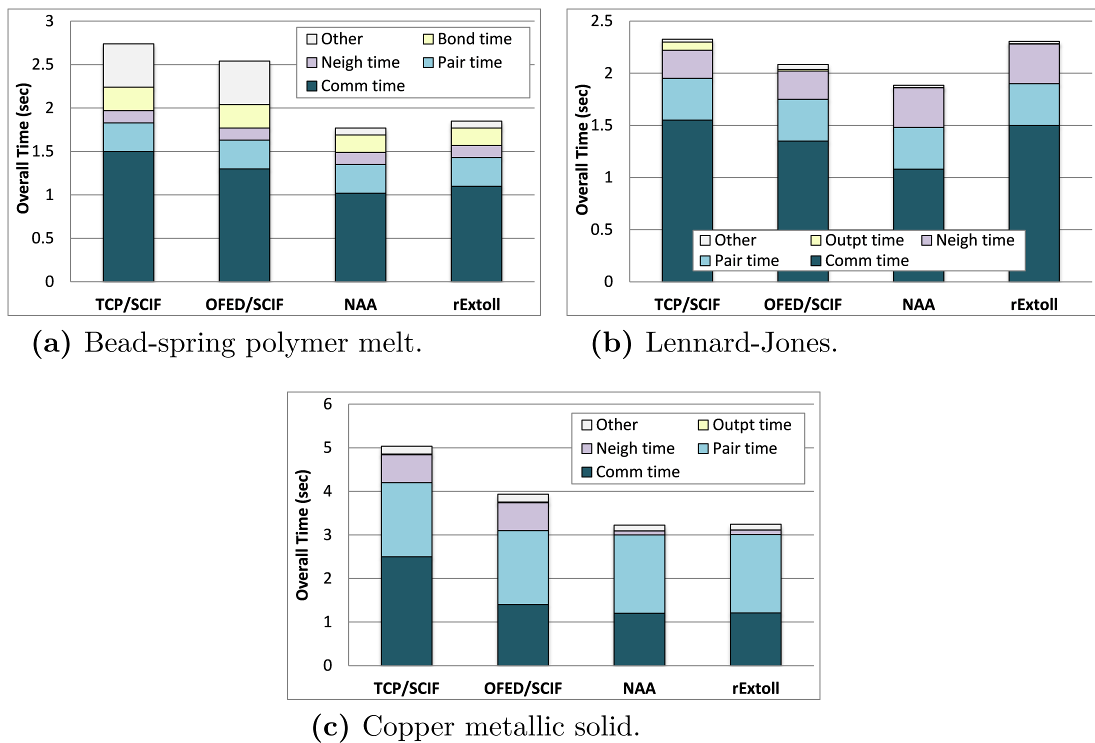
(a) 2 MICs: 4 Threads/MIC.

(b) 2 MICs: 8 Threads/MIC.

(c) 2 MICs: 16 Threads/MIC.

(d) 2 MICs: 32 Threads/MIC.

Figure 4.21: Communication time for the bead-spring polymer melt benchmark.



(a) Bead-spring polymer melt.

(b) Lennard-Jones.

(c) Copper metallic solid.

Figure 4.22: Overall application time for 64 threads, 32 Threads/MIC.

Figure 4.23: Production-ready GreenICE cube.

4.6.4 GreenICE – An Immersive Cooled DEEP Booster

The GreenICE technology [111] implements the NAA approach in a dense form factor and comprises of 32 accelerator nodes, each equipped with an Extoll Tourmalet and an Intel MIC of the Knights Corner generation. Instead of a booster interface node, the system is configured through a Raspberry Pi. The system utilizes an immersive cooling approach, which provides efficient heat removal from the high power components. Figure 4.23 presents a picture of the GreenICE cube. The chassis integrates all system components needed for operation, including the power supply units, the 3D torus cabling, the monitoring system and the cooling infrastructure. The GreenICE system was used as an ASIC evaluator system for the DEEP project.

4.6.5 Lessons Learned

The main advantage of the Intel MIC technology is that it is running its own operating system, and therefore, can leverage the full Extoll software environment. Also once up and running, Intel MICs can be accessed through SSH. The EXN interface (see section 5.4.6) is utilized to provide TCP/IP communication between the devices. Even though, the current implementation of the Booster architecture poses two major limitations:

- The Intel MIC device is not visible in the PCI system hierarchy, and therefore, is not recognized by the operating system. This limits the approach to devices

that are capable of running their own operating systems, and therefore, can run independently from a local host.

- The Intel MPSS code needs to be modified in order to utilize the PCI surrogate layer, but also when creating sysfs entries for the devices. This is mainly because the Extoll device is already registered in the Linux device model, which means that the same PCI device pointer cannot be registered twice.

4.7 Virtualization of Remote PCI Express Devices

While the DEEP Booster software relies on the modification of the accelerator driver, the Virtual PCI (VPCI) [112] software design targets a more generic approach without the need to modify driver code or the Linux kernel. VPCI is designed as a loadable Linux kernel module, which emulates a PCIe device by inserting a virtual device into the kernel's list of devices.

4.7.1 Concept Overview of VPCI

The NAA software requirements and design have been explained in section 4.5. The DEEP booster architecture only supports the Intel MIC technology. In today's HPC systems, GPUs are the predominant accelerator technology. Therefore, it is desirable to extend the NAA approach to GPUs. Recapitulating the findings from section 4.6.5, the VPCI module needs to fulfill the following tasks:

- The general PCI data structures as expected by the Linux kernel should be maintained. This way, the virtual PCI device can be added to the operating system's PCI tree without any modification of the kernel code and the accelerator device's software stack remains unchanged.
- The software needs a mechanism to detect and forward PCI configuration requests to remote accelerator devices, e.g., by implementing a software switch which distinguishes accesses to local and network-attached devices.
- The enumeration process for the virtual accelerator device should be triggered by VPCI. After the successful enumeration of the remote device, its resources can be accessed by user- and kernel-level processes, e.g., through sysfs.
- The solution should be able to transparently forward MMIO requests.

```
1 struct vpci_device {
2     int busnr;
3     int devnr;
4 
5     /* the actual pci device and bus structures allocated
6     and managed by the kernel. */
7     struct pci_dev *dev;
8     struct pci_bus *bus;
9 
10    /* BAR addresses as seen by the host. */
11    struct vpci_bar host_bars [7];
12    /* BAR addresses as seen by the GPU. */
13    struct vpci_bar remote_bars [7];
14    /* Stores the values read from BAR registers after writing
15    all ones to determine the size of the BARs. These should
16    be returned, when the kernel rescans the bus. */
17    u32 bar_init [7];
18 
19    /* Actual io spaces defined by the BAR registers. */
20    struct vpci_bar bar_windows [7];
21 };
```

Listing 4.1: Overview of the VPCI device structure.

- The VPCI module should provide a transparent mechanism to enable and disable the interrupt forwarding between the accelerator device and a host.

4.7.2 PCI Express Device Emulation

Note that in the context of this work, the concept of device simulation or emulation should not be confused with the actual emulation of a device's behavior, as it is done by programs like QEMU or VirtualBox. Instead, the term simulation refers to the concept that an accelerator device appears to be locally connected to a host. In general, there are two possible ways to emulate a PCIe device in the Linux kernel: (1) directly manipulate the kernel's device list, or (2) utilize abstraction layers.

The first approach would directly interfere with the kernel's device model. In theory, it is possible to insert a device structure into the device list and connect it with all related kernel objects. But, this concept is very similar to providing a customized kernel patch, and therefore, cumbersome and error-prone. Even though the modification of the kernel sources and recompilation could be avoided, the result would be similarly difficult to debug and maintain.

The most promising way to implement VPCI is to utilize the abstraction layers provided by the kernel, particularly the PCI layer and its generic API. All relevant addresses and values that are required for the virtual device are stored in a structure

```
1 /* Low-level architecture-dependent routines */
2 struct pci_ops {
3     int (* read)(struct pci_bus *bus, unsigned int devfn,
4         int where, int size, u32 *val);
5     int (* write)(struct pci_bus *bus, unsigned int devfn,
6         int where, int size, u32 val);
7 };
```

Listing 4.2: Generic PCI function pointers defined in <include/linux/pci.h>.

reflecting the virtual PCI device, as shown in Listing 4.1. When the VPCI module's initialization function is triggered, the steps explained in sections 4.7.3 to 4.7.6 are performed, which results in the enumeration of the virtual device and the allocation of all necessary kernel structures.

4.7.3 Forwarding PCI Configuration Space Requests

The Linux kernel offers a set of PCI-specific functions to read from and write to a device's configuration space. What is most interesting about these functions is that they are all implemented on top of two generic kernel routines. They can be accessed via pointers stored in `struct pci_ops` as displayed in Listing 4.2, which in turn are stored in `struct pci_bus`. All configuration requests are handled internally by these two functions. The Linux kernel stores its public symbols – global variables and functions that are not declared as inline or static – in a symbol table. For a symbol to be used by other modules, it usually needs to be exported with the `EXPORT_SYMBOL` or `EXPORT_SYMBOL_GPL` macros. However, non-exported public symbols are also stored in the symbol table. Each entry consists of the symbol's name and address. From user space, this symbol table can be viewed in */proc/kallsyms*. The Linux kernel provides the function `kallsyms_lookup_name()`, which can be used in kernel modules to look up a symbol's address by its name. This way variables or functions that are not exported can still be accessed.

The idea of VPCI is to replace the generic PCI function pointers with a pair of functions provided by the VPCI software. These functions check the targeted bus and device number, and if these numbers match the information of the virtual device, the request is forwarded to the remote accelerator utilizing Extoll's remote register file access mechanism. Otherwise, the parameters are simply passed to the original generic functions. This mechanism ensures the full functionality of all connected PCIe devices. Figure 4.24 displays the control flow; the dashed line illustrates the path of a normal operation.

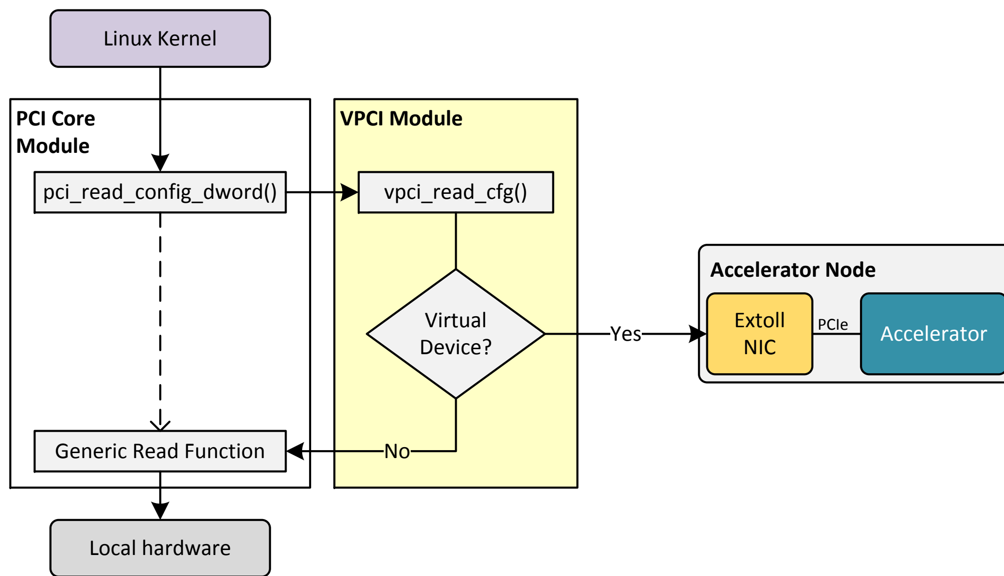


Figure 4.24: Control flow of PCI configuration space accesses.

4.7.4 Device Enumeration

With the configuration space of the accelerator being available, the bus just has to be rescanned to perform the enumeration process. For this purpose, the kernel provides a rescan functionality, which can be triggered by the VPCI kernel module. When the rescan has completed, the device is enumerated and all related kernel objects have been created. One limitation of the current design is that the VPCI design requires a free port on a host system to operate properly. Before loading VPCI, such a free port must be found and its secondary bus number be used for the virtual device. According to the rules for the actual enumeration process, the device and function numbers are both 0. Since the requirement of a free port per device poses a serious constraint on the usability of VPCI, it is possible to emulate a free port via an additional software abstraction.

4.7.5 Forwarding Memory-Mapped I/O Requests

The concept for forwarding memory requests has previously been described in section 4.5.2.3. All that needs to be done is to configure the SMFU on both the cluster and accelerator nodes in order to fulfill the requirements and to prevent the kernel from overwriting the accelerator's BARs once the setup has been completed. The latter issue can be prevented by simply handling such requests in VPCI's configuration functions. The SMFU is configured via the register file. On the local node, the VPCI module can directly interface with Extoll driver to configure the SMFU intervals.

The remote SMFU can be configured by issuing RMA put commands to the remote Extoll NIC's register file.

It is important that the software on the cluster node only requests BAR addresses through the generic PCI routines. In general, the SMFU uses 64-bit addresses, but the configuration space functions can only return 32-bit addresses for one BAR. On the remote side, there is no host from which the SMFU needs to request any memory. So, it has the whole 64-bit address range at hand and can assign a 32-bit address to the BAR. The same is done with the ROM Base Address, which also can only be assigned a 32-bit address. In summary, VPCI performs the following steps:

(1) Determine how many memory windows the device needs. Also, retrieve information about their size and whether they are 64 or 32 bits wide.

(2) Retrieve the physical start address of the host's SMFU memory window.

(3) Configure the individual memory regions on the host SMFU and store the addresses in the virtual device's resources.

(4) Configure the individual memory regions on the remote SMFU and write the addresses into the real accelerator's BARs. When doing this, the memory regions need to be matched with the host's memory windows. This is done by writing the addresses in the same SMFU register file entries.

4.7.6 Interrupt Delivery

Before the remote accelerator device can send any interrupts to a cluster node, VPCI needs to provide a way to register and de-register an interrupt handler in the Extoll interrupt mechanism. The following steps need to be performed by VPCI to enable interrupt delivery from the remote device to a node:

- Configure the MSI address in the device's configuration space;

- Register or de-register an interrupt handler when an accelerator driver allocates or frees an interrupt line.

4.7.6.1 MSI Configuration

PCIe devices utilize the MSI interrupts mechanism to communicate events to a host system. To signal an interrupt, the accelerator device sends a memory write packet to a predefined target address with a predefined payload. In order to enable interrupt delivery, VPCI needs to ensure that the MSI capability register set is initialized with

a target address that will trigger an Extoll interrupt. For this purpose, the address range starting at `0xFD00000000` is reserved.

Once the accelerator device's MSI Address register has been set to such an Extoll-specific address, VPCI needs to ensure that the address is not overwritten by the kernel or the accelerator driver with any other value. Since VPCI is in full control of all configuration space requests, it simple compares the request's offset parameter to the offset of the MSI address register in the configuration space and ignores write requests to this offset. In general, the exact target address is not defined, but can be determined by parsing a device's capabilities list. In case the kernel tries to check the MSI address to be properly initialized, VPCI returns the value the kernel expects, which ensures that the virtual PCI device passes any validation checks.

4.7.6.2 Interrupt Registration with Kernel Probes

One of the main obstacles of VPCI is proper registration and de-registration of interrupt handlers. The problem is that a driver usually does not register its handler when the module is loaded, but whenever its open function is called for the first time. E.g., for NVIDIA GPUs, the open function is called when a CUDA kernel is executed. Similarly, the handler gets de-registered on the last call to the driver's close function, not when the driver module itself is removed. Therefore, there is no deterministic point in time for VPCI to know when to register or de-register an accelerator driver's handler to or from the Extoll interrupt mechanism.

Originally designed to provide a dynamic kernel debugging mechanism, kernel probes (KProbes) [113] can be used to hook into any kernel routine. So called *JProbes* can be used to intercept any kernel routine, retrieve the routine's arguments and forward the call to a function that matches the target function's signature. Therefore, the generic approach to register and de-register an accelerator devices interrupt handler is to install the probes into the kernel's corresponding routines, which are `request_irq()` and `free_irq()`. However, `request_irq` is defined as an inline function, which directly passes its arguments to `request_threaded_irq`. Therefore, VPCI hooks functions into `request_threaded_irq` and `free_irq`. VPCI can distinguish the origin of an interrupt registration routine be comparing the driver module name with the accelerator device's driver module.

4.7.7 Overall Picture

The setup and configuration of a remote accelerator device comprises of several steps, which can be summarized as follows. The first step is to load the VPCI kernel

module, which in turn initializes and enumerates the remote accelerator device. In order to do so, VPCI needs to locate a free port on the system and configures the remote root port. Once the backdoor functionality of the remote Extoll NIC is setup, VPCI determines the types and sizes of the remote BARs, and, the local and remote SMFUs are configured.

The second step of the initialization is to replace the generic PCI read and write kernel routines. In addition, the VPCI module installs function hooks for interrupt registration and de-registration. Upon completion of the setup of the remote device and local device structures, a rescan of the bus where the virtual device is place behind is triggered to perform the enumeration of the virtual device. The virtual device is accessible over sysfs and the accelerator driver can be loaded.

4.7.8 Experimental Evaluation

At the time of this writing, a prototype implementation of VPCI is available, which is capable of operating one accelerator device. In the following, the test system configuration, benchmarks and initial results are presented.

4.7.8.1 Test System

The test system setup comprises of two server machines: one for the VPCI tests and one traditional setup. The VPCI test system is equipped with an Intel Xeon E5-2670 v2 dual socket CPU (10 cores per socket, codename: Ivy Bridge) running at 2.50 GHz, 32 GB of main memory and an Extoll Tourmalet 100G card. The traditional system is a server equipped with an Intel Xeon E5-2620 v2 dual socket CPU (6 cores per socket, codename: Ivy Bridge) and 24 GB of main memory.

Both systems run Ubuntu 16.04, kernel version 4.4.0, as operating system. Both systems utilize an NVIDIA Tesla K20c GPU, which is based on the Kepler GK110GL architecture. The GPU is operated with the NVIDIA driver version 352.99 for Linux x86_64 systems and CUDA (Compute Unified Device Architecture) version 7.5.

4.7.8.2 Benchmarks

For the prototype evaluation, two different benchmark suites are utilized: the CUDA 7.5 sample programs and the SHOC benchmark suite. The CUDA 7.5 runtime comes with several different sample programs to test the basic functionality of a connected CPU device. Of interest for this work are two tests: *bandwidthTest* and *simpleMultiCopy*. bandwidthTest is capable of measuring the memcopy bandwidth

Table 4.3: Overview of the benchmark results.

Benchmark	VPCI	Traditional
CUDA – bandwidthTest		
Host to device	4216 MB/s	5897 MB/s
Device to host	5314 MB/s	6553 MB/s
CUDA – simpleMultiCopy		
Memcopy host to device	4.09 ms (4.1 GB/s)	2.78 ms (6.03 GB/s)
Memcopy device to host	3.17 ms (5.29 GB/s)	2.51 ms (6.68 GB/s)
Fully serialized execution	4.31 GB/s	5.79 GB/s
Overlapped using 4 streams	5.99 GB/s	11.33 GB/s
SHOC		
BusSpeedDownload	4.41 GB/s	6.05 GB/s
BusSpeedReadback	5.53 GB/s	6.71 GB/s

across PCI Express from the host to the device and vice versa. simpleMultiCopy illustrates the usage of CUDA streams to achieve overlapping of kernel execution with data copies to and from the device.

The *Scalable Heterogeneous Computing* (SHOC) benchmark suite [114, 115] is a collection benchmarking programs written in OpenCL and CUDA to test the performance and stability of systems using computing devices with non-traditional architectures for general purpose computing. Of particular interest for this work are the level 0 benchmarks, which measure the low-level characteristics of GPU devices. *BusSpeedDownload* and *BusSpeedReadback* measure the bandwidth across the PCI Express bus connecting the GPU and the host.

4.7.8.3 Results

Table 4.3 presents the results for both the VPCI and the traditional systems with the previously introduced benchmarks. The SHOC benchmark is used to verify the accuracy of the CUDA samples. The current implementation of VPCI provides full access to the remote GPU, but comes with some performance limitations. There are two possible reasons for this behavior. First, the interrupt management of the accelerator device is integrated in the default Extoll interrupt handler. Depending on the amount of incoming interrupts, the serialization of the interrupt delivery poses a potential performance bottleneck. Second, VPCI currently does not configure the PCI Express maximum payload size. If the maximum payload is too small, the Extoll device cannot provide its full throughput capability.

Besides the design of VPCI, another source of performance limitation can be the PCIe connection itself. Previous work about GPUDirect RDMA peer-to-peer communication [116, 117] describes that Intel's Sandy Bridge and Ivy Bridge chip sets only provide poor support for non-posted peer-to-peer operations. The same behavior has also been observed for Intel Xeon Phi coprocessors [91].

4.8 NAA Summary

This chapter has introduced the network-attached accelerator approach, which describes a novel communication architecture for accelerator devices. Accelerators are placed on so called accelerator nodes and the Extoll interconnect technology is used to operate the devices remotely. The disaggregation of PCI Express hosts from their end-points (i.e., the accelerators) enables the optimal application-to-compute-resources mapping at runtime in an N to M ratio. In addition, NAA enables the direct communication between accelerator devices without any host CPU involvement.

This chapter has presented two NAA prototype implementations targeting different classes of accelerators. The first implementation evolved around the Intel Xeon Phi coprocessor of the Knightscorner generation. This particular class of accelerator devices runs their own operating system. With this unique feature, the Intel Xeon Phi coprocessors were used to build a stand-alone cluster of accelerator devices, which is called Booster. After configuring and booting the remote devices, they can be accessed through SSH and utilize native Extoll communication, e.g., through MPI. In particular, the EXN interface, which is presented in the next chapter, is used in the DEEP setup to provide IP addressing between the booster nodes. A test setup comprising of two Intel Xeon Phi coprocessors and two Extoll FPGA cards was used to provide an initial performance evaluation. The results of the MPI microbenchmarks and application evaluation with LAMMPS provide promising results, indicating that the communication time between accelerators can be drastically reduced.

The second implementation focused on the utilization of GPU devices. GPUs pose some unique challenges to the NAA approach. While the Intel Xeon Phi coprocessors can be operated just like real compute nodes, GPUs are typically used to offload massively parallel code. They are not designed to be operated autonomously. VPCI virtualizes accelerator devices by mapping their address spaces through Extoll. Devices appear to be locally attached and are visible in the Linux operating system's PCI hierarchy. A prototype has been used to demonstrate that it is possible to run

CUDA code on remote GPU devices. However, the current implementation provides only limited performance compared to a conventional heterogeneous system setup. Another question that needs to be addressed by future work is how multiple compute nodes can access remote GPU devices.

Nonetheless, this chapter has demonstrated that the Extoll interconnect technology can be used to build unconventional system architectures. With the upcoming exascale challenge, such innovative architectural approaches are needed to overcome the limitations of today's HPC infrastructures.

Besides novel communication architectures, the support of traditional communication schemes and protocols is another important pillar of modern system area networks. In the following chapter, the acceleration of traditional TCP/IP communication through RDMA over Extoll is introduced.

CHAPTER 5

RDMA-Accelerated TCP/IP Communication

Over the last decades, the Internet protocol suite, commonly referred to as TCP/IP, has become the predominant standard for end-to-end network communication and I/O. A vast majority of protocols relies on TCP/IP and expects IP addressing to identify network nodes. Yet, these legacy protocols pose some fundamental challenges to modern large-scale systems and applications. In general, traditional TCP/IP implementations require data copies between the application buffers and socket buffers in kernel space on both the sender and the receiver side, which typically is performed as programmed I/O by the CPU. In addition, traditional TCP/IP communication requires segmentation, reassembly, and transport handling. Another challenge is introduced by the Sockets API, which is the common software network interface, since it requires two-sided communication.

For large-scale HPC environments, these operations consume substantial CPU resources and memory bandwidth, and introduce potential performance bottlenecks. However, most modern network technologies offer RDMA capabilities, which provide the means to bypass the operating system for data transfers and permit for low-latency, high-throughput networking. Extoll is such an RDMA-capable networking technology, which can be used to accelerate the traditional TCP/IP data path through RDMA. By providing TCP/IP protocol support over Extoll, the performance benefits of the interconnect can be leveraged by a broader range of applications, including the seamless support of legacy codes.

This chapter introduces the design and implementation of *Ethernet over Extoll* (EXT-Eth) and *Direct Sockets over Extoll* (EXT-DS), which provide TCP/IP communication means for the Extoll interconnect technology. Both protocols leverage

OSI Layer	TCP/IP Layer	Example Protocols
Application Layer (L7)	Application Layer	HTTP, UDS, FTP, SMTP, POP, Telnet, TSL/SSL
Presentation Layer (L6)		
Session Layer (L5)		
		SOCKS
Transport Layer (L4)	Transport Layer	TCP, UDP, SCTP
Network Layer (L3)	Network Layer	IP (IPv4, IPv6), ICMP
Data Link Layer (L2)	Link Layer	Ethernet, FDDI, Token Bus, Token Ring
Physical Layer (L1)		

Figure 5.1: Comparison of the OSI and TCP/IP reference models.

the RDMA capabilities of the Extoll NIC. While EXT-Eth provides IP addressing and address resolution through asynchronous, two-sided RDMA read operations, EXT-DS targets the acceleration of sockets communication by providing transport offload with kernel bypass data transfers. This chapter summarizes and extents two workshop contributions [13, 14]. The remainder of this chapter is structured as follows. First, an overview of the Internet protocol suite is presented followed by a summary of related work. The main part of the chapter focuses on the design of the two protocols. The chapter concludes with an initial performance evaluation.

5.1 Introduction to the Internet Protocol Suite

A networking protocol is a set of rules defining how information is to be transmitted across a network [118], and typically, organized in several layers. The *Internet Protocol Suite* is such a layered networking protocol, as depicted in Figure 5.1, which provides a set of communication protocols used for end-to-end data communication. Its main application area is the Internet and internetworks, but it is also deployed on a wide range of similar computer networks such as system area networks (SANs) in HPC and data centers. First developed in the mid 1970's by the Defense Advanced Research Projects Agency (DARPA), the *Transmission Control Protocol* (TCP) and the *Internet Protocol* (IP) are the fundamental and most widely used protocols of the suite, which is the reason the Internet protocol suite is commonly referred to as the *TCP/IP reference model*. In comparison with its successor, the *Open Systems Interconnection model* (OSI model), TCP/IP consists of less layers. Figure 5.1 presents an overview of the two reference models and their layers.

In general, the TCP/IP suite is divided in four abstraction layers. The *application layer* provides the user with services to create user data and communicate this data

to other applications on the same or another node. The communication can be characterized by different models such as the client-server model and peer-to-peer networking. The *transport layer* provides end-to-end communication services for applications. The *User Datagram Protocol* (UDP) and TCP are the main protocols of this layer. The *network layer* provides a uniform networking interface. It defines the addressing and routing structures, while hiding the actual topology of the underlying network. The main protocol of this layer is the *Internet Protocol* (IP). The *link layer* is the lowest layer and consists of the device driver and the actual hardware interface, i.e., the network interface controller. The layer is concerned with transferring data across a physical link. One important characteristic is the *maximum transmission unit* (MTU), which defines the upper limit of the size of a frame.

5.1.1 The Network Layer: IP

The network layer is responsible for the packet delivery from the origin to the destination node and performs a variety of tasks. These tasks include the breaking of data into fragments small enough for transmission via the link layer and the routing of data across the Internet, but also the supply of services to the transport layer.

The main protocol in the network layer is the Internet protocol (IP). The first version of IP was introduced in 1981 [119] and is commonly known as the Internet Protocol version 4 (IPv4). It provides a 32-bit addressing scheme for identifying subnets and hosts. In 1998, the IP version 6 (IPv6) standard was released [120], which introduces 128-bit addresses, thus, providing a much larger range of addresses to be assigned to hosts.

IP Datagrams IP transmits data in packets called *datagrams*, which are sent independently across the network. An IP datagram consists of a header and its payload. The header can have up to 60 bytes and contains the destination address, so that the datagram can be routed, but also the source address of the packet. The size of the payload is specified by the upper limit on the size of a datagram, which must at least match the *minimum reassembly buffer size* specified by IP.

Transmission Mode IP is a connectionless, unreliable protocol. It neither provides the concept of a virtual circuit connecting two hosts nor guarantees that packets are delivered in order, will not be duplicated, or even arrive at all. In addition, IP has no error recovery mechanism. Reliability must be provided either by the transport protocol or by the application itself.

Fragmentation and Reassembly IPv4 datagrams can be up to 65,535 bytes in size, while the default IPv6 datagram size is 65,575 bytes (with 40 bytes for the header and 65,535 bytes for the payload) and can be scaled up to 4 GiB with so called *jumbograms*. When the IP datagram size exceeds the MTU of the link layer below, IP fragments the datagrams into suitably sized units for transmission across the network. The fragments are then reassembled at the destination. A major disadvantage of this mechanism is that a datagram can only be reassembled if all of its fragments arrive at the destination; IP does not perform packet retransmission.

5.1.2 The Transport Layer

Residing on top of the network layer, there are two primary transport layer protocols: the *Transmission Control Protocol* (TCP) [121] and the *User Datagram Protocol* (UDP) [122]. UDP provides an unreliable datagram service, while TCP provides flow-control, connection establishment and reliable data transmission. To establish end-to-end communication and to be able to differentiate applications running on a host, both protocols use a 16-bit *port number*.

5.1.2.1 User Datagram Protocol

UDP mainly adds two features to IP: port numbers and a data checksum for error detection and data integrity checks. UDP is connectionless and unreliable. If an application using UDP as the transport protocol requires reliability, this must be implemented by the application itself.

When comparing UDP and TCP, one questions that arises is why one should use UDP at all. For extensive details, the reader is referred to chapter 22 of the book *UNIX Network Programming Volume 1* by Stevens et al. [41]. Some reasons can be summarized as follows. A UDP server can receive datagrams from multiple clients without the need to establish a connection for each client. UDP can be faster for simple request-response communications. Also, UDP sockets permit multicasting or broadcasting of datagrams to multiple or all nodes connected to the network.

5.1.2.2 Transmission Control Protocol

TCP provides a reliable, connection-oriented, bidirectional, error-checked packet delivery service for byte-streams (octet-streams) between two TCP endpoints (i.e., applications). It offers a suitable transport method to user applications which rely on reliable session-based data transmission such as client-server databases and e-mail

clients. In accordance with the Internet standard, TCP must perform the following tasks to provide the described connection features.

Connection Establishment TCP needs to establish a connection prior to the data transmission. A TCP connection is initialized through a so called *three-way handshake*, which exchanges options to advertise connection parameters and returns control information used to establish a virtual communication channel between the two TCP hosts. When a TCP connection is closed, a similar handshake mechanism is used to ensure that no data segments are lost on the sending or receiving side.

Fragmentation TCP fragments user data into segments. Each segment contains a checksum to verify data integrity and is transmitted in a single IP datagram.

Acknowledgments, Retransmissions, and Timeouts TCP ensures reliability by implementing an acknowledgment mechanism. After receiving a TCP segment without an error, TCP sends a positive acknowledgment to the sender. Otherwise, an erroneous segment is discarded and no acknowledgment is sent. Lost and erroneous segments are handled by a timer for each segment on the sender side. If an acknowledgment is not received within the stipulated time, the segment is retransmitted.

Sequencing For each byte transmitted over a TCP connection, a logical sequence number is assigned. Each TCP segment has a field that contains the sequence number of the first byte in the segment. This number allows TCP to reassemble segments at the destination in the correct order before passing the byte stream to the application layer. In addition, the receiver can use the sequence number to acknowledge successfully received segments and to eliminate duplicates.

Flow Control Each TCP endpoint maintains a buffer for incoming data. The TCP flow control mechanism on the receiver side prevents fast senders from overwhelming a slow receiver by advertising the available space in the incoming data buffer with each acknowledgment. In addition, TCP employs the *sliding window* algorithm, which allows unacknowledged segments in total of up to N bytes to be in transit between sender and receiver with N being the offered window size. If the incoming data buffer is completely filled, the sliding window is closed and TCP stops transmitting segments.

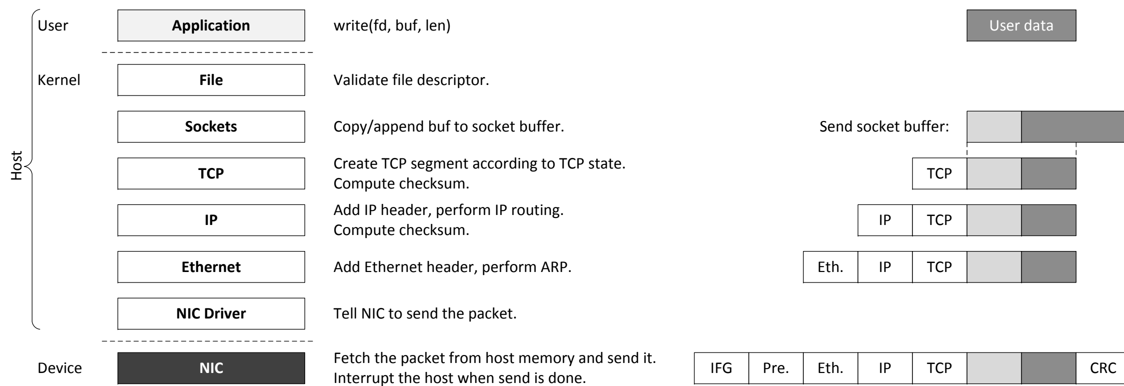


Figure 5.2: Overview of the TCP/IP data transmission path [123].

Congestion Control: Slow-Start and Congestion-Avoidance Algorithms TCP provides a congestion avoidance method called *slow-start* algorithm, which is designed to prevent a fast sender from overwhelming the network. Initially, the sender transmits segments at a slower rate, but exponentially increases the rate as the segments are acknowledged by the receiver until the transmission capacity of the network is saturated. In addition, *congestion windows* limit the amount of unacknowledged data that can be transmitted. Similarly to the slow-start algorithm, the window size increases exponentially until a specified threshold is reached.

5.1.3 Data Transmission and Reception in Linux

Figure 5.2 displays the data transmission path of a packet with the corresponding tasks performed by each layer. The TCP/IP layers can classified in three areas: user area, kernel area, and device area. All tasks in the user and kernel area are performed by the CPU. The device is represented by a network interface controller.

At the beginning of the send path, a user typically establishes a socket connection and initializes the send process, e.g., by calling `write()`. The file layer performs a simple file validation, and then, forwards the data to the sockets layer by calling the socket send function, which in turn copies the data to the corresponding socket buffer structure. For this purpose, the kernel socket has two buffers: the send socket buffer and the receive socket buffer. The send socket buffer is fragmented in TCP segments and appended with a header including the checksum and sequence number for the segment. From there, the segment is handed down to the IP layer where the it is encapsulated in an IP datagram. The IP layer adds an IP header and performs IP routing, which determines the next hop IP on the way to the destination node. Next, the Ethernet layer searches for the *Media Access Control address* (MAC address) of

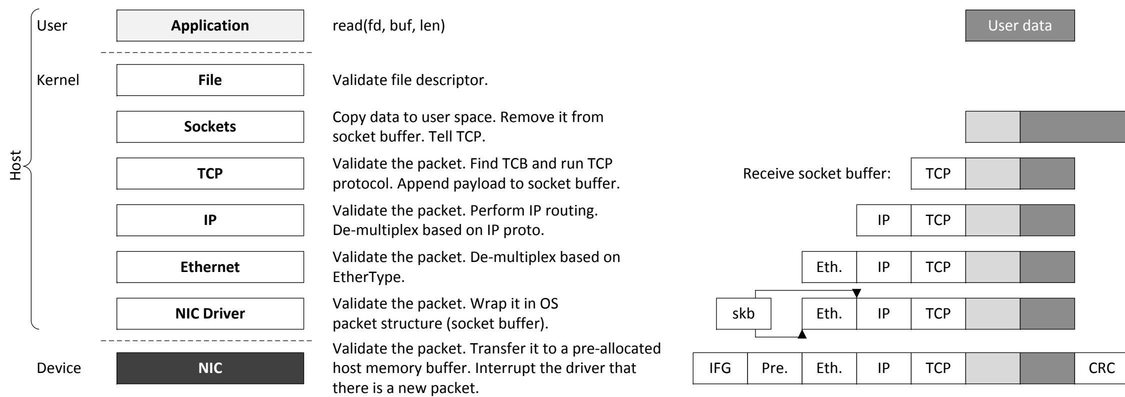


Figure 5.3: Overview of the TCP/IP data reception path [123].

the next IP hop by utilizing the *Address Resolution Protocol* (ARP). Afterwards, the layer adds the Ethernet header and then, the transmitting NIC driver is called to send the packet over the network fabric.

On the receiving side, a similar stack of layers is traversed by an incoming packet. Figure 5.3 displays the different layers and corresponding tasks performed by each layer. The NIC transfers the incoming packet to a pre-allocated host memory buffer and sends an interrupt to the host operating system. The NIC driver then checks whether it can handle the new packet, wraps it in a socket buffer and sends it to the upper layer. The Ethernet layer checks whether the packet is valid and then de-multiplexes the network protocol, in this case the IP layer. The IP layer also checks whether the packet is valid, and then, determines whether the packet's final destination is the host or another system. If it is meant for the local host, the IP layer de-multiplexes the transport protocol from the IP header. In this example, IP sends the packet to the TCP layer. Like the other layers, TCP also validates the packet and then, searches the TCP control block of the corresponding connection. If new data has been received, the TCP layer appends the data to the corresponding receive socket buffer. When the application calls the read system call, the data in the socket buffer is copied to the user memory and removed from the socket buffer.

5.1.4 Interrupt Coalescing and NIC Polling with NAPI

In traditional, purely interrupt-driven systems, each received packet triggers an interrupt in order to inform the system that a new packet is waiting to be processed. Under high traffic load, the high priority of interrupts leads to a state known as *receive livelock* [124]. When a system enters this state, it will dedicate its CPU resources to interrupt handling, while the actual packet processing as well as other processes will

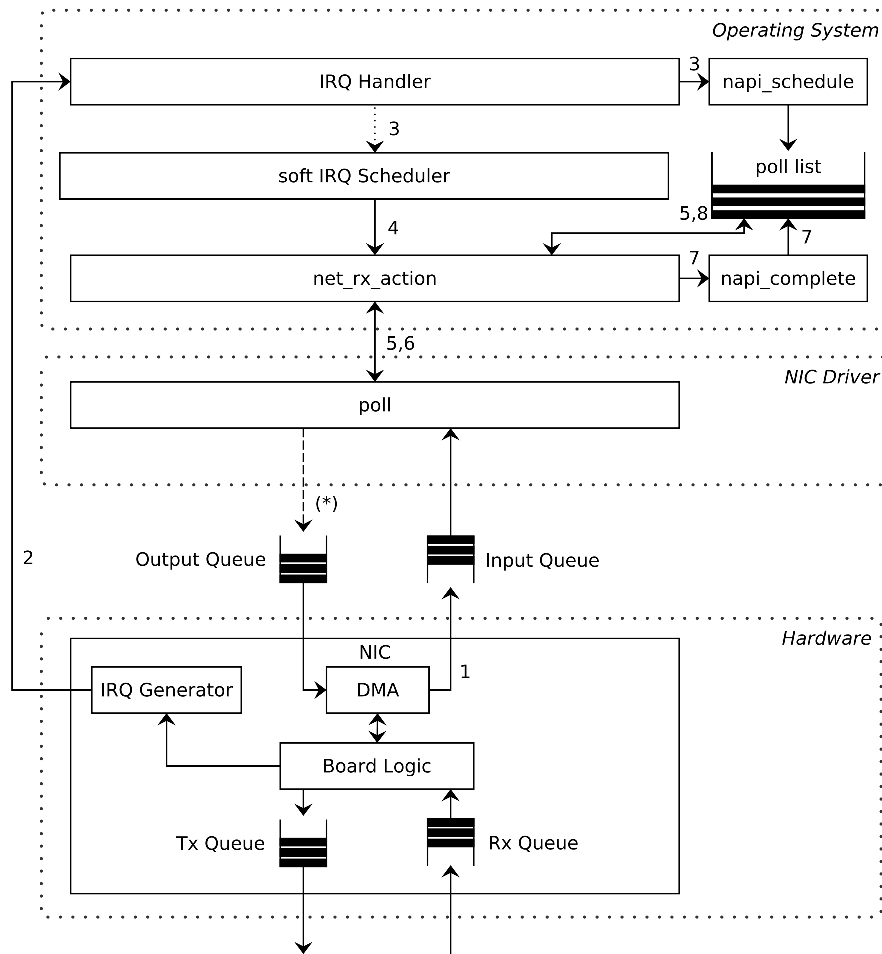


Figure 5.4: Schematic overview of the NAPI functionality [126].

starve. The *New API* (NAPI) [125] is an interface that implements an extension to the traditional device driver packet processing framework. Its major design objective is to improve the performance of high-speed networking while reducing the workload on the CPU. The NAPI design incorporates the following two ideas:

Interrupt Mitigation High-speed networking can create thousands of interrupts per second. NAPI allows device drivers to disable interrupts during times of high traffic load with a corresponding decrease in system load. Instead, NAPI offers a polling functionality on incoming packets.

Packet Throttling When the system is overwhelmed by the amount of incoming packets and must eventually drop them, it is better if those packets are dropped before much effort goes into processing them. NAPI-compliant device drivers can often cause the packets to be dropped by the network card itself before the kernel sees the packets at all.

Figure 5.4 outlines the schematic flow of the NAPI functionality. As described in Beifuß et al. [126], a NAPI-compliant device driver performs the following steps to process incoming packets:

(1) The NIC's DMA engine copies an incoming packet from the RX queue residing on the NIC to an input queue in the main memory.

(2) The NIC triggers a hardware IRQ, which is served by the assigned CPU core. The mapping between an IRQ and a CPU core can be statically assigned.

(3) The interrupt request (IRQ) handler then enqueues an entry referring to the input queue into the poll list and triggers a soft IRQ.

(4) The soft IRQ scheduler completes the soft IRQ and invokes the networking functionality.

(5) `net_rx_action()` peeks the first entry of the poll list and initiates the polling functionality. The implementation of `poll()` is device driver-specific.

(6) A poll returns for two reasons:

    (a) The corresponding input queue is empty.

    (b) `poll()` yields after processing a certain number (specified by a budget) of packets to prevent other input queues from starving.
    → Continue with step 8

(7) The respective entry is removed from the poll list and poll returns.
→ Continue with step 9

(8) The current poll is suspended although the input queue is not empty. The respective entry is re-enqueued into the poll list in a round-robin manner.

(9) If the poll list still contains entries, NAPI continues with step 5. Otherwise, the algorithm terminates.

5.1.5 TCP/IP Protocol Overhead and Bottlenecks

Most modern operating systems implement a networking stack that is based on the TCP/IP reference model, and therefore, adopt its overhead characteristics and bottlenecks. At the time TCP/IP was developed, system resources such as memory were limited, network data rates were low, and network processing at line rates was not possible due to the limitations in the NIC design. The main objective of the networking stack was to provide a good balance between ease of use, performance, and low memory usage.

Table 5.1: TCP/IP protocol overhead for Gigabit Ethernet under Linux 2.6.18.

Processing	**Time (ns)**	**Percentage (%)**
System call and socket	1,779	16.14
TCP	2,356	21.38
IP	1,120	10.16
Protocol Handler invocation	430	3.9
Device Driver	1,987	18.03
Hardware Interrupt	1,770	16.06
NIC + Media	1,580	14.33
Total (round trip)	**11,022**	**100**

5.1.5.1 Overhead

For system area networks, the TCP/IP protocol introduces a lot of unnecessary overhead, which in turn reduces the overall bandwidth. Table 5.1 presents a summary of the TCP/IP overhead breakdown analysis carried out by Larsen et al. [127].

Analyzing the overhead from the different layers within the TCP/IP model, it is evident that the stack is not optimized for reliable system area networks. One of the most time consuming and expensive TCP operations is the checksum generation for data integrity. Typically, this checksum is computed by the host CPU for every frame. Modern NICs provide a *cyclic redundancy check* (CRC) in hardware, which is an error-detecting code used to detect accidental changes to raw data. An additional source of overhead is introduced by the Internet Protocol. When TCP passes packets down to the internet layer, IP splits up the original TCP packets and encapsulates them in additional headers.

5.1.5.2 Bottlenecks

Besides the protocol overhead introduced by TCP/IP and its layers, *bus contention* and *system calls* introduce potential hardware bottlenecks for high-speed SANs.

As outlined in section 5.1.3, a NIC receives a packet, and then, transfers it from its on-board memory to a socket buffer structure residing in the main memory of a host. Although this transfer is carried out by a DMA controller, memory pressure still exists. In the end, the data is copied to a user space buffer, which is accessible by an application. Hence, high-speed packet processing stresses the shared bus capabilities of modern architectures, since it requires a great deal of bandwidth between the CPU and the memory and I/O subsystems, and results in bus contention.

For TCP/IP, there are two important system call properties to be considered. First, software interrupts or traps are used by a CPU to communicate with the kernel. Second, an issued system call has to return before an application can continue to work in user mode. Due to the blocking behavior of system calls such as `send()` and `recv()`, expensive wake-up operations are needed.

5.2 Related Work

Numerous research has been targeting the overall optimization of the TCP/IP stack. A detailed summary can be found in a survey paper by Hanford et al. [128]. Of particular interest for this work are implementations of either TCP/IP or Ethernet over high-performance SANs. While implementations such as *IP over Gemini Fabric* [129] are just briefly mentioned, the Infiniband technology offers a vast variety of research studies targeting the seamless support of legacy applications.

The following sections provide an overview of the OpenFabrics Enterprise Distribution (OFED) targeting Infiniband and Ethernet networks, and a summary of sockets-like interface implementations.

5.2.1 OpenFabrics Enterprise Distribution

The *OpenFabrics Enterprise Distribution* (OFED) [108] is an open-source software stack offering different network adapter drivers for Infiniband and Ethernet devices, middle/upper layer kernel core modules and related libraries and utilities for RDMA and kernel bypass applications. Figure 5.5 provides an overview of the supported protocols and interconnects.

IP over Infiniband *IP over InfiniBand* (IPoIB) [130, 131, 132] is a protocol that specifies how to encapsulate and transmit IPv4/IPv6 and Address Resolution Protocol (ARP) packets over Infiniband. Therefore, IPoIB enables IP-based legacy applications to run seamlessly on an Infiniband fabric. IPoIB is implemented using either the *unreliable datagram* (UD) mode [130] or the *reliable connected* (RC) mode [132]. In Linux, the *ib_ipoib* kernel driver implements this protocol by creating a network interface [84, chapter 17] for each Infiniband port on the system. This way, an HCA acts like an Ethernet NIC. Every such IPoIB network interface has a 20 bytes MAC address, which may cause problems since the "standard" Ethernet MAC address is 6 bytes (48 bits) in size. Also, the IPoIB protocol does not fully utilize the HCA's capabilities such as it does not implement any kernel bypass, reliability, RDMA, and

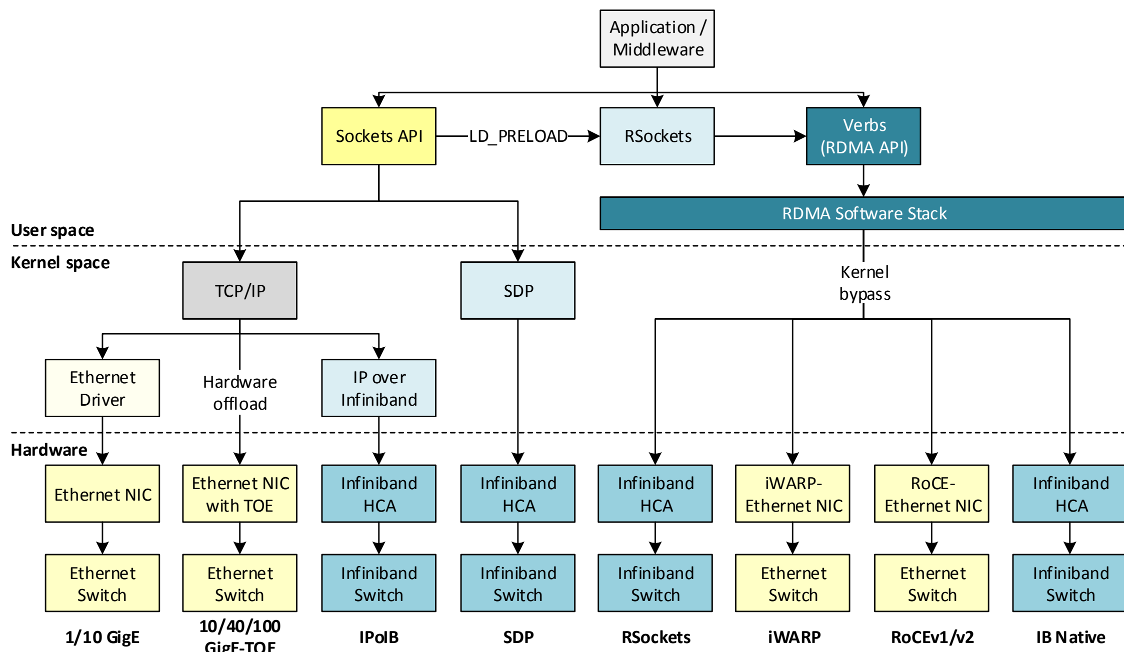


Figure 5.5: Overview of interconnects and protocols in the OpenFabrics stack.

splitting and assembly of messages to packets. The network traffic traverses through the normal IP stack, which means a system call is required for every message and the host CPU must handle breaking data up into packets.

In recent years, there have been efforts to cope with the limitations of IPoIB. The first attempt is the introduction of *user space Ethernet verbs* [133], which bypasses the TCP/IP stack for Ethernet frames. A similar approach is proposed by user space IPoIB packet processing over Verbs [134]. Also, an acceleration to the IPoIB kernel modul itself is proposed [135], including interrupt moderation and RDMA capabilities. The latest approach introduces *Ethernet over Infiniband* [136] as a replacement for the IPoIB kernel module. It decouples the Ethernet link layer from the underlying Infiniband network, which is a must for virtualization.

Sockets Direct Protocol The Sockets Direct Protocol (SDP) [137], included as an annex of the Infiniband specification, was a first attempt to implement a transport-agnostic protocol to support TCP-like stream sockets over an RDMA-enabled network fabric. The initial implementation of SDP used a buffer copy method similar to BSD sockets, therefore referred to as *BCopy mode* and provided support for zero-copy data transfers for asynchronous I/O operations. Later, the zero-copy mode, referred to as *ZCopy*, was expended to support the synchronous socket calls `send()` and `recv()`. The first attempt to implement a ZCopy mode [138] pinned and registered

the application buffers in the SDP implementation and supported two different modes: *Read ZCopy* and *Write ZCopy*. The ZCopy mode utilized Infiniband's *Fast Memory Region* mechanism to transfer data between two HCAs. However, it did not allow simultaneous send requests, a `send()` call would block until the data was received. This blocking behavior was necessary to prevent the modification of the user memory involved in the data transfer while being processed.

The *Asynchronous Zero-Copy SDP* (AZ-SDP) [139] allows multiple simultaneous send requests and introduces the `mprotect()` call as a safeguard mechanism, which forces a segmentation fault whenever a user modifies the memory region of an ongoing transfer. This protection mechanism results in an additional kernel trap for every data transfer, which forces the user application to block or copy the memory area. The main objective of SDP is to run with unmodified sockets applications. Therefore, this costly mechanism is needed since applications can reuse memory as soon as the sockets library returns control to the application. The SDP protocol has been deprecated by several different user space libraries providing the same functionality.

RSockets The *RSockets* [140] library implements a user space protocol for byte streaming transfers over RDMA, which provides parity with standard TCP-based sockets. It comes with its own blocking API, which is similar to standard socket calls such as `rsend()` and `rrecv()`, and typically performs buffer copies on both sides. Existing socket applications can utilize RSockets by using the pre-loadable conversion library, which exports socket calls and maps them to RSockets. A zero-copy functionality is available as a set of extra functions on top of RSockets, i.e., `riomap()` and `riowrite()`.

Internet Wide Area RDMA Protocol The *Internet Wide Area RDMA Protocol* (iWARP) enables RDMA over TCP/IP infrastructures, including zero-copy and protocol offload, if the underlying NIC provides RDMA functionality. iWARP is layered on the congestion-aware protocols TCP and the *Stream Control Transmission Protocol* (SCTP) [141] and is defined by a set of RFCs, specifically the *RDMA Protocol* (RDMAP) [142], *Direct Data Placement* (DDP) protocol [143], *Marker PDU Aligned (MPA) Framing* [144], and DDP over SCTP [145]. DDP is the main component in the protocol, which permits the actual zero-copy transmission. iWARP only supports reliable connected transport services and is not able to perform RDMA multicasts. Applications implementing the Verbs API can utilize iWARP.

RDMA over Converged Ethernet In contrast to iWARP, *RDMA over Converged Ethernet* (RoCE) [146] is an InfiniBand Trade Association standard designed to provide Infiniband communication on Ethernet networks. RoCE preserves the InfiniBand Verbs semantics together with its transport and network protocols and replaces the InfiniBand link and physical layers with those of Ethernet. RoCE packets are regular Ethernet frames with an EtherType allocated by IEEE which indicates that the next header is a RoCE value global route header, but they do not carry an IP header. Therefore, they cannot be routed across the boundaries of Ethernet L2 subnets. RoCE version 2 (RoCEv2) is a straightforward extension of the RoCE protocol that involves a simple modification of the RoCE packet format. Instead of the global route header, RoCEv2 packets carry an IP header which allows traversal of IP L3 Routers and a UDP header that serves as a stateless encapsulation layer for the RDMA transport protocol packets over IP.

5.2.2 Sockets-like Interfaces

GMSOCKS GMSOCKS [147] is a direct sockets implementation that maps standard Windows socket calls onto the Myrinet/GM device driver without the need to modify the source code, or relink or recompile the application. GM is a message passing system for Myrinet networks and includes a driver, a Myrinet-interface control program, a network mapping program, and the GM API, library and header files. GMSOCKS is implemented as a thin user space software layer in between the Winsock Direct architecture and the GM user library. The Windows socket calls are intercepted with the Detours runtime library [148], which provides dynamic interception of arbitrary Win32 binary functions at runtime on x86 machines.

Depending on the Winsock version, GMSOCKS utilizes the buffered copy or write zero-copy mode for data transfers. GMSOCKS uses so-called companion sockets to retrieve information about socket descriptors and match them with the corresponding GM information. Since GM provides only one receive queue, a worker thread dispatches incoming messages and inserts them into the corresponding receive queues of established point-to-point connections. GMSOCKS provides full semantics and is fully functional against the TCP/IP implementation.

Mellanox's Messaging Accelerator *Mellanox's Messaging Accelerator* (VMA) [149], formerly known as Voltaire Messaging Accelerator, is an open source project. It comes as a dynamically linked user-space library and implements the native RDMA verbs API. The VMA library does not require any code changes or recompiling of

user applications. Instead, it is dynamically loaded via the Linux OS environment variable, `LD_PRELOAD` and intercepts the socket receive and send calls made to the stream socket or datagram socket address families. The VMA library bypasses the operating system by implementing the underlying work in user space.

uStream *uStream* [150] is a user-level stream protocol over InfiniBand and eliminates context switches and data copies between kernel and user space. It utilizes threads to implement asynchronous send requests and uses internally pre-registered send and receive buffers. The communication management is split in two communication channels, a data and a control channel. While this mechanism simplifies the data path, it requires extra resources to manage the connections. uStream is complemented by *jStream*, which provides uStream functionality to Java applications.

UNH Extended Sockets Library The *UNH EXS* library is an implementation of the Extended Sockets API (ES-API) for RDMA over Infiniband. The ES-API specification offers a sockets-like interface that defines extensions to the traditional socket API in order to provide asynchronous I/O, but also memory registration for RDMA. UNH EXS offers both stream-oriented and message-oriented sockets. Unlike SDP and VMA, UNH EXS is not designed to run with unmodified sockets applications, which simplifies the design considerably. The library introduces an algorithm that dynamically switches between buffered and zero-copy transfers over RDMA, depending on current conditions. If a send call exceeds the memory window assigned to the receive call, the additional chunks are written to a "hidden" intermediate buffer on the receiver side. To enable true zero-copy, the library implements an advert mechanism that allows direct transfers depending on the receive buffer size.

5.3 Objectives and Strategy

The TCP/IP network stack is the predominant protocol family for network communication, including HPC system networks where system area networks are deployed. As presented in section 5.1, the layered architecture of TCP/IP introduces considerably high overhead and performance bottlenecks, especially for interconnection networks that provide reliable connections. Even though, it is desirable to implement TCP/IP communication means for an interconnect such as Extoll. Transparent and seamless support of legacy code, the Socket API and IP address resolution enables a broad range of applications and services to exploit the benefits of Extoll, including its RDMA and transport offloading capabilities, without any code modifications.

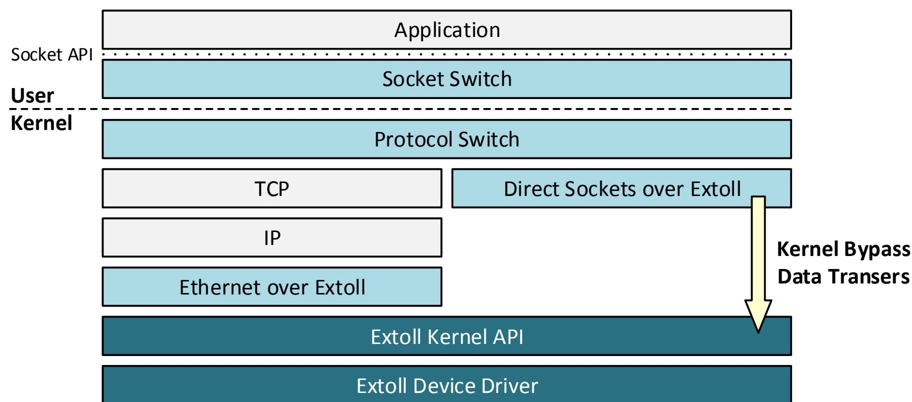


Figure 5.6: Overview of the Extoll software stack with TCP/IP extensions.

For Extoll, a twofold approach is chosen. *Ethernet over Extoll* (EXT-Eth) provides a mechanism for emulating Ethernet communication over Extoll by encapsulating Ethernet frames in Extoll network packets. This way, a fully functional TCP/IP implementation is provided, which can leverage the Linux kernel's support for Ethernet devices while providing IP addressing. The second pillar of the TCP/IP protocol support for Extoll is presented through the specification of the *Direct Sockets over Extoll* (EXT-DS) protocol for stream sockets, which relies on EXT-Eth for connection establishment and address resolution. The purpose of EXT-DS is to provide a transparent, RDMA-accelerated alternative to the TCP protocol by providing kernel bypass data transfers. Figure 5.6 displays an overview of the Extoll software environment with TCP/IP extensions. Recapitulating the findings from previous sections, the tasks of the software stack can be summarized as follows:

- The design and implementation should be transparent to legacy applications, and should provide IP addressing and Ethernet MAC resolution.
- Both EXT-Eth and EXT-DS should leverage the capabilities of the Extoll NIC as good as possible. The RDMA functionality should be utilized to maximize the throughput seamlessly. For EXT-Eth, RDMA can be used to maximize the MTU for the link layer. EXT-DS offers a zero-copy data transmission model.
- The software should be able to transparently switch between EXT-Eth and EXT-DS. For example, a user-level protocol switch could transparently exchange the protocols.
- The design of EXT-DS should maintain the socket semantics. Legacy applications should work out of the box without any code modifications or re-compilation.

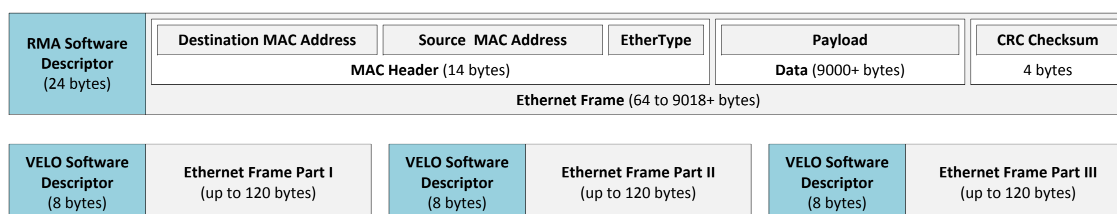


Figure 5.7: Ethernet over Extoll software frame format.

5.4 Transmission of Ethernet Frames over Extoll

The following sections describe the design of EXT-Eth, a method for encapsulating and transmitting Ethernet frames over Extoll. First, the Ethernet frame encapsulation format and transmission protocols are introduced followed by a message matching mechanism and the MTU size. Afterwards, the unicast and broadcast addressing methods are presented. The section concludes with the introduction of the EXN module, which implements the EXT-Eth protocol in form of a network driver.

5.4.1 Link Frame Transmission and Reception

The EXT-Eth protocol adopts transmission mechanisms, which can be compared to the MPI message passing protocols. Depending on the payload size of an Ethernet frame, the design internally switches between the asynchronous eager and rendezvous protocols. The following sections describe the frame format and the communication protocols.

5.4.1.1 Frame Format

An Ethernet frame is the payload of an Ethernet packet, and typically, is transported on an Ethernet link. In general, an Ethernet frame consists of a MAC header, the payload, and a checksum. The MAC header contains the destination and source MAC addresses, while the payload carries the data including any headers for other protocols. The frame ends with a 32-bit cyclic redundancy check (CRC) used to detect any in-transit corruption of data. Depending on the Ethernet frame type, the payload varies in its size. Ethernet Type II frames typically carry between 50 and 1500 bytes, while jumbo frames allow up to 9000 bytes. Super jumbo frames are frames that have a payload size of over 9000 bytes. Figure 5.7 presents the Ethernet over Extoll frame formats from a software perspective. The Extoll software descriptor can either be 8 bytes for VELO transmissions or 24 bytes for RMA transactions. On wire, the Extoll packet is encapsulated in a *start of packet* (SOP) and *end of packet*

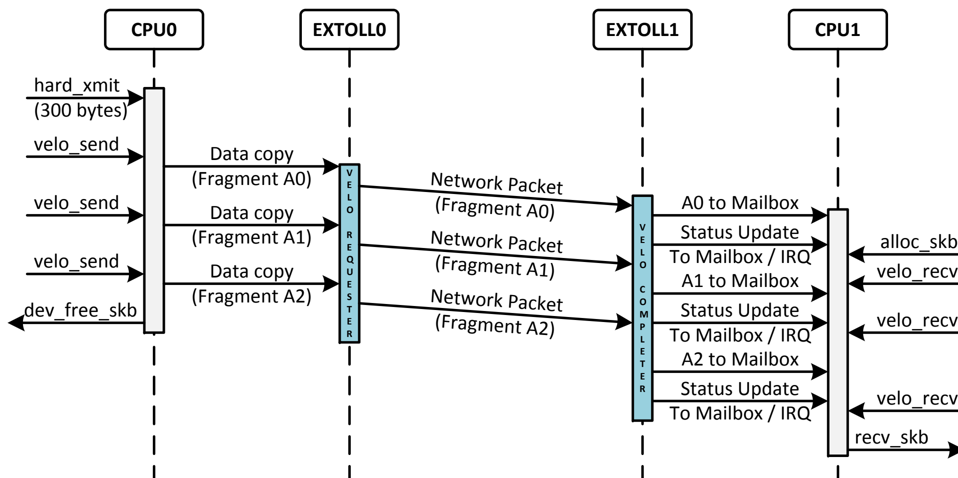


Figure 5.8: Transmission of large messages via eager protocol.

(EOP) frame. The SOP provides routing information and the target VPID, while the EOP carries a CRC and error information.

5.4.1.2 Eager Protocol

In general, the MPI eager protocol is an asynchronous, two-copy protocol that allows a send operation to complete without the acknowledgment from a matching receive call. The sending side makes the assumption that the receiving process can store the incoming message and shifts the responsibility of buffering the message to the receiving side. Typically, this protocol is used for smaller messages (up to KB). The main advantage of the eager protocol is that it reduces the synchronization delay, but it can lead to memory exhaustion and wasted CPU memory cycles for copying data to or from the network into buffer space.

The idea of the eager protocol aligns well with the design of Extoll's VELO unit. The VELO functional unit provides fast two-sided (synchronous and asynchronous) communication for small messages and guarantees in-order delivery of packets. A VELO packet is 128 bytes in size with the first 8 bytes being used for the Extoll software descriptor. For a configurable VELO threshold, Ethernet frames are split in 120 byte-sized fragments and encapsulated in multiple VELO sends. Figure 5.8 presents an example transmission using the eager protocol. On the sender side, the Ethernet frame, here 300 bytes in size, is split in three data fragments A0-A2, and then, encapsulated in three VELO sends. First, the data is copied to the VELO Requester via programmed I/O. Then, the fragments are encapsulated in Extoll network packets and sent to the receiving node. On the receiving side, the VELO Completer copies the network packet payloads to the corresponding VELO mailbox

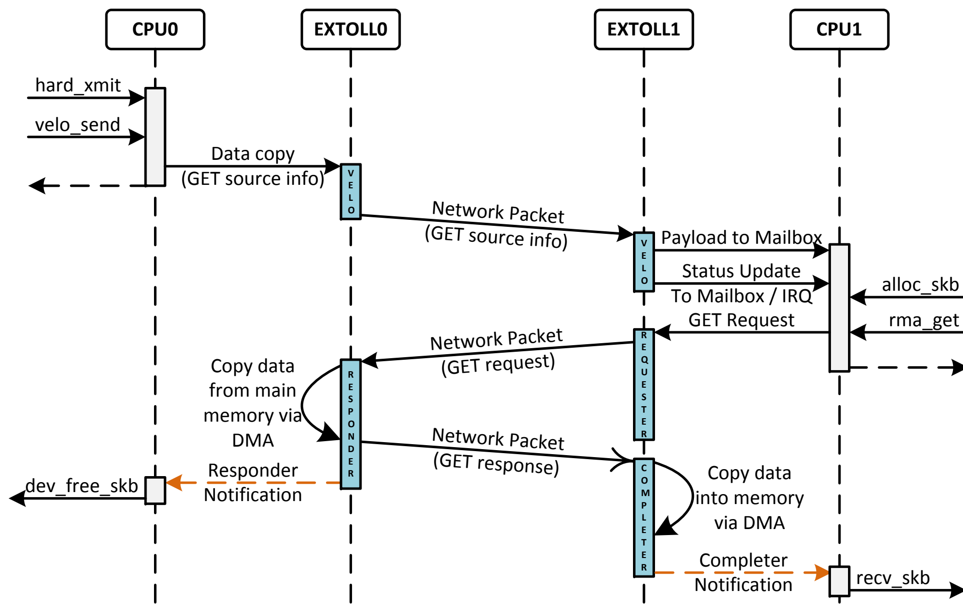


Figure 5.9: Transmission of large messages via rendezvous protocol.

(one mailbox per VPID), which resides in the main memory of the node. After the completer writes the status word to the mailbox, an interrupt is triggered to inform the CPU that new messages have been received.

5.4.1.3 Rendezvous Protocol

The MPI rendezvous protocol is used when assumptions about the receiving process buffer space cannot be made, or when the limits of the eager protocol are exceeded. Before any data can be transmitted, an initial handshake needs to be performed between the sender and the receiver. Typically, the sender process informs the target process about the desired data transmission by sending a message that advertises transmission details such as length and source address. The target process receives this information, and when buffer space is available, it replies to sender that the message can be sent, which in turn, initiates the data transmission. The rendezvous protocol provides better scalability, when compared to the eager protocol, and robustness by preventing memory exhaustion and termination on receive processes. It also enables true zero-copy communication by eliminating data copies.

Figure 5.9 displays the sequence diagram of the rendezvous protocol over Extoll for transmitting Ethernet frames. For large messages, the transfer is advertised through a VELO send, which contains the information needed to initiate an RMA GET transfer from the receiving node. The VELO message provides the length of the Ethernet frame and the physical start address of the payload in the main memory of

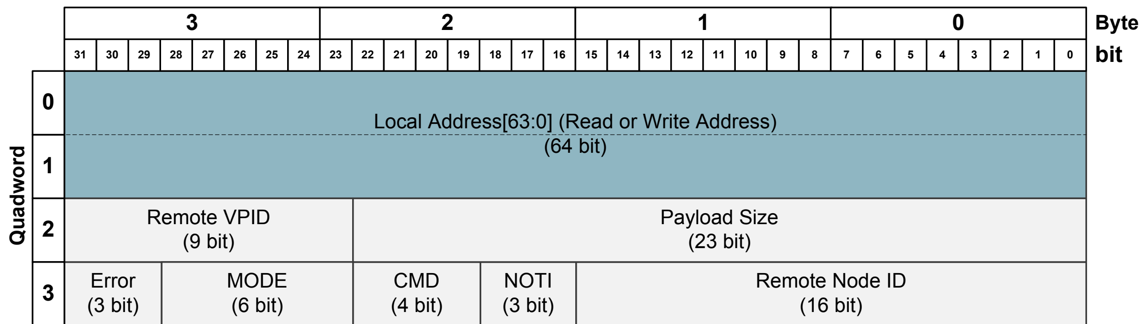


Figure 5.10: RMA standard notification format.

the sender node, and therefore, describes the GET source buffer. Once the target node has processed the VELO message and setup a socket buffer as the GET sink buffer, it initiates the data transmission by issuing an RMA GET request, which places the Ethernet frame directly into the socket buffer on the target node. Upon completion of the RMA GET transmission, the RMA notification mechanism is used to inform sending and receiving side about the completion of the data transmission.

5.4.2 Message Matching and Resource Management

Besides the actual transmission of the Ethernet frames, Extoll needs to maintain a list of outstanding RMA transmissions. On the link layer for every transmission, the NIC driver is handed over a socket buffer (`struct sk_buff`). Upon completion of a transmission, the socket buffer needs to be returned to the kernel, so that it can be re-used for the next data transmission. For transmissions using the VELO unit, socket buffers are returned immediately after the send is performed.

5.4.2.1 RMA Notification Format

In order to match completed GET RMA transactions with outstanding socket buffers on the sender node, the RMA notification mechanism (see section 3.2.1.1) is used to implement an efficient message matching mechanism. Recall from the previous section that RMA transfers are triggered through a rendezvous mechanism, where an RMA GET is initiated on the destination node. Upon completion, the RMA functional unit informs the source and target node processes by issuing a notification on both sites. Figure 5.10 presents the RMA standard notification format. Of particular interest for the message matching is the *local address* entry. This field contains the address that was (or will be) affected by the transaction on the notification receiving node. It always contains the local address from the request or response, which the

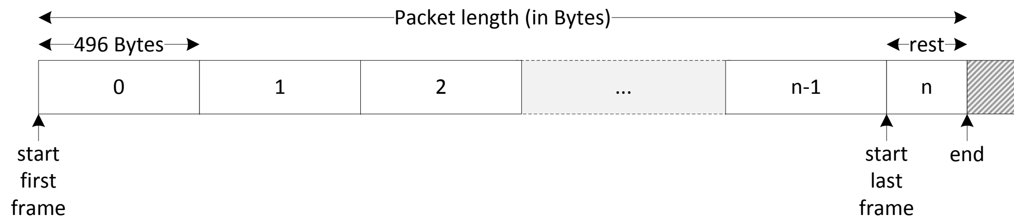


Figure 5.11: Payload layout after RMA MTU fragmentation.

current unit has received. If the received packet contains a virtual address (network logical address) that has to be translated before being used, this field contains the virtual address for security reasons.

Ethernet over Extoll uses byte-sized GET requests for data transmission, where three different notification types can distinguished. If the notification is written by the *RMA Requester*, the local address field contains the write address where the GET data will ultimately be written to. When it is written by the *RMA Responder*, the field contains the read address from where the data was originally read. When the byte-sized GET response arrives at the *RMA Completer* and a notification is written, it contains the write address.

When a single RMA software descriptor has been split into multiple network descriptors at any point, then only the last network descriptor will write a notification. This happens when the payload size of an RMA transaction exceeds 496 bytes, which is known as the RMA MTU. Then, the RMA unit splits the data in multiple RMA packets with up to 496 bytes of payload. In the case that no error occurs along the way, the notification will then contain the address that was used in the last partial network descriptor. For Ethernet over Extoll, two notifications are generated. On the sender node, an RMA responder notification is written while on the destination node, an RMA completer notification is generated.

5.4.2.2 Message Matching Mechanism

On both the sender and target node, the received RMA GET request notification needs to be matched with the corresponding socket buffer, before the socket buffer can either be freed on the origin node or handed over to the next upper TCP/IP layer on the target node. Figure 5.11 displays the packet layout after fragmentation in RMA MTU sized data chunks. Note, depending on the payload size, the last RMA MTU frame can be smaller than 496 bytes, as indicated by the rest in the figure.

In general, a socket buffer is a contiguous memory region, which describes a network buffer. When sending the packet data from a socket buffer on the sender

node or receiving the data into a newly allocated socket buffer on the target node, the physical start address of the packet payload is queried from the kernel and used to initiate the RMA GET on the target node. To match outgoing and incoming socket buffers with their respective RMA notifications, for each transmission a metadata structure is kept with a pointer to the socket buffer, the physical start address of the packet data, and the start address of the last RMA MTU frame of the payload, which is calculated as follows. First, the module operation from Equation 5.1 is used to determine whether the payload length is a multiple of the RMA MTU (*rmtu*).

$$rest = length \bmod rmtu \tag{5.1}$$

Depending on the result of the module operation, two cases can be distinguished. If the length is a multiple of 496 bytes ($rest = 0$), the start address of the last RMA MTU frame can be calculated as described in Equation 5.2 (I). Otherwise, the start address is calculated following Equation 5.2 (II).

$$lastframe = \begin{cases} start + length - rmtu, & \text{(I) if } rest = 0; \\ start + length - rest, & \text{(II) if } 1 \leq rest \leq rmtu - 1. \end{cases} \tag{5.2}$$

When receiving either an RMA responder or completer notification, the local address from the notification is compared to the queue with outstanding socket buffer transmissions. There are separate queues for the send and receive RMA path.

5.4.3 Maximum Transmission Unit

The *Maximum Transmission Unit* (MTU) is the size of the largest *protocol data unit* (PDU) that can be processed in a single network layer transaction. It can be applied to communication protocols and network layers. The maximum frame size that can be transported on the data link layer, e.g., an Ethernet frame, and the MTU are not identical, but related to each other. As a rule of thumb, larger MTUs are associated with reduced overhead, while smaller MTUs can reduce the network delay. In general, the MTU depends on the underlying network technology, and must be adjusted, either manually or automatically, to not exceed these capabilities.

In the context of the Internet Protocol, the MTU describes the maximum size of an IP packet that can be transmitted over a given medium without fragmentation. The size includes the IP header and the packet payload, but excludes headers and protocol information from lower layers such as the link layer. Typically, the link and

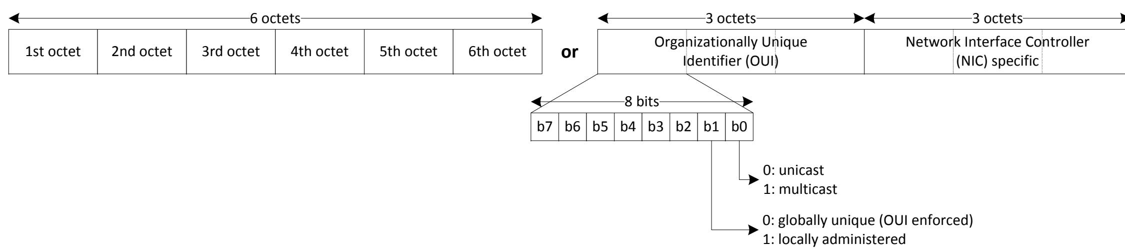


Figure 5.12: MAC address layout universal versus local.

physical layers add overhead to the payload to be transported. Therefore, in order to calculate the physical maximum frame size of a given device, this protocol overhead needs to be subtracted to determine the device's MTU. For example, with Ethernet, the maximum frame size is 1518 bytes, of which 18 bytes are overhead introduced by the header and frame check sequence, resulting in an MTU of 1500 bytes.

For the IPv4 and IPv6 path MTUs, a maximum of 65,536 bytes is allowed. Ethernet jumbo frames support up to 9198 or more bytes. Therefore, the default MTU size for Ethernet over Extoll is 65,536 bytes. IPv6 also allows so called *jumbograms*, which allow an MTU size up to 4 GiB. The current Ethernet over Extoll design supports one RMA transaction per frame, which results in a maximum MTU size of 8 MB.

5.4.4 Address Mapping – Unicast

Unicast, or point-to-point, addressing uses a one-to-one association between the sender and the destination: each destination address uniquely identifies a single receiver endpoint. The MAC address is expected to uniquely identify each node and allows frames to be marked for specific hosts. Thus, it forms the basis of most of the link layer (OSI Layer 2) networking upon which upper layer protocols rely to produce complex, functioning networks.

The following section describes the Extoll Ethernet hardware address format followed by an introduction to the address resolution in IPv4 and IPv6 subnets.

5.4.4.1 Link-Layer Address/Hardware Address

The *Media Access Control address* (MAC address) uniquely identifies a NIC for communication at the data link layer. For most network technologies following the IEEE 802 standard family [151] for local area and metropolitan area networks, MAC addresses are used as network addresses in the medium access protocol sublayer. Together with the logical link control sublayer, they build the data link layer, which provides flow control and multiplexing. Typically, MAC addresses are assigned by

Figure 5.13: Extoll MAC address layout.

the manufacturer and stored in the hardware. They are also referred to as burned-in addresses, Ethernet hardware addresses, or simply hardware or physical addresses. Each NIC must have a unique MAC address.

The original IEEE 802 MAC address format utilizes a 48-bit addressing scheme (6 octets) and addresses up to 2^{48} MAC addresses. Figure 5.12 displays the MAC address layout, which can either be a *universally administered address* (UAA) or a *locally administered address* (LAA). The UAA is uniquely assigned by the manufacturer and can be identified by the first three octets, which are known as the *organizationally unique identifier* (OUI). In case of EXT-Eth, locally administered addresses are used. UAAs and LAAs can be distinguished by the second-least-significant bit of the first octet, which is also referred to as the Universal/Local bit (U/L bit). If the U/L bit is set to 1, the address is locally administered. The least-significant bit of the first octet indicates whether this is a unicast or multicast address.

Figure 5.13 presents the Extoll hardware address layout. The first to fourth octets are randomly chosen from the list of unused MAC addresses. The fifth and the sixth octet are used to encode the Extoll node ID, which is uniquely assigned during the network setup with EMP (see section 3.3.3). A node ID comprises of 16 bits. Therefore, it can by split in the octets. After performing an address resolution either for IPv4 or IPv6, the MAC address associated with an IP address can be used by Extoll to determine the target node. By using two octets or 16 bits to decode the node ID, up to 2^{16} (= 65,536) Extoll nodes can be distinguished, which is the maximum amount of Extoll nodes in one 3D torus. Although intended to be a permanent and globally unique identification, most modern operating systems allow to change the MAC address, which is a necessary feature for network virtualization. For Extoll, the first four octets can be modified at configuration or run time.

5.4.4.2 Address Resolution in IPv4 Subnets

Address resolution in IPv4 subnets is accomplished through the *Address Resolution Protocol* (ARP) [152], which uses a simple message format containing one address resolution request or response. The link and network layer address sizes affect the ARP message size. Since EXT-Eth encapsulates complete Ethernet frames in Extoll network packets, the Ethernet address resolution is performed as for any

Octet offset	0	1	2	3
0	Hardware type		Protocol type	
4	Hardware address length	Protocol address length	Operation	
8	Sender hardware address (bytes 0-3)			
12	Sender hardware address (bytes 4-5)		Sender protocol address (bytes 0-1)	
16	Sender protocol address (bytes 2-3)		Target hardware address (bytes 0-1)	
20	Target hardware address (bytes 2-5)			
24	Target protocol address			

Figure 5.14: Internet Protocol (IPv4) over Ethernet ARP packet.

other Ethernet device. Figure 5.14 displays the structure of a typical IPv4 over Ethernet ARP packet. The *hardware type* field specifies the network link protocol type [153]. For EXT-Eth, the type `ARPHDR_ETHER` is used, which is the standard type for Ethernet devices. For IPv4, the *protocol type* contains `0x0800`. The *hardware length* describes the length of the hardware address. This is of special interest for non-standard formats, e.g., Infiniband hardware addresses. The *sender hardware address* contains the MAC address of the sender. In an ARP reply, the *target hardware address* indicates the address of the host that originated the ARP request.

5.4.4.3 Address Resolution in IPv6 Subnets

In IPv6 subnets, address resolution is accomplished through the *Neighbor Discovery Protocol* (NPD) [154]. NPD defines five *Internet Control Message Protocol for IPv6* (ICMPv6) packet types to perform router solicitation, router advertisement, neighbor solicitation, neighbor advertisement, and network redirects.

The address resolution process can be described as follows. A node requesting the link layer address of a target node multicasts a neighbor solicitation message (ICMPv6 packet type 135) with the target IPv6 address. The target sends back a neighbor advertisement message (ICMPv6 packet type 136) containing its link layer address (MAC address). Neighbor solicitation and advertisement messages are also used for neighbor un-reachability detection, which involves detecting the presence of a target node on a given link. This mechanism can be directly applied to Ethernet over Extoll without any special adjustments or the use of extension options.

5.4.5 Multicast Routing

In its most generic form, *multicast networking* is a type of group communication, where a data transmission is addressed to a group of network nodes simultaneously in a one-to-many or many-to-many fashion. *IP multicast* [155] is the IP-specific

implementation of the multicast networking paradigm. It enables the sending of IP datagrams to a group of interested receivers by using specially reserved multicast address blocks in IPv4 and IPv6. *Broadcasting* is a special case of multicast networking, which distributes a message in a one-to-all manner.

5.4.5.1 Technical Overview

IP multicast is a real-time communication technique for sending IP datagrams in a one-to-many or many-to-many distribution over an IP infrastructure. In general, a packet is sent only once and the network nodes (typically network switches and routers) replicate and forward the packet to reach multiple receivers. In order to send and receive multicast messages, senders and receivers use IP multicast group addresses. While senders use the group address as the IP destination address, receivers use the IP multicast group address to notify the network they are interested in receiving packets from the respective multicast group. Typically, receivers join a group by utilizing the *Internet Group Management Protocol* (IGMP). After joining a group, a multicast distribution tree is constructed.

As explained in the previous section, unicast packets are delivered to a target node by setting a specific Ethernet MAC address. Broadcasts are delivered by using the broadcast MAC address, which is `FF:FF:FF:FF:FF:FF`. For IPv4, IP multicast packets are delivered by using the reserved MAC address range between `01:00:5E:00:00:00` and `01:00:5E:7F:FF:FF`. Note, the multicast bit is set in the first octet of the MAC addresses. In case of IPv6 multicast packets, the Ethernet MAC address is derived by taking the four low-order octets of the IPv6 address and performing a bitwise OR with the MAC address `33:33:00:00:00:00`. For example, `FF02:DEAD:BEEF::1:3` would be mapped to the MAC address `33:33:00:01:00:03`.

5.4.5.2 Extoll Multicast Groups and Routing

The VELO functional unit is currently the only unit that can issue multicast messages from software side. When assembling the software descriptor for a VELO packet, the multicast bit must be set to 1. The target node ID is then interpreted as the multicast group ID. The minimal data granularity of the Extoll network protocol is a *cell*, which is 64 bits in size. As mentioned before, the first cell of an Extoll network packet is the SOP cell. Its payload contains information about the packet, including the multicast bit, routing information (adaptive/deterministic), the traffic class, the node ID (16 bits), the target functional unit, and the VPID. The node ID is split into two segments, which results from dividing the Extoll cluster into N segments,

each M nodes large. Otherwise, the routing tables would need 65,536 entry RAMs, which is too large to handle. Currently, N is set to be 64 and M is set to be 1024. The table for the segments is called *global routing table*, the table for the nodes inside these segments *local routing table*.

The Extoll on-chip network switch (crossbar) connects the functional units through link ports with the network ports of the NIC, and provides hardware support for efficient multicast networking, especially for broadcasts. The routing in the network layer relies on table-based routing. Each crossbar inport has its own global, local, and multicast routing tables, but the entries in these tables are the same over all crossbar inports. The multicast routing table can distinguish up to 64 Extoll multicast groups and provides information about where to forward the multicast packet to. In general, the routing tables are written to the Extoll register file for every crossbar inport during network configuration time, when the EMP initializes the network.

5.4.5.3 IP Multicasts and Broadcasts over Extoll

IP multicast routing requires a network interface to be configured to listen to all link layer IP multicast group addresses. For an Ethernet interface, this is achieved by turning on the promiscuous multicast mode on the interface. When the promiscuous mode is enabled, the MAC filtering is disabled on the interface and all packets received are sent to the CPU regardless of the destination address of the packet.

The Extoll NIC is not an Ethernet NIC and utilizes the Extoll network protocol to transmit packets. Therefore, it does not provide any hardware support for the promiscuous multicast mode. Extoll can only route packets according to their network descriptors and corresponding routing table entries. Consequently, all multicast and broadcast support comprises of the correct configuration of the routing tables on every Extoll NIC and a software layer, which identifies IP multicast group and broadcast addresses and encapsulates them in corresponding VELO packets with the multicast bit set. By default, the EMP configures multicast group ID 63 on every Extoll NIC to broadcast a packet sent to this ID to every node in the Extoll network. On the software side, the software identifies a broadcast based on the matching broadcast Ethernet MAC address and encapsulate the corresponding Ethernet frame in VELO multicast messages. Besides from the broadcast, EXT-Eth currently does not support IP multicast. In theory, it is possible to define multiple Extoll multicast groups, but the software would need to keep track of group members and facilitate the routing tables accordingly.

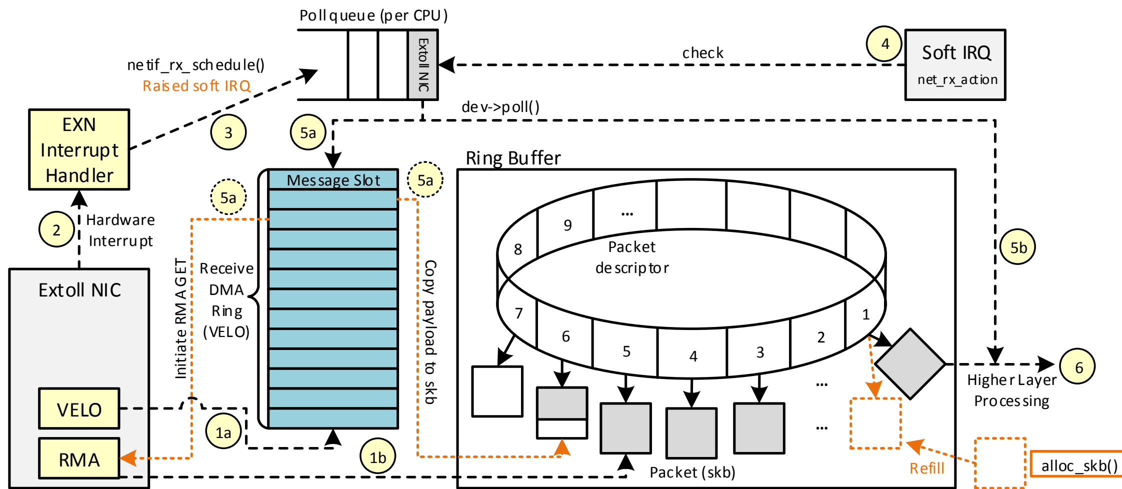


Figure 5.15: Path of an incoming packet in NAPI mode.

5.4.6 EXN: Extoll Network Interface

The network subsystem of the Linux kernel is designed to be completely protocol-independent, which applies to both networking and hardware protocols. The interaction between a network interface driver and the kernel deals with one packet at a time and allows protocol issues to be hidden neatly from the driver, but also hides the physical transmission from the protocol. This section introduces the EXN module, which has been implemented as part of this work. It provides the network interface between the Extoll hardware and the TCP/IP stack.

5.4.6.1 Network Driver Overview

The *Extoll Network* (EXN) interface belongs to the Ethernet class and implements the EXT-Eth protocol as a loadable kernel module. Emulating Ethernet has the benefit that the implementation can take full advantage of the kernel's generalized support for Ethernet devices. The most important tasks performed by a network interface are the data transmission and reception. Whenever the kernel needs to transmit a data packet, it calls the `hard_start_transmit()` method of the driver, which puts the data in the outgoing queue. EXN implements the aforementioned eager and rendezvous protocols for data transmission, and uses a reserved RMA and VELO VPID for process security. For packet reception, EXN supports both the interrupt-driven and the NAPI mode.

Figure 5.15 illustrates the path of an incoming packet for EXN running in NAPI mode. Depending on the transmission protocol, data is received either through the VELO (1a) or RMA (1b) units. VELO writes the incoming packet to the next free

```
1 exn0: flags=67<UP,BROADCAST,RUNNING>  mtu 65536
2 inet 10.2.0.8  netmask 255.255.255.0  broadcast 10.2.0.255
3 inet6 fe80::2c87:4ff:fe07:1  prefixlen 64  scopeid 0x20<link>
4 ether 2e:87:04:07:00:01  txqueuelen 1000  (Ethernet)
5 RX packets 22  bytes 1572 (1.5 KiB)
6 RX errors 0  dropped 0  overruns 0  frame 0
7 TX packets 22  bytes 1572 (1.5 KiB)
8 TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Listing 5.1: Example output of ifconfig for Extoll network interface exn0.

message slot in a receive ring buffer associated with the used VPID. RMA, on the other hand, writes the packet directly to a pre-allocated socket buffer. Upon completion, the Extoll NIC triggers a hardware interrupt (2). Extoll can distinguish different sources of interrupts, e.g., different functional units. Therefore, EXN has two different interrupt handlers, one for VELO and one for RMA interrupts. As the processing in the interrupt context should be as low as possible, `netif_rx_schedule()` puts a reference to the Extoll device into the poll queue (3), which moves the packet processing in the software interrupt context. Then, `net_rx_action()` peeks the first entry of the poll list (4). If there are packets available for reception, the function disables all interrupts and calls the `poll()` method of the driver. In case of EXN, there are two different `poll()` methods registered, one to process VELO interrupts (5a) and one for RMA (5b). Depending on the message tag, a VELO message can either carry the payload of a packet or advertise an RMA GET operation. If it contains a payload fragment, the data is copied to a free socket buffer entry in the ring buffer, and after receiving the complete payload, passed to the upper layers (6). If it advertises an RMA operation, the `poll()` method writes the RMA software descriptor and initiates the GET operation. In case of an RMA interrupt (5b), the packet has already been received in a pre-allocated socket buffer and can be passed to the upper layers (6) for further processing.

Listing 5.1 displays the command line output of the *ifconfig* tool for the EXN interface. It can be seen that EXN utilizes the described MAC address format to encode the node ID information. IP addresses can either be statically or dynamically assigned. For static IP addresses, the interface can be configured with a preassigned IP address through an interface configuration file. Otherwise, the *Dynamic Host Configuration Protocol* (DHCP) can be utilized for assigning IP addresses. By default, the interface supports IP broadcasts; IP multicasting is currently not supported.

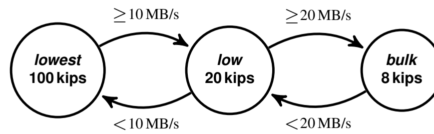


Figure 5.16: Interrupt throttling rate state transitions of the *ixgbe* driver [126].

5.4.6.2 Protocol Thresholds for Efficient Communication

As described in section 5.4.1, EXT-Eth relies on two communication protocols for data transmission at the link layer. To switch seamlessly and efficiently between the two protocols, the EXN module implements two different protocol thresholds.

Eager/Rendezvous Protocol Switch The EXN module internally switches between the eager and rendezvous protocol depending on the size of the payload. For smaller payloads, the eager protocol provides a low latency path to transmit packets through the VELO unit. For large messages, the rendezvous protocol is used. The initial "handshake" is initiated by a VELO message containing the information to setup the the GET sink on the target node. Based on Extoll micro-benchmark results, the threshold for the switch between VELO and RMA data transmission should be between 120 and 480 bytes, which translates to a maximum of four VELO packets for the eager protocol. This way, the module provides a good trade-off between latency performance and bandwidth.

NAPI Budget The Linux kernel uses the interrupt-driven mode by default and only switches to polling mode when the flow of incoming packets exceeds a certain threshold, known as the weight of the network interface. For NAPI-compliant network drivers, the budget module parameter or interrupt throttling rate (ITR) places a limit on the amount of work the driver may do, e.g., interrupts per second. Each received packet counts as one unit of work. The return value of the `poll()` function is the number of packets which were actually processed. If, and only if, the return value is less than the budget, a NAPI driver re-enables interrupts and turns off polling. For the EXN module, there are two NAPI budgets, one for processing incoming VELO packets and one for processing RMA notifications. For both, the default budget value is 64 packets, but it is implemented as a configurable module parameter.

Dynamic ITR As previously describe, NAPI-based packet processing can be configured by the interrupt throttling rate. For the EXN module, this is a static value that only can be changed at module startup time. In order to automatically adapt

the ITR to the current traffic load, Intel's 10 GbE driver *ixgbe* proposes a dynamic ITR [126]. When a poll finishes, the ITR is recalculated. There are three ITR states: lowest, low (initial state), and bulk. Each ITR state is associated with a specific ITR value in thousand interrupts per second (kips) as depicted in Figure 5.16. The current ITR state and the throughput determine the transition to a new ITR state. For instance, if the current load is low, and thus, the throughput is low, the ITR becomes high and vice versa. Future versions of EXN will adopt a similar mechanism.

5.5 Direct Sockets over Extoll

Today's data centers demand that the underlying interconnect technologies provide the utmost bandwidth with extremely low latency. While high bandwidth is important, without low latency bandwidth is not worth much. Moving large amounts of data through a network can be achieved with TCP/IP, but only RDMA can produce the low latency that avoids costly transmission delays.

The *Direct Sockets over Extoll* (EXT-DS) protocol describes an efficient mechanism to utilize the RDMA-enabled Extoll NIC for TCP/IP communication and complements EXT-Eth by providing kernel bypass data transfers for specific TCP point-to-point connections. RDMA allows data to be transferred without passing the data through the CPU and main memory path of TCP/IP Ethernet.

5.5.1 Protocol Overview

In its conceptual design, EXT-DS resembles the Sockets Direct Protocol (SDP) specification [137]. The design of EXT-DS relies on two architectural goals:

- The traditional sockets semantics for `SOCK_STREAM` as commonly implemented over TCP/IP should be maintained. Issues that need to be addressed include the TCP connection teardown, the ability to use TCP ports, IP addressing (IPv4 and IPv6), the connection establishment, and the support of common socket options and flags.
- The support of byte-streaming socket communication over a message passing protocol, including the support of kernel bypass and zero-copy data transfers.

EXT-DS utilizes the RMA and VELO functional units of the Extoll NIC and, depending on the mode, combines VELO send operations with RMA PUTs and GETs. The *Buffered Copy* (BCopy) mode transfers data through intermediate private

buffers and provides a twofold data transmission approach. Similar to EXT-Eth, BCopy switches between an eager transmission mode relying on VELO sends and a rendezvous protocol utilizing RMA PUT operations. The deciding factor is the size of the application buffer to be sent. The *Zero-Copy* (ZCopy) mode transfers data directly between pinned RDMA buffers and uses either RMA PUT (write) or RMA GET (read) operations. EXT-DS has two buffer types:

- *Intermediate kernel buffers* – private buffers used for all synchronization and flow control messages, but also for data transmitted through the BCopy mode.
- *RDMA buffers* – ring buffers used for all RMA transmission. The BCopy mode uses them as intermediate buffers for larger transmissions. The ZCopy mode can pin them to physically contiguous memory for zero-copy transmissions.

5.5.2 Setup and Connection Management

Approximately 15% of the functions provided by the Sockets interface are related to data exchange. One of the most expensive interface calls is the connection setup, but the setup procedure happens only once. For the connection establishment, EXT-DS relies on the fundamental TCP/IP behavior by utilizing EXT-Eth, which creates a traditional connection between two TCP endpoints. The following sections describe the address resolution and connection management, including TCP port mapping.

5.5.2.1 Address Resolution

EXT-DS relies on IP addressing (either with IPv4 or IPv6 addresses) and utilizes the EXT-Eth address resolution mechanism to map an IP address to an Extoll node ID, which is needed to communicate between Extoll nodes. Instead of defining a new methodology, EXT-DS simply passes the IP address to EXT-Eth, which returns the MAC address for a given IP. The node ID is encoded in the MAC address. Thus, the EXT-DS protocol begins after the source destination IP addresses have been resolved during the connection setup.

5.5.2.2 Connection Establishment and Port Mapping

The connection sequence of a characteristic server-client application is displayed in Figure 5.17. TCP is a connection-based protocol and only supports point-to-point connections. Before any data can be exchanged, two socket endpoints need to establish such a point-to-point connection. The basic function to create a socket

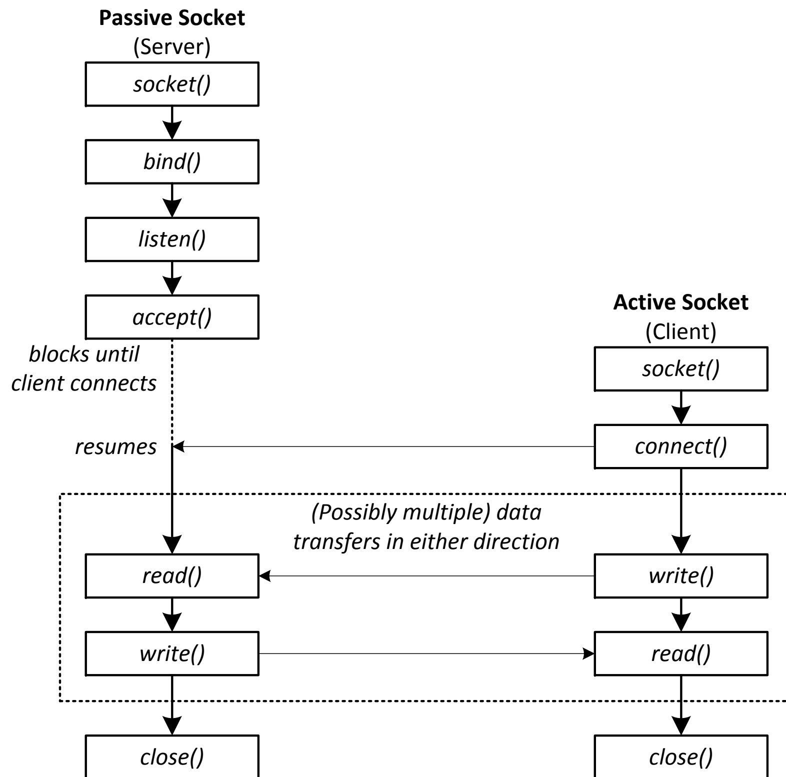


Figure 5.17: Overview of system calls used in stream sockets connection.

handle which can be used with send and receive functions is the `socket()` call. After a handle has been allocated, a connection can be established through the `accept()` and `connect()` calls. When using the EXT-DS protocol, the connect/accept sequence establishes a so called *shadow socket connection* over EXT-Eth, which returns a TCP/IP socket descriptor containing the IP addresses, MAC addresses, and ports used for the connection, and allocates the intermediate kernel and RDMA buffers. In addition, EXT-DS performs an initial handshake between the two Extoll nodes to exchange the RMA receive buffer space addresses. The shadow socket descriptor can be used to interface with the user application.

EXT-DS maps each TCP port to a *virtual device*, which provides an RMA and VELO handle to interface with the functional units. A virtual device basically defines a management structure that is pinned to a specific Extoll virtual process ID. Extoll reserves a user-tunable number of RMA and VELO VPIDs for EXT-DS to provide concurrent sends and receives. The mapping methodology relies on a simple module operation, which provides a static mapping between a port and a virtual device. For example, equation 5.3 performs the modulo operation for 16 VPIDs.

$$\text{virtual device} = (\text{portnr} \ \& \ \texttt{0x000F}) \bmod 16 \tag{5.3}$$

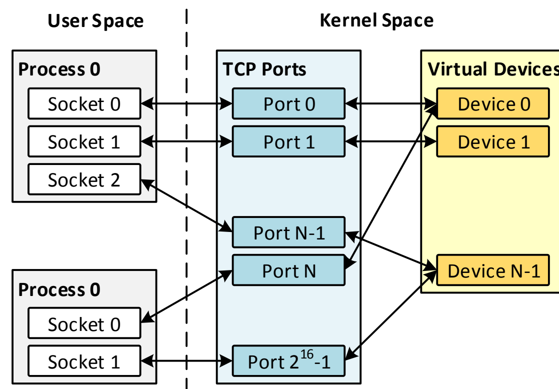


Figure 5.18: Relation between socket handles, ports, and virtualized hardware.

The relation between the socket handles in user space, the TCP port numbers, and the Extoll virtual devices in kernel space is displayed in Figure 5.18. Multiple ports are mapped to the same virtual device. The virtual device also contains a pointer to the corresponding shadow socket descriptor, which can be used to de-multiplex incoming data to the correct TCP port.

5.5.2.3 Connection Teardown

The TCP protocol defines two ways to tear a connection down: (1) a graceful close, where any posted and outstanding data transmission is completed before the connection is closed, and (2) the abortive close, where the connection is immediately terminated. EXT-DS emulates the TCP connection teardown functionality.

Graceful Close Depending on the set socket options, the graceful close, also known as half-closed connections, can describe two types of behavior:

(1) Graceful shutdown with delaying – delays return until all queued messages have been successfully sent or the linger timeout has been reached.

(2) Graceful shutdown with immediate return – immediately returns, allowing the shutdown sequence to complete in the background.

When the shutdown sequence is initiated by calling `close()`, EXT-DS needs to check whether the socket option `SO_LINGER` is set for the socket. If the option is set and the socket has outstanding data transmissions, then `close()` shall block for up to the current linger interval or until all data is transmitted. In addition, a VELO message with the user tag `SHUTDOWN` (refer to section 5.5.3.1) is triggered to initiate the shutdown on both sides of the connection.

Table 5.3: Overview of VELO user tags.

Type	Description
`VELO`	The user tag `VELO` indicates that the payload of the incoming VELO message carries a fragment of user data, which needs to be copied into the application buffer space.
`VELO_LAST`	The user tag `VELO_LAST` indicates that the payload of the incoming VELO message carries a fragment of user data, but also notifies the receiver that the complete buffer has been transmitted and can be passed to the application.
`RMA_INFO`	This user tag indicates that the next user buffer will be transferred either through a RMA PUT or RMA GET operation. The tag ensures message ordering between VELO and RMA transfers.
`RMA_AVAIL`	The `RMA_AVAIL` tag is used to notify the sender that a user application has read data from the RMA sink buffer. The VELO message also provides the amount of data (in bytes) that has been freed.
`SHUTDOWN`	The `SHUTDOWN` user tag indicates that one side has triggered the TCP shutdown sequence.

Abortive Close The abortive shutdown sequence returns immediately for the `close()` call. If the EXT-DS protocol is violated, e.g., an Extoll error occurs, the connection is abortively closed and all outstanding transmissions are dropped.

5.5.3 Data Transfer Mechanisms

The EXT-DS protocol introduces three different transfer mechanisms:

- *Buffered Copy* – transfers application data from private send into private receive buffers. This mode requires an additional data copy from / to user space.
- *Zero-Copy Read* – transfers application data directly through RMA GET operation. This mode requires the application buffers to mapped via Extoll.
- *Zero-Copy Write* –transfers application data directly through RMA PUT operation. This mode requires the application buffers to mapped via Extoll.

In case of an error or insufficient memory space, a fall back mechanism over EXT-Eth is provided. The following sections present an overview of available message types and describe the different transfer modes.

5.5.3.1 Message Types and Flow Control Orchestration

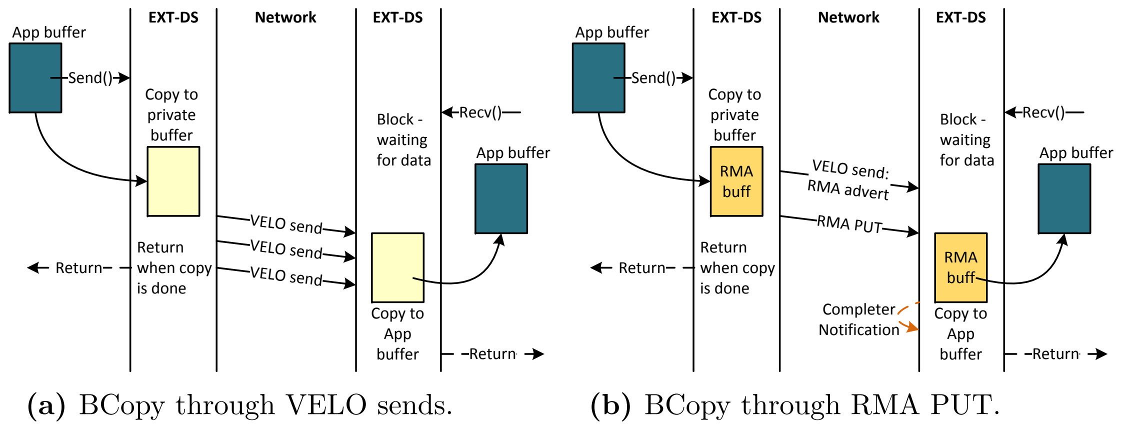
EXT-DS utilizes the VELO and RMA functional units for data transmission. TCP is a protocol that provides connection-based, reliable data exchange. Therefore, an important issue is the synchronization of the VELO and RMA data streams. EXT-DS describes five different data transfer and flow control message types. All of them are transmitted using VELO sends and can be distinguished by their VELO-specific user tags in the software descriptor. Table 5.3 presents an overview of the VELO message user tags defined by EXT-DS.

In order to establish a flow control between incoming VELO and RMA transmissions, the VELO messages are organized in a FIFO-like structure. Extoll guarantees the ordering of VELO messages. Thus in combination with the different message types, they can be used to enforce the ordering of data fragments.

5.5.3.2 Buffered Copy Mode

Per established connection, the BCopy mode uses dedicated, pre-registered private EXT-DS buffers. Typically, application data is copied from the user space into an EXT-DS buffer, and then, transferred over the network. There are two different types of private buffers, intermediate kernel buffers for VELO sends and receives, which are organized in a FIFO-like structure, and RMA buffers for rendezvous PUT operations. Depending on the size of the application buffer and a user-tunable threshold, the BCopy mode switches between a VELO-based eager protocol similar to the protocol specified by EXT-Eth (refer to section 5.4.1.2) and a rendezvous PUT protocol. Figure 5.19 displays the two data transmission protocols.

The sender-side RMA buffer can be seen as a mirrored version of the receiver-side RMA buffer. When the sender buffer is full, the protocol provides a fall back to transmit data using multiple VELO sends until a message of type `RMA_AVAIL` is received. The flow control messages advertise how much RMA buffer has been freed on the receiver side and re-enable the RMA path. Figure 5.20 presents a BCopy rendezvous PUT transfer sequence including the RMA buffer on sender and receiver side with corresponding pointer updates. The physical address of the free receiver-side RMA space can be calculated by adding the write pointer offset to the remote start address.



(a) BCopy through VELO sends. **(b)** BCopy through RMA PUT.

Figure 5.19: BCopy transmission flow.

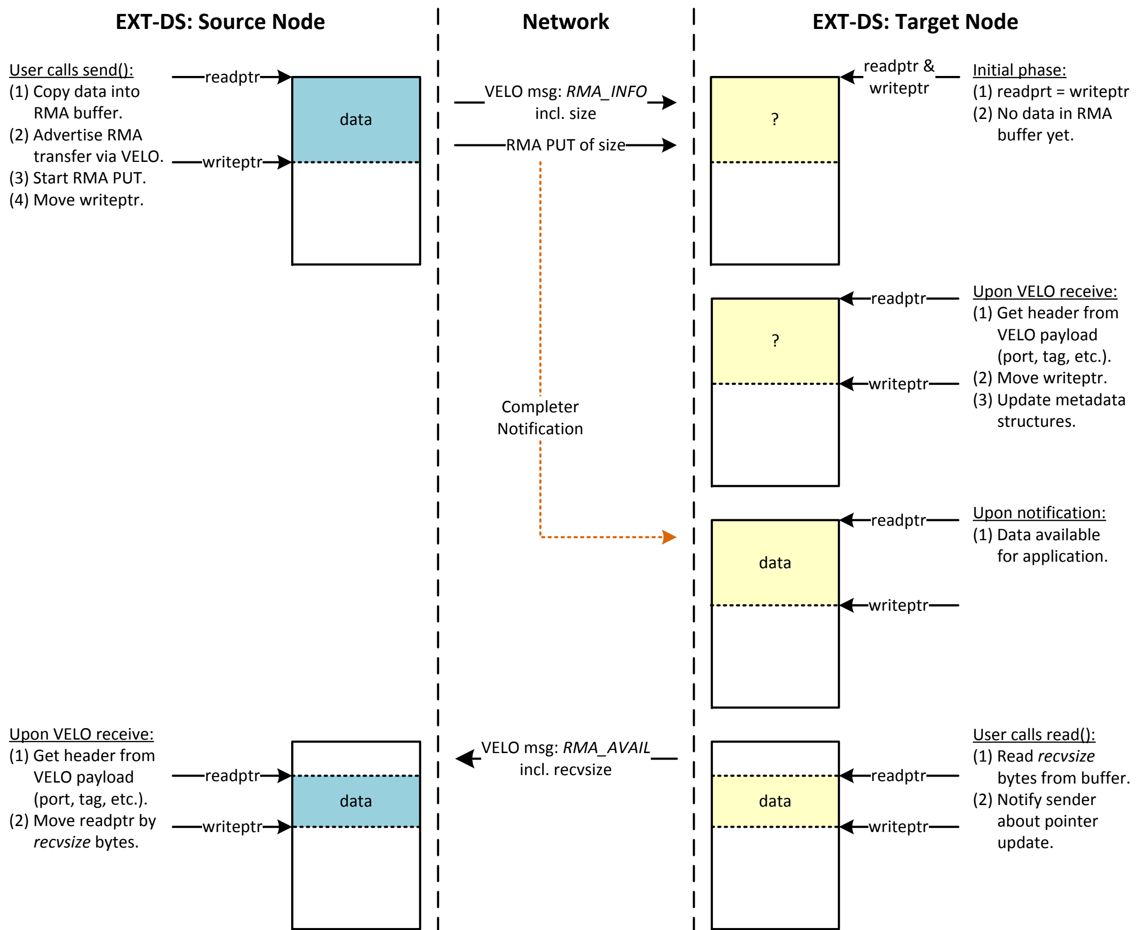


Figure 5.20: RMA transfer sequence and buffer copies for BCopy mode.

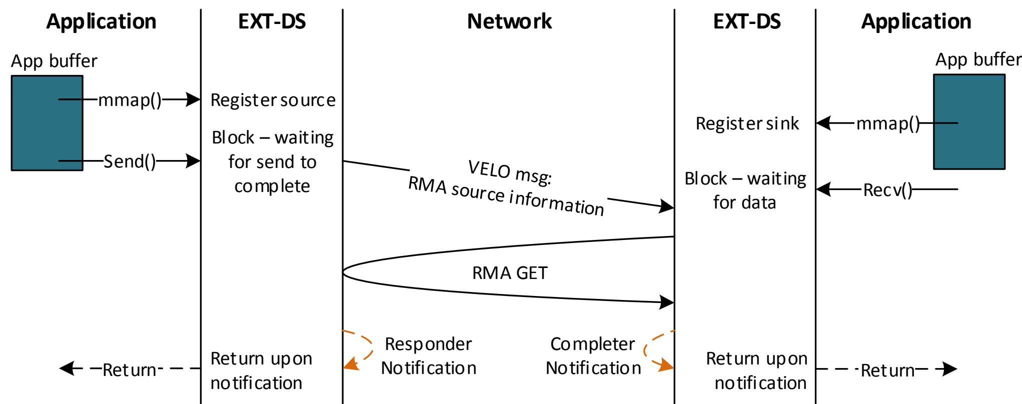


Figure 5.21: ZCopy read transmission flow.

5.5.3.3 Zero-Copy Mode

In general, the ZCopy mode enables a direct data transfer path between application buffers through RMA operations and avoids unnecessary data copies. However, the bypassing technique requires that user buffers are pinned and registered. The Sockets API expects that application buffers are available immediately after returning from the send operation. Therefore, the ZCopy mode can only be utilized for blocking synchronous socket calls. Non-blocking synchronous calls, specified through the `MSG_DONTWAIT` flag or the `O_NONBLOCK` socket option, always rely on the BCopy mode to preserve their socket semantics. There are two possible techniques to preform zero-copy communication: RMA GET and RMA PUT.

ZCopy Read relies on Extoll's RMA GET operation to perform the zero-copy data transfer. The transmission flow is displayed in Figure 5.21. The application buffer needs to be registered with Extoll on both sides, e.g., through an `mmap()` call. Then, the sender side advertises the data source by sending a VELO message. The send call blocks until the data transmission is completed. Upon receipt of the VELO message, the sender side performs an RMA GET operation to copy the data into the data sink. Once the RMA operation completes, notifications are written on both sides, and the send and receive calls return. Similarly, ZCopy Write utilizes an RMA PUT for the data transfer. The data sink needs to advertise the availability of its buffer by sending a VELO messages with the buffer information. The data source then initiates the data transfer by performing an RMA PUT. Upon completion, RMA notifications are written on both sides an the send and receive calls return.

As an alternative to `mmap()`, kernel release 4.14 introduces a new socket option, which enables zero-copy data transmission for TCP sockets [156]. First, the socket option `SOCK_ZEROCOPY` must be set through `setsockopt()`. Then, the zero-copy

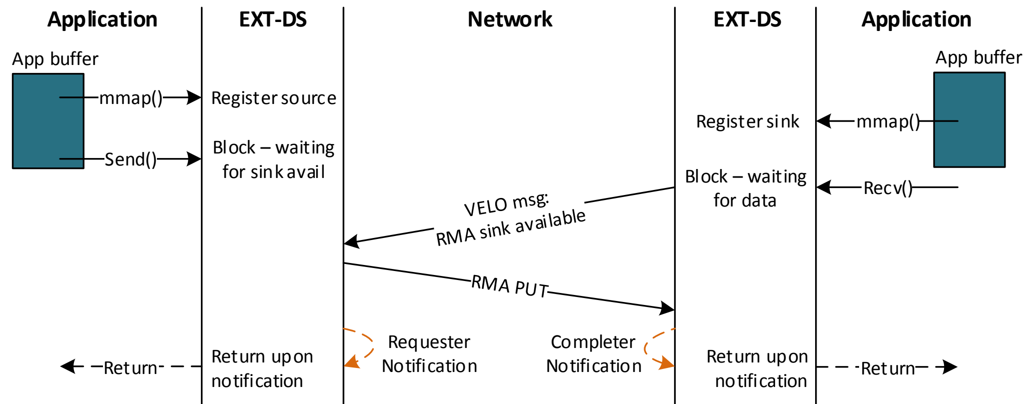


Figure 5.22: ZCopy write transmission flow.

data transfer can be performed by calling `send()` with the flag `MSG_ZEROCOPY`. The transmission pins the application buffer into the man memory and starts the data transfer. Note when the `send()` call returns, the transmission is not complete yet. Therefore, the application needs to periodically check the error queue for new notifications to determine whether the transmission has completed successfully.

5.5.4 AF_EXTL: A Prototype Implementation of EXT-DS

This section provides an overview of the design decisions and current status of the EXT-DS implementation. At first, a discussion about a user versus kernel space implementation is presented followed by an introduction of portability mechanisms to seamlessly utilize the protocol with legacy applications. Afterwards, a brief overview of the implementation status is provided. The section concludes with a description of the de-multiplexing mechanism for incoming messages.

5.5.4.1 Establishing Full Semantics and Process Management

When designing a direct sockets implementation, it is vital to be fully functional against the TCP/IP reference model. In general, two different implementation paradigms can be distinguished: user space and kernel space. While a user space implementation offers the possibility to implement a true operating system bypass, it comes with some limitations and overhead to provide full socket semantics.

Traditional TCP/IP implementations rely on system calls to exchange messages. In general, the operating system is responsible for data transfers, and also, is able to distinguish data transmission from other API calls. Typically, a socket describes an interface between the user application and the operating systems. Starting

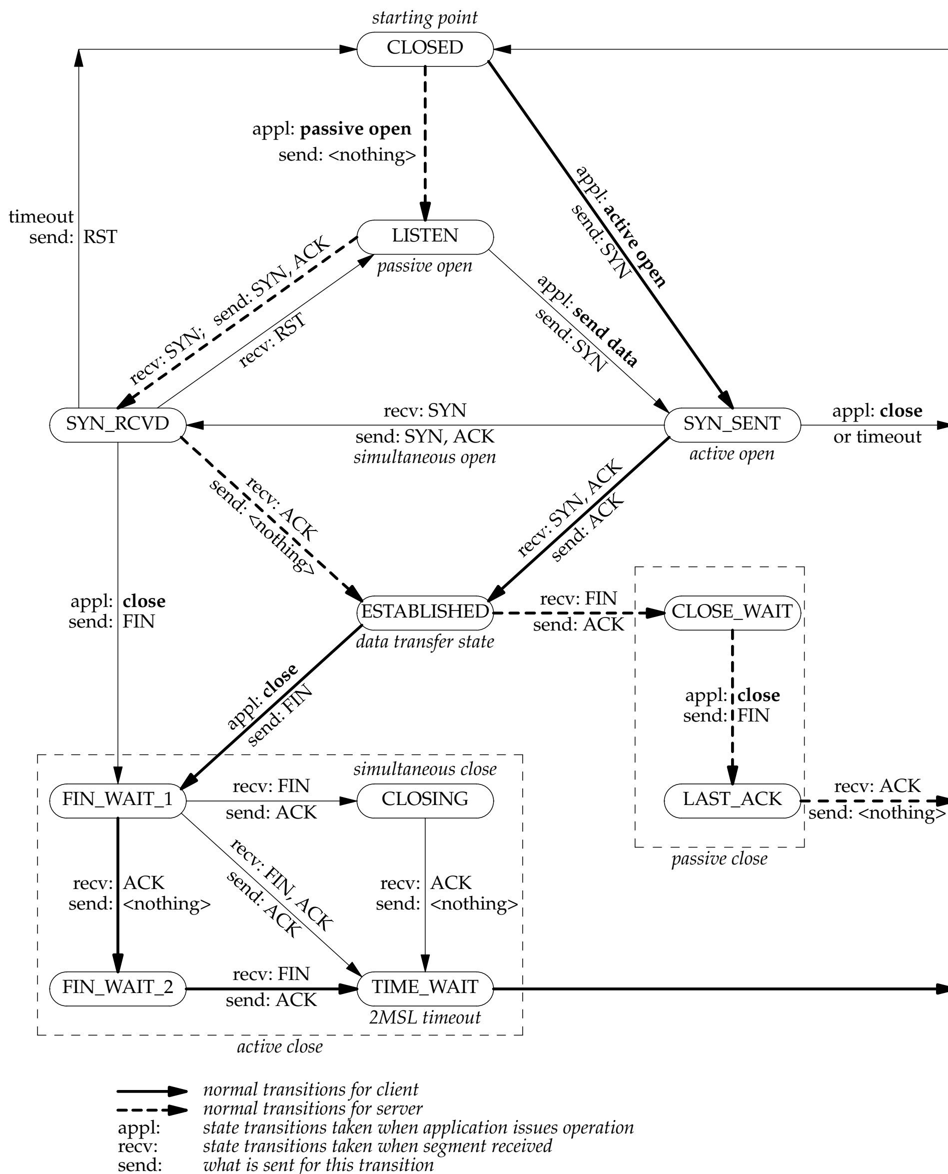


Figure 5.23: TCP state transition diagram [157].

with its creation through the function call `socket()`, a socket can have multiple different states during its lifetime, as depicted in Figure 5.23. The direct sockets implementation has to keep these states in mind. User-level implementations do not have access to the states and would require the software to keep a copy in user space.

Another problem that arises with the user space solution is the handling of unexpected process termination, e.g., through a user initiated ctrl-C or a segmentation fault in the application. For a kernel-level implementation, the operating system serves as a central instance of control, while for a user-level implementation additional functionality would have to be implemented to achieve the same behavior.

Single point-to-point connections are considered as standard situations. Function calls such as `fork()` introduce a new level of complexity. For instance, `fork()` creates a child process by duplicating the parent process. The support of fork is crucial since many services make heavy use of it. For a user-level implementation, a dispatching instance such as the operating system is outside of the critical path. When calling `fork()`, all file descriptors including the socket descriptor of a process are shared between two or more processes. In user space, this poses a problem for the data receiving path and would require additional control tokens.

In summary, it is possible to design a user-level implementation, but it requires additional code to handle exceptions and functions such as `fork()`. To ensure a broad applicability, the implementation of EXT-DS relies on the introduction of a new sockets address family called `AF_EXTL` and a shared user library, which provides a transparent switching functionality between standard TCP sockets and EXT-DS.

5.5.4.2 Direct Sockets Portability and Adaptability

The Sockets interface is one of the most widely used APIs for network communication. Besides the explicit source code modification to use the `AF_EXTL` address family instead of `AF_INET`, it is of importance to provide portability mechanisms, which allow legacy applications to seamlessly utilize the direct sockets implementation. By providing a user library that utilizes library interposition, it is possible to intercept Sockets API calls and automatically switch between the different protocols.

Library interposition [158], also known as function interposition, is a powerful linking technique that allows programmers to intercept calls to arbitrary library functions. Linking can be described as the process of collecting and combining various pieces of code and data into a single binary file that can be loaded into memory and executed. Interposition can occur at different times:

- *Compile time* – when the source code is compiled.
- *Link time* – when the relocatable object files are statically linked to form an executable object.
- *Load/run time* – when an executable object file is loaded into memory, dynamically linked, and then executed.

The following discusses possible automatic address family conversion mechanisms through function interposition at link and at run time and introduces the concepts of static and dynamic linking.

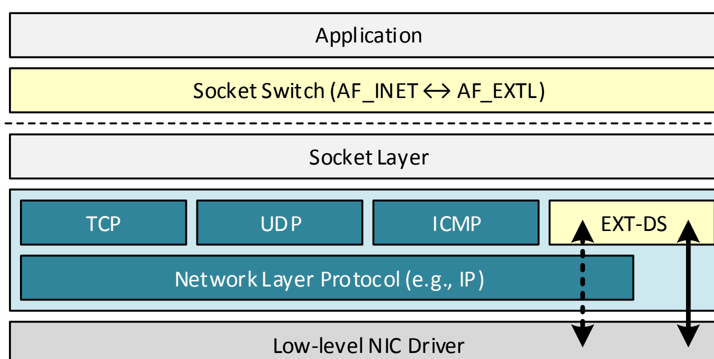


Figure 5.24: Overview of the socket software stack.

Automatic Conversion at Link Time Static applications are statically linked, which means that all library routines are copied into the executable object by the linker. This technique may result in a larger binary file, but is both faster and more portable. For statically linked applications, the `LD_PRELOAD` environment variable [159] has no effect. An alternative is the *wrap* command line switch of the GNU linker [160]. When specified for a given symbol with `--wrap=symbol`, a wrapper function is called instead of symbol. Any undefined reference to symbol will be resolved to `__wrap_symbol`, while any undefined reference to `__real_symbol` will be resolved to `symbol`. For an implementation of EXT-DS, this would mean that for every supported Socket API call a wrapper function must be passed to the linker.

Automatic Conversion at Run Time A much easier approach is offered through shared libraries. When an application is dynamically compiled against shared libraries, a list of undefined symbols is included in the application's binary, along with a list of libraries the code is linked with. The `-fPIC` compile flag generates position-independent code (PIC), which is suitable for use in shared libraries. There is no correspondence between the symbols and the libraries; the two lists just tell the loader which libraries to load and which symbols need to be resolved. At runtime, each symbol is resolved using the first library that provides it. For dynamic applications, this means that function interposition can occur at run-time, and also, that applications do not need to be re-compiled against the interposition library. The intercepting library can be preloaded via the `LD_PRELOAD` environment variable, which contains the path to the pre-loadable library. The environment variable indicates that the user-specified shared object should be prioritized over all others libraries when resolving any symbols.

Algorithm 1 Sockets Switch

```
1:  procedure Socket Switch (Protocol Family, Socket Type, Protocol)
2:
3:      if (domain == AF_INET) && (type == SOCK_STREAM) then
4:          domain ← AF_EXTL;                          ▷ Forward to EXT-DS
5:      else if (domain == AF_INET) && (type == SOCK_DGRAM) then
6:          ___real_socket(AF_INET, type, protocol);
7:      end if
8:
9:      if (___real_socket(domain, type, protocol) == -1) then
10:         ___real_socket(AF_INET, type, protocol);    ▷ Fallback to EXT-Eth
11:     end if
12:
13: end procedure
```

5.5.4.3 Overview and Status of the AF_EXTL Module

Within the scope of this work, EXT-DS is implemented as a kernel module and registered with the kernel as a new transport layer protocol family called `AF_EXTL` (illustrated in Figure 5.24). It can be used to create a communication endpoint for sockets of type `SOCK_STREAM`, and is built on top of the Extoll software environment, especially the kernel API. In addition, a socket switch is provided through a preloadable, user-level shared library. The library seamlessly intercepts the endpoint creation call `socket()` and determines whether a standard TCP (`AF_INET`) or an EXT-DS (`AF_EXTL`) socket should be created, as shown in Algorithm 1.

The shadow socket mechanism is implemented by establishing the connection through the EXN interface, which means that standard TCP protocol function pointers are used for connection establishment. The dashed line in Figure 5.24 indicates this path. The current implementation of the EXT-DS module only supports the BCopy mode, which preserves socket semantics for legacy applications, and utilizes 16 VELO and 16 RMA VPIDs for the virtual device management. Since the entire implementation is in kernel space, it does not leverage Extoll's kernel bypass capabilities, but the transport is offloaded to the NIC while establishing full semantics. It implements all socket entry points that can be invoked by the kernel, including `bind()`, `release()`, and `connect()`.

The EXT-DS implementation has a very resource conservative approach and only allocates and maps `AF_EXTL` objects when an application is requesting a connection. In case of an error or insufficient resources, the module provides a fallback mechanism through the EXN module. The de-multiplexing of incoming RMA and VELO packets

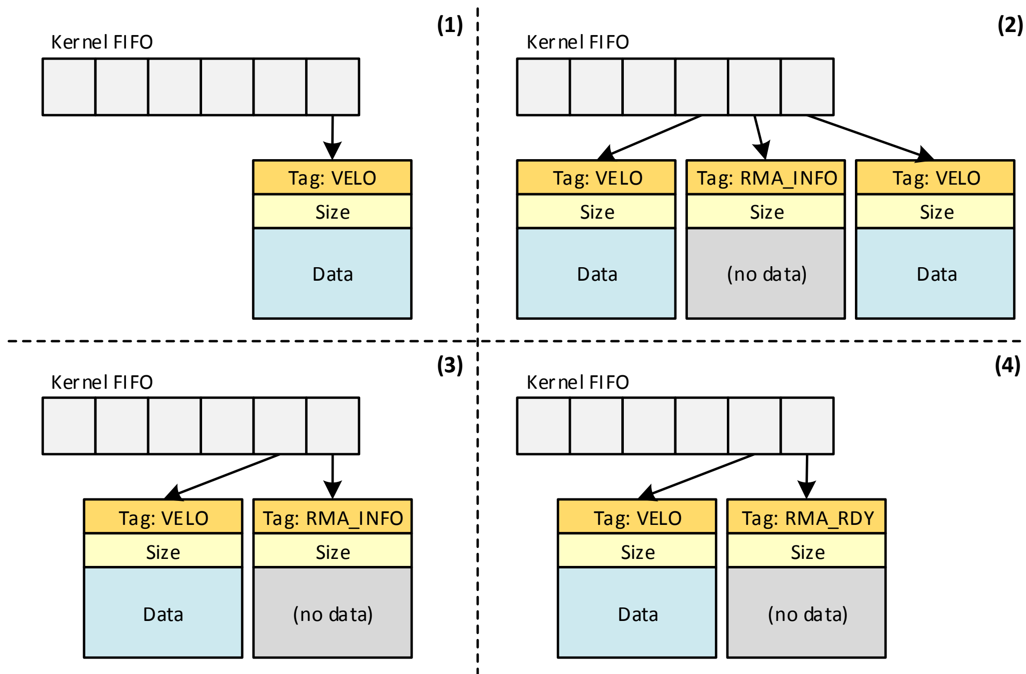


Figure 5.25: Overview of the Kernel FIFO usage with different message types.

to the ports is handled through a kernel FIFO structure, which is presented in the next section. At the time of this writing, only the VELO path is fully functional and used as a proof of concept.

5.5.4.4 De-Multiplexing of Incoming Messages

Each virtual device relates to one RMA VPID and one VELO VPID, and is shared among multiple port numbers. VELO and RMA notifications arrive at a virtual device and need to be forwarded to the right port structure. The de-multiplexing of incoming messages is handled by a kernel thread, which implements a progress function that snoops on the mailboxes for a given VPID for VELO and RMA. In order to avoid an interrupt-driven mechanism, the progress function either is called whenever a socket system call is triggered by the user or when the module internal timer expires. To establish a flow control for the RMA buffer resources, VELO messages are utilized and can be distinguished by their user tag, as described in section 5.5.3.1. The progress function retrieves the port number from the payload of a VELO message, and then, enqueues it to the corresponding kernel FIFO. Figure 5.25 displays different scenarios:

- When the kernel thread is running, incoming VELO messages are enqueued in the kernel FIFO. They can either indicate that (1) the VELO message has

payload attached, or (2) state that the next chunk of the payload is to be received through an RMA PUT operation (RMA_INFO).

- When calling the receive function, the user can only read messages up to the RMA_INFO entry (3). Then, the user gets blocked until new data is available.
- When an RMA PUT completes, a notification is written to the corresponding mailbox. The progress function retrieves the notification, matches it with the corresponding message in the kernel FIFO and sets the user tag to RMA_RDY (4), which indicates the data can be read.

5.6 Performance Analysis

In this section, a performance evaluation of the EXN module and the direct sockets address family AF_EXTL is presented. The section starts with an introduction to different TCP/IP tuning techniques for Linux operating systems deploying 40G or 100G network cards. Afterwards, several microbenchmarks are utilized to evaluate the bandwidth and latency performance followed by a brief discussion about the host CPU utilization. The section concludes with an evaluation of the MPI performance over EXN. Note that at the time of this writing, AF_EXTL is still in an early prototype stage and can only be used for simple latency benchmarks.

5.6.1 TCP/IP Configuration Tuning in Linux Systems

Most Linux operating systems are configured to provide optimal performance with Gigabit Ethernet NICs. This configuration can drastically limit the performance of 10G, 40G and 100G network cards. The Linux kernel offers a vast majority of user-tunable parameters and system configurations. The following system settings have the biggest impact on the TCP performance: CPU governor, TCP host kernel parameters, and application and interrupt binding.

5.6.1.1 TCP Host Kernel Parameters

By default, the Linux network stack is not configured for high-speed large file transfers across SAN links. The main reason for this is to save memory resources and power. Therefore, the default size of the TCP buffers are way too small. The same applies to the default size of the send and receive socket buffers. Another important kernel parameter set configures the TCP receive window scaling and congestion control.

```
1 # add to /etc/sysctl.conf
2 # allow testing with 2GB buffers for send/receive socket buffers
3 net.core.rmem_max = 2147483647
4 net.core.wmem_max = 2147483647
5 # allow auto-scaling up to 2GB buffers for TCP buffers
6 net.ipv4.tcp_rmem = 4096 87380 2147483647
7 net.ipv4.tcp_wmem = 4096 65536 2147483647
8 # receive window scaling
9 net.ipv4.tcp_no_metrics_save = 0
10 net.ipv4.tcp_window_scaling = 1
11 # congestion control
12 net.ipv4.tcp_congestion_control = htcp
13 net.ipv4.tcp_timestamps = 0
```

Listing 5.2: TCP host kernel parameters configuration via sysctl.

Listing 5.2 displays the recommended `sysctl` settings for 40G and 100G NICs, including setting the default buffer sizes to the maximum of 2 GB.

5.6.1.2 CPU Governor

The CPU governor defines the power characteristics of the system CPUs, which in turn affects the overall CPU performance. CPU frequency scaling allows the operating system to scale the CPU frequency up and down in order to save power. The default setting for most modern operating systems is to run the CPU in its power saving mode, which can heavenly impact the performance of high-speed network technologies. Depending on the Linux distribution, there are different command line tools, which allow the user to adjust the default clock speed of the processor on the fly. For CentOS, the governor can be changed to the performance mode by running the tool *cpupower* with the following command line:

```
$ cpupower frequency-set -g performance
```

5.6.1.3 Interrupt and Application Binding

Today's system architectures typically comprise of NUMA nodes. To fully maximize the single stream performance for both TCP and UDP on such architectures (i.e., Intel Sandy and Ivy Bridge), it is essential to observe which CPU socket is being used. In general, it is advised to bind both applications and interrupts to the CPU socket that is closest to the PCIe slot the NIC connected to. This can be achieved by disabling the interrupt balance option in the Linux, e.g., by modifying the boot options, and then, bind the NIC's interrupts to a specific CPU socket.

5.6.2 Test System

The test setup comprises of two systems named after the codename of their CPU architecture: *Haswell* and *Skylake*. The Haswell system comprises of two nodes each equipped with an Intel Xeon E5-2640 v3 dual socket CPU (8 cores per socket) running at 2.6 GHz and 32 GB of main memory. The Haswell system has two on-board Intel 10-Gigabit X540-AT2 Ethernet controllers. One provides Gigabit Ethernet (GigE), the other one 10 Gigabit Ethernet (10GigE). The Skylake system comprises of three nodes. Each node has an Intel Xeon Silver 4110 CPU (one socket with 8 cores) running at 2.10 GHz and 32 GB of main memory.

For both systems, all system nodes are equipped with an Extoll Tourmalet 100G NIC and run CentOS Linux version 7.3.1611 with kernel release 3.10.0-514.26.2.el7. For the Extoll NICs, the software environment version 1.4 is installed on every node.

5.6.3 Microbenchmark Evaluation

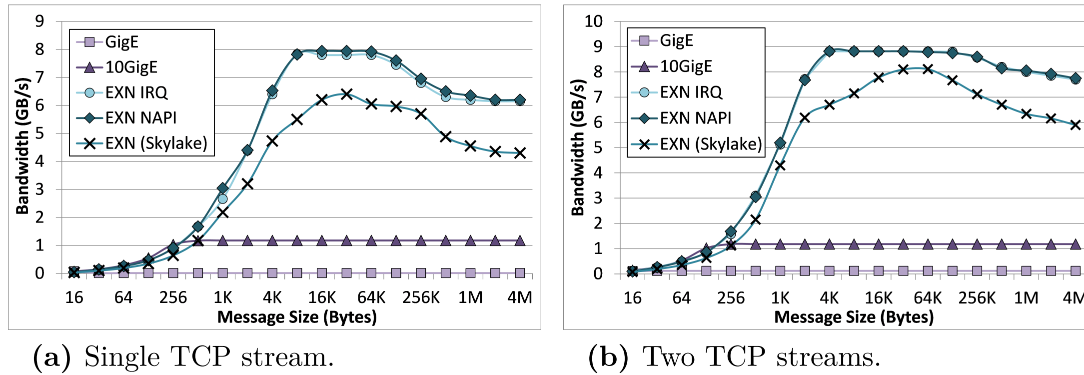
The following three microbenchmarks have been used for the performance evaluation:

netperf The *netperf* benchmark [161] can be used to measure various aspects of networking performance with a particular focus on bulk data transfers and and request/response time. The benchmark is available for TCP and UDP connections, and relies on the Berkeley Sockets interface. Originally designed by Hewlett Packard, it has become the de-facto standard for the bandwidth and latency evaluation of interconnect technologies.

TCP-PingPong The *TCP-PingPong* benchmark provides a simple benchmarking tool to measure the half round trip latency for TCP and UDP socket connections with message sizes ranging between 1 and 2048 bytes.

SockPerf The *SockPerf* tool [162] is an open-source network benchmarking utility provided by Mellanox Technologies. It is designed for testing the performance of high-performance systems, with a focus on throughput and latency. SockPerf utilizes most of the Sockets API calls and options. It is used to verify benchmarking results obtained through netperf.

In addition to the microbenchmarks, *vmstat* and *numactl* are utilized. numactl can be used to control the NUMA policy for processes or shared memory. Examples are NUMA-aware socket binding and memory placement policies. vmstat can be utilized to monitor system resources such as main memory and CPU utilization.



(a) Single TCP stream. **(b)** Two TCP streams.

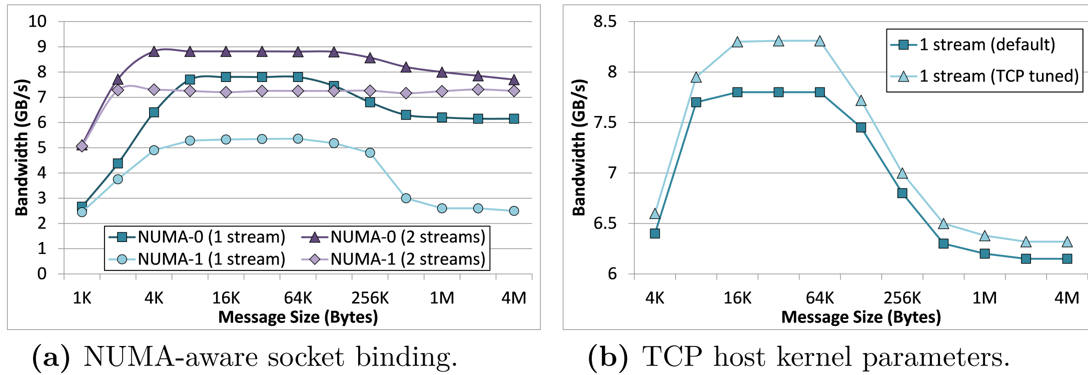
Figure 5.26: Netperf bandwidth performance for TCP streaming connections.

Each benchmark is run for 10 times. The results in the graphs are calculated as the arithmetical average of the runs. As mentioned before, all benchmark results are verified through SockPerf.

5.6.3.1 Bandwidth Results

The data in Figure 5.26 presents the bandwidth results measured with netperf for TCP streaming connections on the Haswell and Skylake systems. The bandwidth results include the GigE, 10GigE, and EXN over Extoll performance from the Haswell system and compare it to the performance of the EXN module on the Skylake system. The Haswell systems comes with a dual socket CPU, which means that NUMA phenomena can occur. For these bandwidth tests, netperf has been bound to the socket that is closest to the Extoll NIC. For the Haswell system, this is NUMA node 0. For EXN, the interrupt-driven (EXN IRQ) and the polling mode (EXN NAPI) are compared. On the Skylake system, EXN is run in the NAPI mode.

Figure 5.26a displays the results for a single TCP stream connection between two nodes. As expected, the performance of EXN IRQ and EXN NAPI is similar. The maximum performance of about 7.9 GB/s is achieved for packet sizes in between 8 KB to 64 KB, which correlates with the EXN MTU size of 64 KB. A single stream cannot fully utilize the theoretical peak performance of Extoll. This limitation is imposed by the general concept of TCP/IP, which includes memory copies, packet fragmentation, and protocol overhead. Surprisingly though, the Skylake chipset shows severe performance degradation compared to the older Haswell CPU. One plausible explanation is the latency inflected by the PAUSE instruction of the Skylake microarchitecture [163, 164]. When using a spinlock in kernel space to ensure mutual exclusion, the Linux operating system uses the CPU's PAUSE instruction. While



(a) NUMA-aware socket binding.

(b) TCP host kernel parameters.

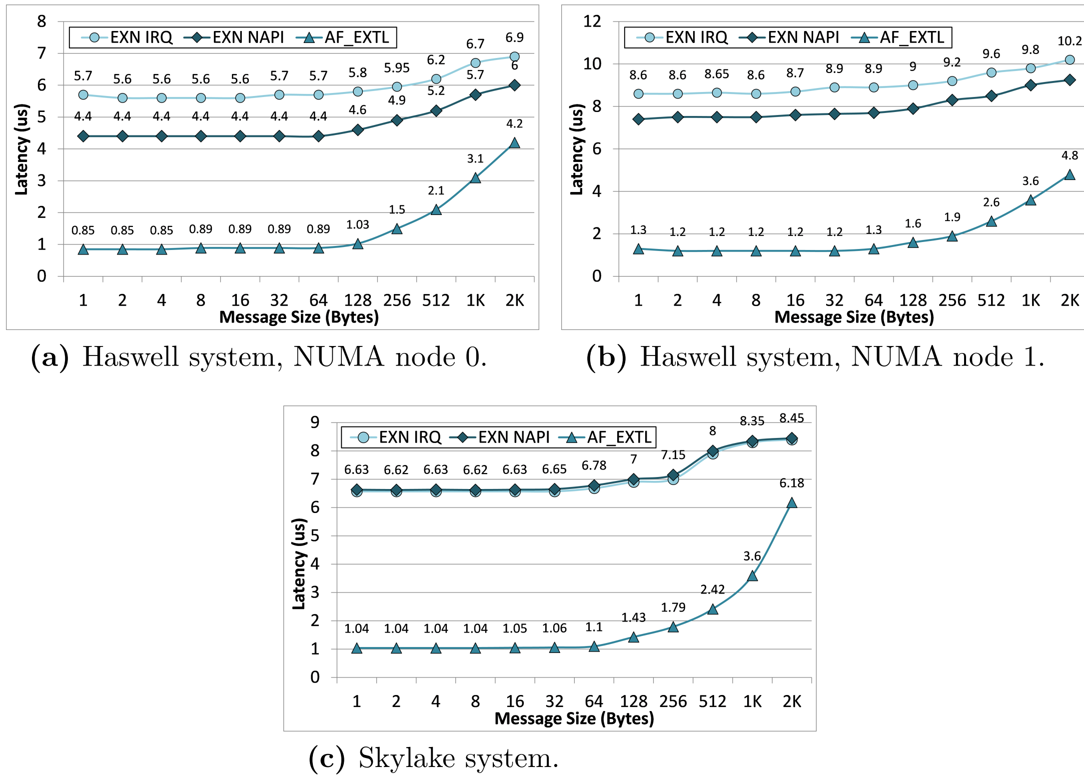
Figure 5.27: Netperf bandwidth performance for EXN with different TCP/IP tuning configurations.

the latency of the PAUSE instruction in older microarchitectures is about 10 CPU cycles, the Skylake microarchitecture has been extended to as many 140 cycles.

Figure 5.26b presents the netperf benchmark results for two TCP streams. It is noticeable that, starting with two parallel streams, EXN is able to provide the full bandwidth performance of the Extoll technology. For message sizes ranging from 4 KB to 128 KB, EXN provides 8.82 GB/s. Similar to the single stream experiments, the Skylake system provides an inferior bandwidth performance compared to Haswell.

In addition, Figure 5.27 presents two experiments where the TCP/IP system configuration has been modified. Figure 5.27a presents the results of the evaluation of the NUMA phenomenon. When an application is executed without any system tuning, the operating system can schedule the application to different CPU cores residing on different sockets depending on the current workload of the system. Therefore, between different application runs, variability in the overall performance can be observed. In case of network communication, this can result in a decreased bandwidth performance. In these experiments, netperf has been assigned to a specific socket through the numactl tool, i.e., NUMA node 0 and NUMA 1. The EXN module has been run in the polling mode. As can be seen in Figure 5.27a, the NUMA aware socket binding has a severe impact on the overall bandwidth performance. When being assigned to NUMA node 1, the peak bandwidth for a single TCP connection is 5.3 GB/s, while two streams result in a peak performance of 7.2 GB/s.

The second experiment series is presented in Figure 5.27b. The Haswell system's sysctl is configured as described in section 5.6.1.1. Especially for larger message sizes, a single stream TCP connection can benefit from this configuration. The peak bandwidth performance is increased to 8.3 GB/s for a single stream. For multiple streams, this configuration has no impact on the overall bandwidth.



(a) Haswell system, NUMA node 0. **(b)** Haswell system, NUMA node 1.

(c) Skylake system.

Figure 5.28: TCP/IP PingPong benchmark results for TCP connections.

5.6.3.2 Latency Results

Another important metric for HPC systems is the latency performance. Figure 5.28 shows the results of the TCP-PingPong benchmark for the EXN module and the direct sockets address family `AF_EXTL`. Figure 5.28a and Figure 5.28b display the latency results from the Haswell system, including experiments for the NUMA phenomenon. Depending on the CPU socket, the observed latency of AF_EXTL varies between 0.85 us and 1.3 us for small messages. For EXN, the observed latencies vary even more, i.e., for small messages in between 5.7 and 8.6 us. In Figure 5.28c, the latency performance of the Skylake system is displayed. As expected, the observed latency for EXN is much higher as for the Haswell system when bound to NUMA node 0. AF_EXTL, on the other hand, is an interesting case. Even though it does not utilize any spinlocks in its implementation, the latency performance is inferior when compared to the Haswell system. This phenomenon needs further investigation.

5.6.3.3 Host CPU Utilization

For most large-scale applications, transferring data with high bandwidth and low latency is a key criterion for increasing the overall application performance. However,

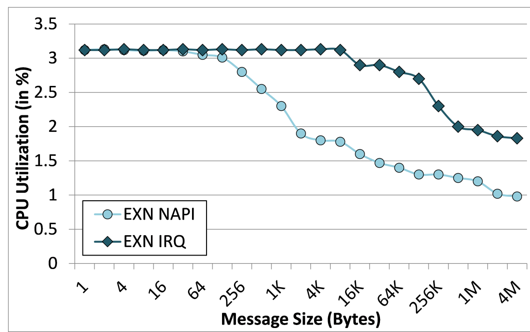


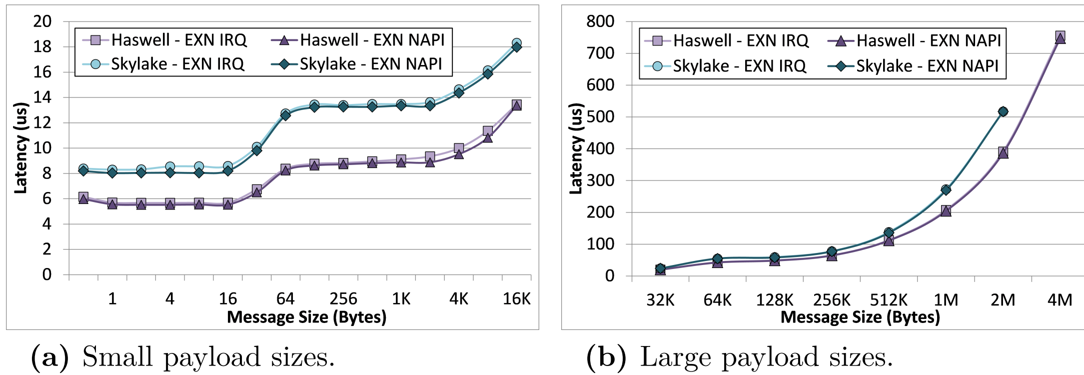
Figure 5.29: Host CPU utilization during netperf bandwidth test.

the overhead associated with a message transfer is also an important factor for the overall system performance. As described in section 2.2.2, there are two data transmission methods: PIO and DMA. PIO is typically utilized for small message transfers and performed by the CPU. In case of DMA, a descriptor is usually assembled, which describes the data to be transferred. The actual data transfer is performed by a DMA controller without the involvement of a CPU, which allows the overlapping of communication and computation.

In order to monitor the host CPU utilization, two methods have been used during the experiments. netperf provides a command line option to enable the monitoring of the local and remote host CPU. In addition, vmstat can be used to collect and display a summary about the main memory and CPU utilization, but also to monitor processes, interrupts and block I/O. Both methods have been utilized to monitor the system statistics of the EXN module on the Haswell system. Figure 5.29 shows the CPU utilization results for interrupt-driven and polling mode. Note that both modes have a moderate CPU utilization even though both need to traverse the TCP/IP stack. Recall that EXN has an internal switch between the eager protocol utilizing the VELO unit and the rendezvous protocol, which relies on the RMA unit for data transmission. With increasing message sizes, both modes consume less CPU resources. This is mainly because VELO enforces a PIO data transfer mechanism while RMA offloads the data transfer to the Extoll NIC.

5.6.4 MPI Performance

Besides the microbenchmark analysis, the point-to-point MPI benchmark collection of the OSU Micro-Benchmarks suite 4.3 [74] is evaluated. The intention of this analysis is to receive an understanding about the potential performance an application would observe when being run over EXN. As before, each benchmark is run 10 times and the results are calculated as the arithmetical average of the runs.



(a) Small payload sizes.

(b) Large payload sizes.

Figure 5.30: OMB latency performance comparing Skylake and Haswell.

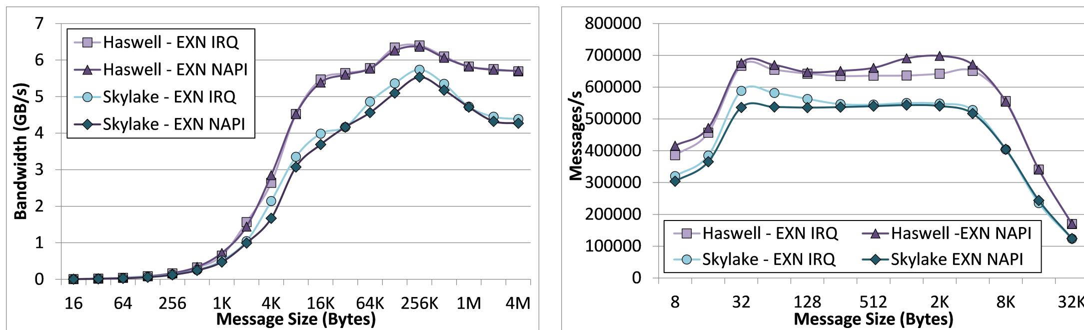


Figure 5.31: OMB bandwidth performance comparing Skylake and Haswell.

Figure 5.32: OMB message rate performance comparing Skylake and Haswell.

Figure 5.30 displays the results of the *osu_latency* benchmark for the Haswell and the Skylake systems. As previously observed, the Skylake system performs inferior compared to Haswell, when run over a code that relies on Linux's spinlock mechanism. Note that MPI applications run over the native Extoll OpenMPI implementation can utilize the full performance capabilities of the Extoll NIC, even when executed on a Skylake architecture. The reason for this is that the Extoll OpenMPI MTL is based on the user space libraries, and therefore, bypasses the operating system. It is noteworthy that the latency results of MPI over EXN are comparable to the raw performance measurements obtained through netperf.

Figure 5.31 presents the results of the *osu_bw* benchmark, which measures the unidirectional bandwidth of a point-to-point communication. While the latency performance is almost identical for MPI and netperf, the bandwidth results fall short, when compared to netperf. Even though the Haswell system is configured in accordance with the knowledge obtained through the microbenchmarks, the OpenMPI installation cannot utilize the bandwidth capabilities for a single connection. One

possibly reason could be that the OpenMPI configuration is suboptimal and needs to be tuned for TCP/IP.

Besides bandwidth and latency, another important metric is the message rate. As introduced in chapter 2, the message rate describes the number of messages that can be sent by a single process or rank in a specified period of time. The message rate is a good indicator for how well the processing of independent processes can be overlapped, but varies for different message sizes, is limited by the bandwidth and highly depends on the sending and receiving overhead. Figure 5.32 presents the message rate results for MPI over EXN. It can be seen that the message rate is pretty stable for message sizes between 32 bytes and 8 KB.

5.7 TCP/IP Summary

Even though the traditional TCP/IP communication stack introduces significant overhead, the support of legacy application codes enables the broad utilization of specialized SAN technologies. In addition, TCP/IP is supported by almost every communication environment. By providing TCP/IP communication means, different solutions can be tested before providing a native software implementation.

This chapter introduced two TCP/IP-based communication protocols for Extoll: EXT-Eth and EXT-DS. EXT-Eth describes the transmission of Ethernet frames over Extoll and emulates Ethernet communication. The protocol adds IP addressing and address resolution support to the Extoll interconnect and provides asynchronous, two-sided RDMA operations for large message sizes. The default MTU size enables the usage of super jumbo frames, which allows the protocol two fully utilize Extoll's bandwidth capabilities, and provides L3 routing support. The EXN module implements the EXT-Eth protocol and provides a traditional network interface to the Linux kernel. An initial performance evaluation with two Extoll Tourmalet ASICs shows promising results. While the latency is impacted by the latency of traversing the TCP/IP stack, the peak bandwidth of Extoll can be fully leveraged by EXN.

EXT-DS, on the other hand, introduced the acceleration of TCP point-to-point connections by providing kernel bypass data transfers, while maintaining traditional streaming socket semantics. The protocol describes two modes: BCopy and ZCopy. The BCopy mode provides seamless support for legacy applications, which means that the additional buffer copy from user to kernel space is still required. The ZCopy mode provides a true zero-copy data path, but expects application buffers to be pinned to the Extoll card, which allows physical addressing of the memory and

prevents the exemption of the buffer through the operating system. Besides the protocol specification, a prototype implementation of EXT-DS was presented through the introduction of a new transport protocol called `AF_EXTL`. Currently, the address family supports the BCopy mode and is limited to VELO communication. However, an initial evaluation showed that the latency performance of EXT-DS is able to provide similar results as the native Extoll communication. For small messages, the latency is about 0.85 us. Given that the EXN module was able to provide the peak bandwidth of Extoll, it is expected that EXT-DS will be able to achieve similar results while minimizing the host CPU utilization even further.

Chapter 6

Efficient Lustre Networking Protocol Support

Top-tier scientific HPC systems are constantly increasing in scale to improve application resolution and reduce the time to solution. Such an increase in scale also intensifies the system complexity and decreases the overall system reliability. To cope with system failures, most scientific applications periodically write out memory states. Besides storing numerical output from simulations, this defensive, bursty I/O (i.e., checkpointing) is one of the main sources of I/O activity in HPC environments. Therefore, the efficient storage connectivity is an essential requirement to serve the needs of large-scale scientific codes. This chapter discusses how a parallel file system such as Lustre can effectively leverage the capabilities of the Extoll interconnect.

Lustre is an open-source, parallel, distributed file system for Linux-based system environments [165, 166]. Its design targets massive scalability, high performance, and high availability. Within the scientific HPC community, Lustre is one of the most deployed file system solutions due to its support for data-intensive applications, which makes it an attractive choice for this work. The majority of the worldwide TOP100 supercomputing deployments utilizes Lustre for their high-performance parallel file and storage systems. For example in the TOP500 list from November 2017, nine of the TOP10 and more than 70 of the TOP100 systems deployed Lustre. However, one of the key disadvantages of Lustre is its lack of implementation-specific documentation, in particular the Lustre networking protocol called LNET.

The contributions of this chapter are twofold. First, it provides a comprehensive analysis and summary of the Lustre networking protocol semantics and interfaces, which need to be understood in order to design the protocol support for a custom interconnect such as Extoll. The second contribution is the design and implementation

of the *Extoll Lustre Networking Driver* (EXLND), which maps the LNET protocol semantics onto the Extoll networking technology. This chapter summarizes and extends contributions to international workshops and conferences [15, 16].

6.1 Introduction to the Lustre File System

Lustre provides a single, POSIX-compliant name space for small to medium-scale deployments, but also for large-scale storage platforms with hundreds of petabytes. Lustre's architecture employs a client-server network where distributed, object-based storage is managed by servers and accessed by clients utilizing an efficient network protocol. On a client, the mounted Lustre file system is represented as a coherent POSIX file system to applications. But, clients do not access storage directly; all I/O transactions are sent over the network.

The key concept of Lustre is to separate large, throughput-intensive block I/O from its small, random, IOPS-intensive metadata traffic. In other words, Lustre separates the metadata storage (inode) and block data storage (file content).

The following sections provide an introduction to the Lustre file system components, the network protocol, I/O operations and file striping. More details about the Lustre file system internals can be found in a technical report written by Wang et al. [167] and the Lustre Operations Manual [168].

6.1.1 File System Components

Lustre is a parallel file system utilizing distributed, object-based storage, which is managed by servers and accessed by client computers over a high performance interconnection network. Three different categories of servers can be distinguished: management, metadata, and object storage servers. The following provides an overview of the major Lustre hardware components [168]:

MGS + MGT The *Management Server* (MGS) acts as a global registry and stores the configuration information and service state for all active Lustre file system components. There is usually only one MGS for a given network and all Lustre components register with the MGS during startup. Lustre clients retrieve information from the MGS when mounting the file system. The MGS stores the configuration data on the *Management Target* (MGT).

MDS + MDT The *Metadata Server* (MDS) records and presents the file system namespace (file and directories) and is responsible for defining the file layout, i.e.,

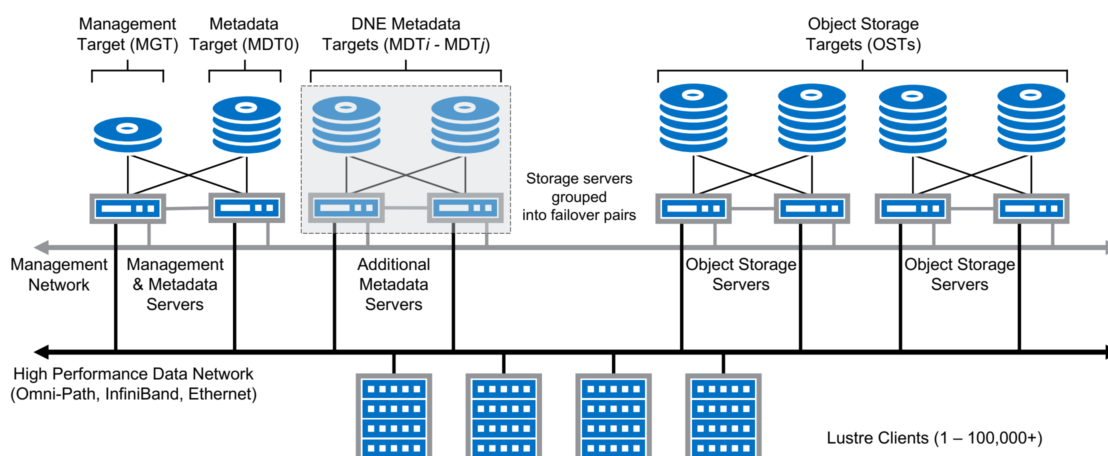


Figure 6.1: Lustre file system components in a basic setup [165].

the location of data objects. The MGS stores the metadata in key-value index objects, which contain the file system inodes (such as filenames, directories, permissions and file layout), on the *Metadata Target* (MDT). The MGS and MDS can be co-located and share storage space. To improve metadata scaling and system-wide fault tolerance, the *Distributed Namespace* (DNE) feature can be used to store metadata across multiple MDTs.

OSS + OST The main task of *Object Storage Servers* (OSSs) is to provide file I/O services and network request handling for their local *Object Storage Targets* (OSTs). An OSS serves as a block device to the clients. Files can be striped across multiple OSTs and written to concurrently.

Clients *Lustre clients* can be computational, visualization or desktop nodes. Each of the clients expose the Lustre file system to the operating system by mounting a POSIX-compliant instance utilizing the *Lustre network protocol* (LNET). Applications can use standard POSIX calls for Lustre I/O.

Network One of the most crucial components is the network. Lustre is a network-based file system and all I/O transactions are sent over the network using RPCs. It is common that clients have no local persistent storage.

Each type of server has its storage target devices directly attached. Various forms of data storage are possible, including simple hard disks, but also enterprise grade network attached storage. Recent versions of Lustre (≥ 2.6) utilize the *Zettabyte File System* (ZFS), based on the OpenZFS implementation, as the default data storage back end [168], older versions use Lustre's native LDISKFS, a modified version of

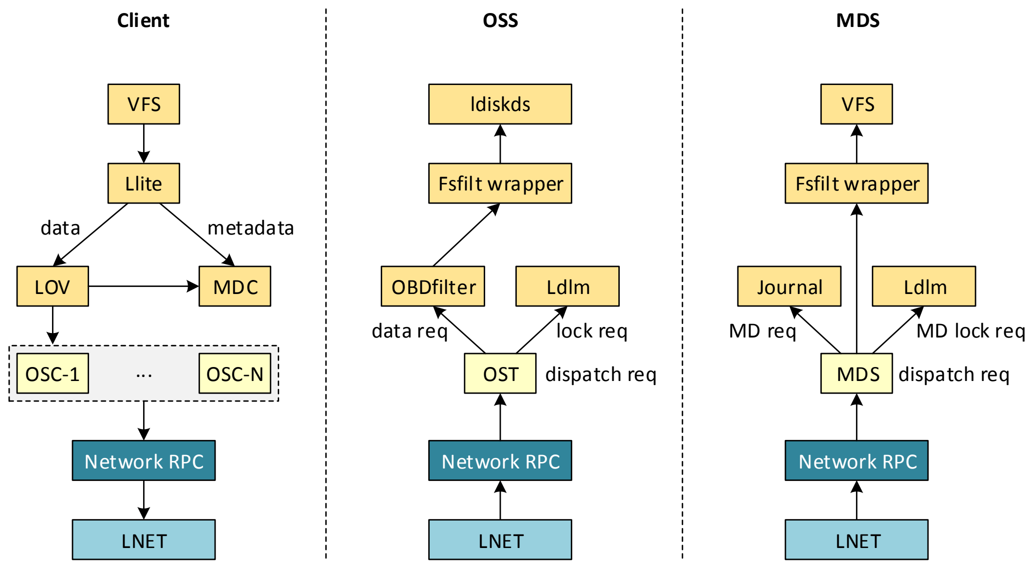


Figure 6.2: An overview of the internal Lustre architecture [167].

EXT4. Figure 6.1 displays an example configuration deploying all of the previously described Lustre file system components. The clients are connected to the storage system over a high performance data network, e.g., Infiniband or Omni-Path. On the top left, an MGS with its MGT and a MDS with its MDT are displayed. To improve the metadata performance and provide failover services, additional metadata servers and metadata targets can be configured. On the top right, the OSSs are shown. Each OSS serves four OSTs. In addition to the high performance data network, the servers are connected over an additional management network.

Another important Lustre component is its modularized software architecture. Figure 6.2 provides an overview of the architecture internals on the client, OSS, and MDS. On the client site, the *Virtual File System* (VFS) module provides the POSIX interface to the Lustre clients. The clients access files or directories using POSIX calls through the VFS and Lustre returns the metadata and data. All Lustre network I/O is transmitted using the Lustre Network protocol (LNET).

6.1.2 Network Communication Protocol

The Lustre Networking protocol, called LNET, provides the network abstraction for Lustre [169, 170]. LNET is implemented as a modular software architecture, as shown in Figure 6.3, which provides an abstract communication interface to Lustre. It consists of two layers:

(1) the LNET code module, which is used by Lustre to interface the network;

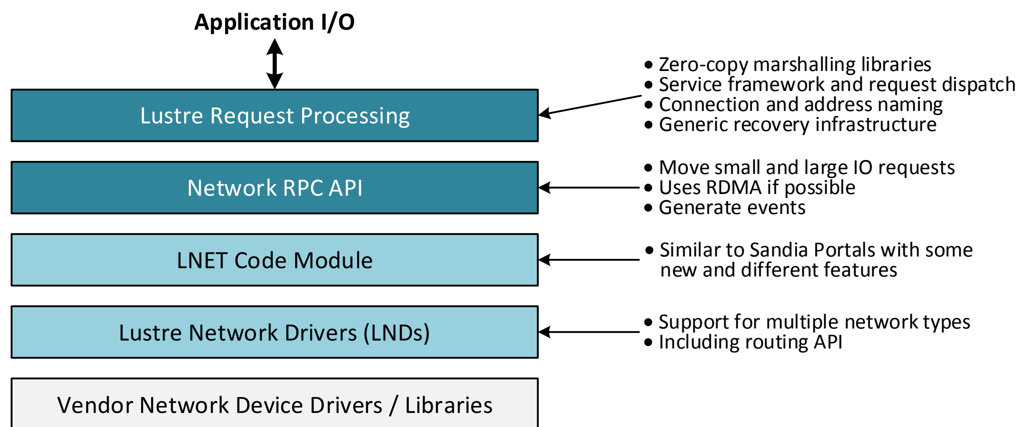


Figure 6.3: Overview of the LNET software architecture [165].

(2) the Vendor-specific Lustre Network Driver (LND), which provides a layer of abstraction from the network hardware to LNET.

Each supported network type implements its own hardware-specific LND. The LND utilizes the native network interface below and features an LNET-specific standard interface to communicate with upper layers. LNET is independent from the Lustre file system; the LNET core module and the LNDs are implemented as kernel modules in a separate directory of the source tree. The Lustre network protocol is connection-based and end-points maintain a shared, coordinated state. For each active connection from a client to a server storage target, a server keeps exports, while a client keeps imports as an inverse of the server exports. The Lustre manual [168] lists the key features of LNET as follows:

- Native support for multiple, commonly-used networks via LNDs, including Infiniband and TCP/IP-enabled networks like Gigabit Ethernet;
- RDMA support, depending on the underlying network technology;
- High availability and recovery features with failover for the servers;
- Support of multiple network types simultaneously, with routing between multiple LNET subnets and disparate networks by employing LNET routers.

LNET supports heterogeneous network environments and routing natively, which means that Lustre nodes can exist in two or more different physical networks. I/O can be aggregated across multiple independent network interfaces, which enables network multipathing. Dedicated routing nodes, called LNET routers, provide a gateway between different LNET networks and allow servers and clients to be multi-homed.

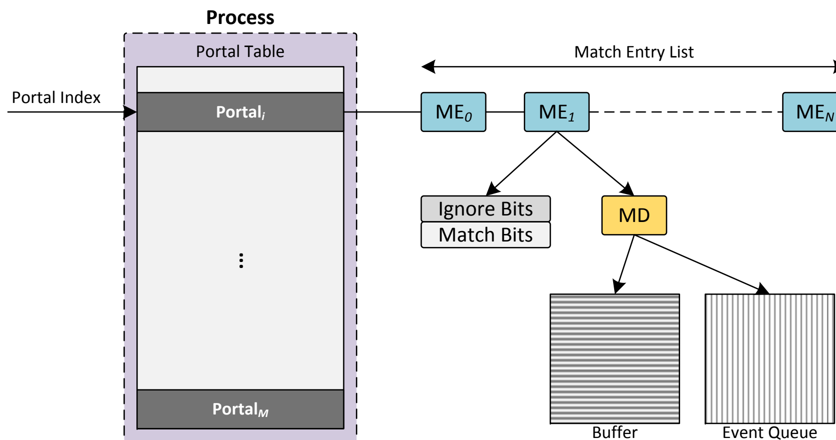


Figure 6.4: Illustration of the Lustre LNET addressing scheme [167].

LNET is a message passing API, which originated from the Sandia Portals API [171]. As a consequence, LNET inherits some of the Portals API's properties. Internally, peers are referenced by a global process ID. The global process ID consists of a network identifier (NID) in the format `<address>@<network type>[network number]` and a Lustre-internal process ID. In this context, a request posted to another node is performed through an RPC. In Lustre, a *portal* consists of a list of match entries (MEs). Each ME can be associated with a buffer, as displayed in Figure 6.4, and contains match bits and ignore bits, which are 64-bit identifiers that are used to decide if the incoming message can utilize the associated buffer space. The ME list of a portal is used to associate an incoming LNET message with its portal, and also, to find the corresponding memory descriptor (MD) [167].

LNET differentiates different payload types, including requests, bulk read/write data (large I/O to be written or read by clients), and metatraffic (file lock grants/releases, OSTs for specific file, etc.). To illustrate the Portals-like RPC mechanism, the following describes how a bulk data transfer from an OSS to a client is initiated (as shown in [167]). First, the client posts a request to a server to send the data back to the client. For example, this could be an RPC containing the request to read ten blocks from a file and the information that the client is ready to receive a bulk data transmission. After receiving the request, the OSS initiates the bulk data transfer to the client. Once the data transmission is completed, the server notifies the client of the completion by sending a reply message. As there are two different data flows, the bulk data and the reply data, the Lustre client needs two portals for this transfer. The portals in this example are called *bulk portal* and *reply portal*.

Most LNET actions are initiated by Lustre clients. The most common activity for a client is to initiate an RPC to a specific server target. A server may also initiate

an RPC to a target on another server. For example, an MDS sends an RPC to the MGS to retrieve configuration data; or an RPC from MDS to an OST to update the MDS's state with available space data. An OSS is relatively passive: it waits for incoming requests from either an MDS or Lustre clients.

6.1.3 Client Services and File I/O

The Lustre client is implemented as a kernel module. It provides an interface between the Linux virtual file system and the Lustre servers, and presents an aggregate view of the Lustre services to the host operating system as a POSIX-compliant file system. To applications, a Lustre client mount looks just like any other file system, with files being organized in a directory hierarchy.

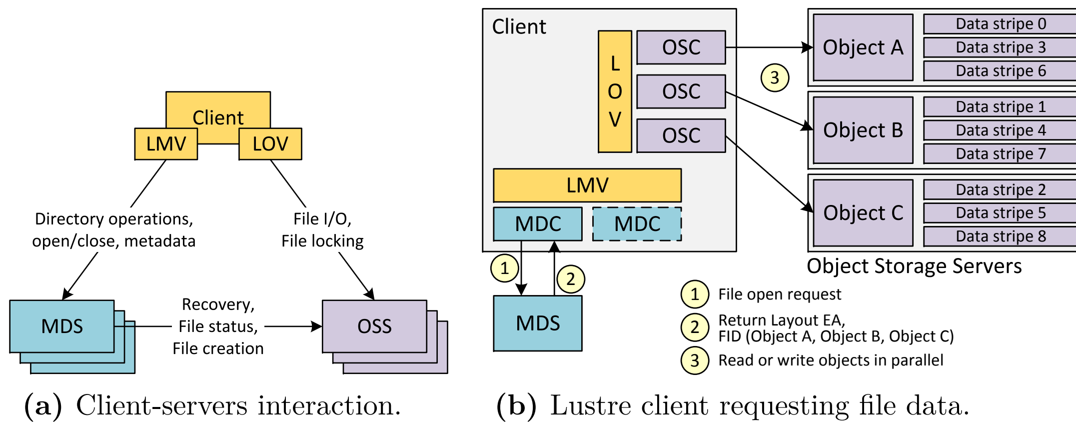
6.1.3.1 Client I/O Overview

Lustre employs a client-server communication model. Each connection has an initiator (the client end of the connection) and a target (the server process). The client must be able to retrieve file-specific metadata from the MDS when opening or closing a file, but also needs to interface with the OSTs for direct data access. The main Lustre client services [165] are the *management client* (MGC), one or more *metadata clients* (MDC) and one *object storage client* (OSC) per configured OST in the file system.

The *logical metadata volume* (LMV) aggregates the MDCs and presents a single logical metadata namespace to the Lustre clients, which enables the transparent access to all MDTs. This allows the client to see the directory tree stored on multiple MDTs as a single coherent namespace, and striped directories are merged on the clients to form a single visible directory to users and applications.

The *logical object volume* (LOV) aggregates the OSCs to provide transparent access across all the OSTs. Thus, a client with a mounted Lustre file system sees a single, coherent, synchronized namespace, and files are presented within that namespace as single addressable data objects, even when striped across multiple OSTs. Several clients can write to different parts of the same file simultaneously, while, at the same time, other clients can read from the file. Figure 6.5a provides an overview of the client services and possible client-server interactions

Figure 6.5b illustrates an exemplary file access initiated by a Lustre client. First, the Lustre client sends an RPC to the MDS to request a file lock (1). This can either be a read lock with look-up intent or a write lock with create intent. When the MDS has processed the request, it returns a file lock and all available metadata and file layout extended attributes to the Lustre client (2). If the file is new, the MDS will



(a) Client-servers interaction. **(b)** Lustre client requesting file data.

Figure 6.5: Overview of Lustre I/O operations [165].

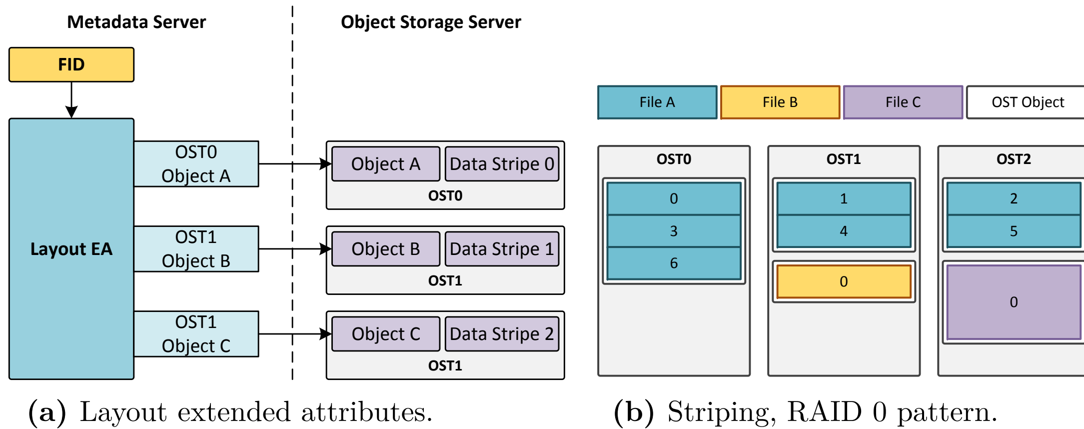
also allocate OST objects for the file based on the requested layout when the file is opened for the first time. With the help of the file layout information, the client is able to access the file directly on the OSTs (3).

6.1.3.2 Overview of Metadata and File Layout Key Terminology

Each file in Lustre has a unique layout. The layout is described through several attributes. In the following, an overview of metadata attributes, the file layout, and the locking mechanism is presented.

Lustre Inode On Linux and other Unix-like operating systems, an inode is a data structure that stores a file's metadata information except its name and the actual data. In the context of Lustre, a Lustre inode corresponds to an MDT inode. The default inode size is 2 KB and contains the metadata for a Lustre file. The metadata consists of typical metadata from `stat()`, e.g., the user identifier, group identifier and permissions, and *layout extended attributes* (Layout EAs).

File Identifier With the introduction of the Lustre 2.x series, *Lustre file identifiers* (FIDs) have replaced UNIX inode numbers and are used to identify files or objects. An FID is a unique 128-bit value comprising of a 64-bit sequence number, which is used to locate the storage target, a 32-bit object identifier (OID), which references to the object within the sequence, and a 32-bit version number, which is currently unused. The sequence number is unique across all Lustre targets in a file system (OSTs and MDTs). FIDs are not bound to a specific target, they are never re-used and can be generated by Lustre clients.



(a) Layout extended attributes. **(b)** Striping, RAID 0 pattern.

Figure 6.6: Lustre file layout overview [165].

Layout Extended Attributes Lustre utilizes Layout EAs to save additional information about a file. Layout EAs are stored on an MDT inode and contain POSIX access control lists (ACLs), the file layout information (e.g., the location of the data) such as OSTs and object ID, and OST pool memberships. When a client wants to access (read from or write to) a file, it first queries the MDT with a request to fetch the list of FIDs containing the file's data. With this information, a client can directly connect to the OSSs where the data objects are located at and perform I/O on the file. If the file is a regular file (not a directory or symbolic link), the layout points to 1-to-N OST object(s) on the OST(s) that contain the file data. There are mainly two possible cases. If the layout EA points to one OST object, all of the file data is located on that particular OST. If it points to more than one OST object, the file data is *striped* across multiple different OSTs with one object per OST.

File Striping File striping is one of the key factors leading to the high performance of Lustre at larger scales. Lustre has the ability to stripe data across multiple OSTs in a circular round-robin fashion, which is comparable to the RAID 0 pattern. The file layout is allocated by the MDS when the file is accessed for the first time and fixed once the file is created. The stripe layout is selected by the client, either by the configured policy (e.g., inheritance from the parent directory) or by the user or application at runtime utilizing the Lustre user library `llapi`. `llapi` provides a set of commands used for setting Lustre file properties within a C program running on a cluster environment. Striping can be used to improve the overall throughput performance by enabling the aggregate bandwidth to a single file to exceed the bandwidth of a single OST. The number of objects in a single file is called *stripe count*. When a segment or "chunk" of data which is written to a particular OST

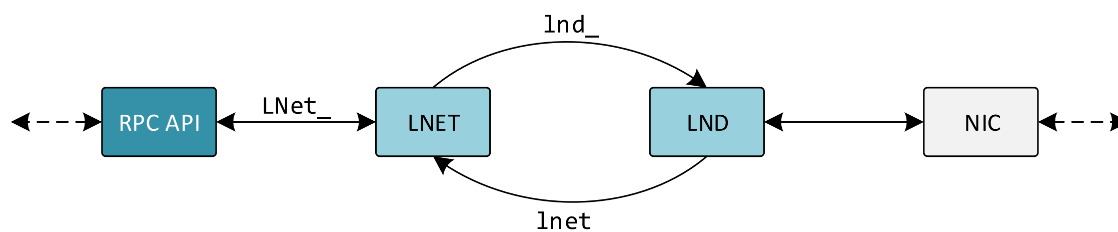


Figure 6.7: LNET API naming conventions.

object exceeds the specified *stripe size*, the next chunk of data is written to the next object. The maximum number of OSTs that a single file can be striped across is 2000, even though a Lustre file system can comprise of more than 2000 OSTs.

Lustre Distributed Lock Manager The *Lustre Distributed Lock Manager* (LDLM) provides byte-granular file and fine-grained metadata locking. It ensures that files are coherent between all clients and servers. Each MDT and OST has its own LDLM. Metadata servers use inode bit-locks for file lockup, state, extended attributes, and layout. Object storage servers provide extent-based locks for OST objects. File data locks are managed for each OST. Clients can be granted read extent locks for a part or all of a file, allowing multiple concurrent readers access to the same file. Clients can be granted non-overlapping write extent locks for regions of the file. Multiple Lustre clients may access a single file concurrently for both read and write, which should minimize potential bottlenecks during file I/O.

6.2 Lustre Networking Semantics and Interfaces

LNET is designed to be lightweight and efficient. It supports message passing for processing RPCs and RDMA for bulk data transfers. Even though the Lustre project is open source, there is almost no documentation available about its implementation details. Especially for LNET, the documentation is limited to its source code, a master thesis by T. Groschup [172] focusing on Lustre version 2.4, and a technical report by Wang et al. [167], which describes Lustre's internals for version 1.8, and therefore, is mostly outdated.

In order to map the Lustre network protocol to a new interconnect technology such as Extoll, it is particularly important to understand Lustre's networking semantics and interfaces. The following sections present a comprehensive overview of LNET's building blocks, including an API summary, LNET's communication semantics, and the credit system. In addition, an overview of existing LNDs is provided.

```
1 typedef struct lnet_lnd {
2     int (*lnd_startup)(struct lnet_ni *ni);
3     void (*lnd_shutdown)(struct lnet_ni *ni);
4     int (*lnd_ctl)(struct lnet_ni *ni, unsigned int cmd, void *arg);
5     int (*lnd_send)(struct lnet_ni *ni, void *private,
6           lnet_msg_t *msg);
7     int (*lnd_recv)(struct lnet_ni *ni, void *private,
8           lnet_msg_t *msg, int delayed, unsigned int niov,
9           struct kvec *iov, lnet_kiov_t *kiov, unsigned int offset,
10          unsigned int mlen, unsigned int rlen);
11    int (*lnd_eager_recv)(struct lnet_ni *ni, void *private,
12          lnet_msg_t *msg, void **new_privatep);
13    void (*lnd_notify)(struct lnet_ni *ni, lnet_nid_t peer,
14          int alive);
15    void (*lnd_query)(struct lnet_ni *ni, lnet_nid_t peer,
16          cfs_time_t *when);
17    int (*lnd_accept)(struct lnet_ni *ni, cfs_socket_t *sock);
18    void (*lnd_wait)(struct lnet_ni *ni, int milliseconds);
19    int (*lnd_setasync)(struct lnet_ni *ni, lnet_process_id_t id,
20          int nasync);
21 } lnd_t;
```

Listing 6.1: The LND struct.

6.2.1 Naming Conventions and API Summary

For the communication between LNET and an LND, three API groups can be distinguished, as illustrated in Figure 6.7. Function names starting with `LNet` are external function calls from the upper layers. All other functions using lower cases belong to the internal API of LNET. Within LNET, an LND has two sets of APIs. If a function name starts with `lnd_`, it is part of the API used by LNET to communicate with an LND. If a function name starts with `lnet_`, it describes the interface for the LND to communicate with LNET. In addition, there are two important data structure types, which are used in nearly every function call to the LND: (1) `lnet_ni_t`, which provides all information about the LND for Lustre, and (2) `lnet_msg_t`, which describes an LNET message.

The LND has to be registered with LNET. Otherwise, LNET does not know how to utilize the network interface. This is done in the manner of a Linux kernel module. LNET provides a structure called `lnd_t`, which consists of function pointers, as displayed in Listing 6.1. Even though an LND is not required to implement all of the listed functions, the following list describes the minimal set of functions that need to be implemented by an LND in order to provide full functionality:

(1) `lnd_startup()`, `lnd_shutdown()`: These functions are called from LNET to bring up or shut down a network interface corresponding to the LND.

(2) `lnd_send()`, `lnd_recv()`, `lnd_eager_recv()`: These functions are called for sending or receiving outgoing and incoming LNET messages respectively.

(3) `lnd_query()`: Despite being described as optional [167], LNET will not start an LND if this method is not implemented.

The second set of function calls are LNET routines exported for LNDs to use:

(1) `lnet_register_lnd()`, `lnet_unregister_lnd()`: Each LND module has to call these functions to register or unregister an LND type at module startup or shutdown respectively.

(2) `lnet_parse()`: For each message received, an LND has to call this function to notify LNET that a new message has arrived. After parsing the header, LNET forwards the message to an LND's receive function. An LND can indicate its RDMA capability by calling this function with the RDMA flag set to one.

(3) `lnet_finalize()`: This function is called by LNDs for both incoming and outgoing messages. When a message has been sent, this call informs LNET about the status by generating a corresponding LNET event. For an incoming messages, this call indicates that the payload has been received.

6.2.2 Memory-Oriented Communication Semantics

The Lustre network protocol utilizes two memory-oriented communication semantics to propagate requests between Lustre clients and servers:

`LNetPut()` This function sends data asynchronously and is mostly issued by clients to request data from a server or to request a bulk data transmission to a server. Servers use `LNetPut()` to send a reply back to the client.

`LNetGet()` This function serves as a remote read operation and is used by servers to retrieve data in bulk read transmissions from clients. Clients issue this function only in one case, the router pinger. This technique is used to verify whether an LNET router is still alive.

Internally, these two functions are mapped onto LNET messages, and LNET expects a specific behavior from an LND, which depends on the message type. The following sections provide an overview of the LNET message types and the expected communication semantics, but also describe the characteristics of the payload of an LNET message.

Table 6.1: Overview of LNET message types.

Type	Description
PUT	A PUT message is used to indicate that data is about to be sent through an RDMA transfer from the node to the peer. This is the simplest LNET message.
ACK	An ACK message is used when the connection of a client or server is established. It is also used as an optional response to a PUT and can be explicitly requested by the sender. The ACK message is then generated by the LNET layer once it receives the PUT.
GET	A GET message is used by a sender to request data from a peer and always expects a REPLY message in return.
REPLY	The REPLY message is the response to a GET message.

6.2.2.1 LNET Message Types

LNET defines four main message types, as presented in Table 6.1. The type is indicated in the field `msg_type` of the structure `lnet_msg_t`. The following LNET message combinations can occur: `PUT`, `PUT+ACK`, `ACK`, and `GET+REPLY`. A Portal RPC can either be one or a union of the described message combinations. Also, an LND must support all four message types to be fully functional. In general, an `LNetPut()` initiates the transmission of a `PUT` message with an optional `ACK`, while `LNetGet()` is translated into the message combination `GET+REPLY`.

6.2.2.2 Communication Semantics

For both `LNetGet()` and `LNetPut()`, LNET calls an LND's send function and hands the LNET message over in form of a `lnet_msg_t` structure. It is the task of an LND to decode the message and process it accordingly. After an LNET message has been sent, LNET expects to be notified from an LND by calling `lnet_finalize()` once the transmission is completed.

For an `LNetPut()`, this is all that needs to be done on the client (initiator) side. The `LNetGet()` needs to be treated differently. As the initiating node does not have any data to be sent to the target node, since the LNET message representing a `GET` does not have any data attached. But, it does contain a memory descriptor, which describes the data sink for the `GET` message. To notify LNET that the data has been written to the sink buffer, an LND needs to generate a `REPLY` message. Both the `GET` and the `REPLY` message need to be passed to `lnet_finalize()` upon completion of the data transmission.

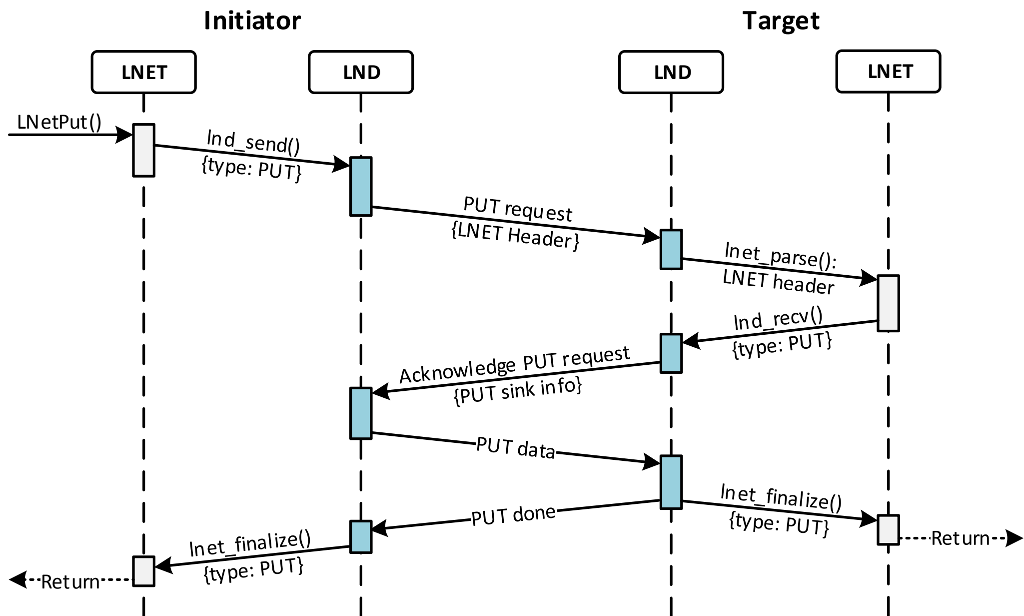


Figure 6.8: Overview of LNetPut() communication sequence.

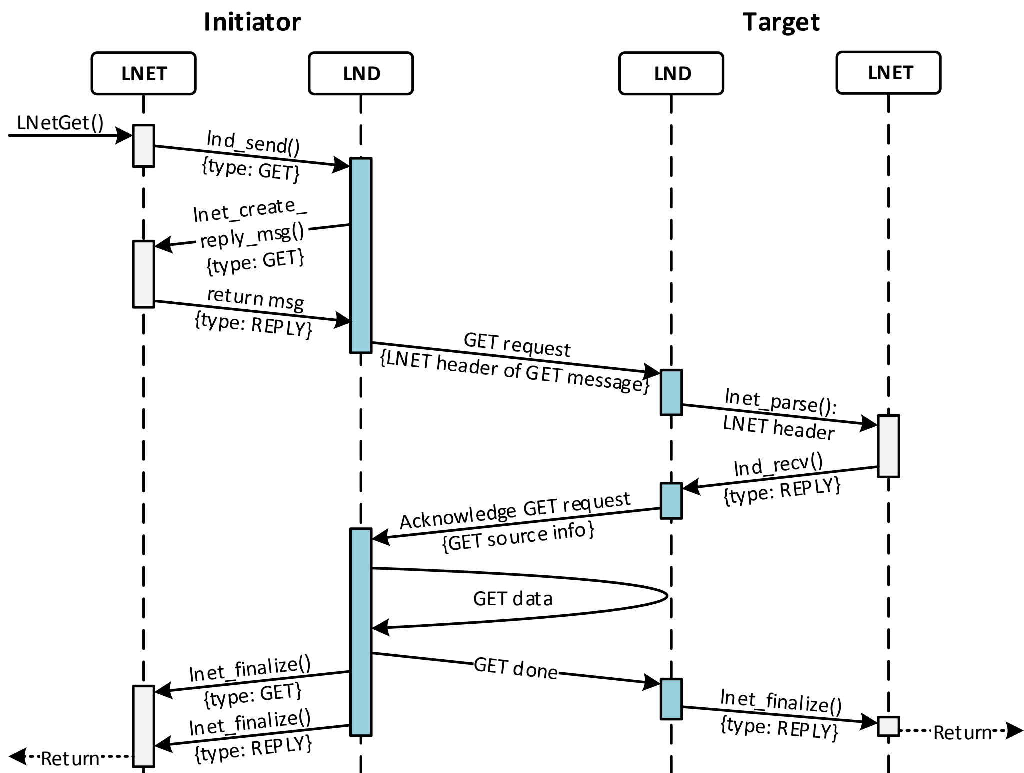


Figure 6.9: Overview of LNetGet() communication sequence.

On the target side, the hardware must notify an LND that a new message has arrived. This can be done through interrupts or an LND can periodically poll for new messages, e.g., by using kernel threads. After receiving a new message, an LND forwards the header of the message to LNET by calling `lnet_parse()`. LNET checks if the message has arrived at its final destination and calls an LND's receive function for the received message. In addition, an LND can forward private data to the receive function through the parameter `void *private` of `lnet_parse()`.

In case of an `LNetPut()`, the receive function is called with an LNET message containing the local data sink, and is expected to write the data to this buffer. After the copy operation, the LND must call `lnet_finalize()` for the corresponding LNET message to signalize that the data has been received. In case of an `LNetGet()`, the receive function is called with the data that has to be sent to the initializing node, which is attached to an LNET message. After the data has been sent, `lnet_finalize()` has to be called with the corresponding LNET message.

The chronological sequence of LNET and LND API calls for `LNetPut()` and `LNetGet()` is illustrated in Figures 6.8 and 6.9 respectively. It is at the discretion of an LND how the actual data transfer is implemented.

6.2.2.3 Payload and Memory Descriptors

As mentioned before, the payload of an LNET message is restricted to a maximum of 1 MB, and the number of segments cannot exceed 256 pages. Unlike TCP/IP, LNET does not fragment or re-assemble messages. LNET assumes that upper layers, such as Portal RPC, never provide a payload bigger than 1 MB. There are several reasons for this limitation – for example the pre-set limit makes the buffer management easier. Also, some low-level drivers have a limited number of scatter/gather buffers, such as 256. Another benefit is that an upper layer such as Portal RPC can fragment data more easily if buffers are posted in pages.

The payload attached to an LNET message resides behind a memory descriptor (MD). The send function of an LND only receives an LNET message, and has to check for the data in the LNET message. In case of a `PUT`, the payload is directly attached to the LNET message. In case of a `GET+REPLY`, there is no data attached, but the data sink information is encoded in the message. The receive function of an LND receives an LNET message with explicit pointers to the memory regions that have to be used. A memory descriptor points to one of two different buffer types:

1. Kernel virtual address space, which is continuously mapped and organized in data structures of type `struct kvec` (see Listing 6.2);

```
1 struct kvec {
2     void *iov_base; /* *never* holds a userland pointer */
3     size_t iov_len;
4 };
```

Listing 6.2: A memory descriptor fragment with a kernel virtual address.

```
1 typedef struct {
2     /* Pointer to the page where the fragment resides */
3     struct page    *kiov_page;
4     /* Length in bytes of the fragment */
5     unsigned int   kiov_len;
6     /* Starting offset of the fragment within the page. Note that
7      * the end of the fragment must not pass the end of the page;
8      * i.e., kiov_len + kiov_offset <= PAGE_CACHE_SIZE. */
9     unsigned int   kiov_offset;
10 } lnet_kiov_t;
```

Listing 6.3: A page-based fragment of a memory descriptor.

2. A page list, which is organized in data structures of type `lnet_kiov_t`, as displayed in Listing 6.3.

For messages larger than a page, both buffer types are organized as arrays of the respective type. An array describes the whole memory of the message, but the pages do not have to be continuous in physical memory. Also when the memory region is fragmented, all fragments but the first one start on a page boundary, and all but the last end on a page boundary. For bulk data transfers, an LND needs to map the buffer onto scatter/gather lists, which can be physically addressed by the NIC and are pinned in the main memory.

6.2.3 Credit System

Lustre provides a user-tunable credit system for the send and routed receive paths. The credit counts can be accessed through `/proc/sys/lnet/` and configured through module parameters. The following credits need to be configured on a Lustre client for a given LND:

Peer Credits The peer credit specifies the number of concurrent sends to a single peer. A negative credit count indicates the number of messages are awaiting a credit.

TX Credits The TX credit specifies the number of concurrent sends to all peers. A negative credit count indicates the number of messages are awaiting a credit.

Every send consumes one peer credit and one TX credit. In addition, Lustre provides credits accounting when routers receive a message destined for another peer. The peer and TX credits have an impact on how well a network connection scales with an increasing number of concurrent accesses. But, the higher the credit count, the more memory resources are consumed by an LND. For communication to routers not only the TX and peer credits must be tuned, but also the global router buffer and peer router buffer credits need to be configured:

Peer Router Credit The peer router credit governs the number of concurrent receives from a single peer. Its main objective is to prevent a single peer from using all router buffer resources. A credit is given back when the receive completes.

Global Router Buffer Credit This credit allows messages to be queued in order to select non-data RPCs versus data RPCs with the intent to avoid congestion. LNET router nodes have a limited number buffers. The router buffer credits ensure that a receive only succeeds if appropriate buffer space is available.

6.2.4 Available Lustre Network Drivers

The most widely used network technologies for Lustre storage systems are TCP/IP networks and RDMA-capable networks that support the OFED software environment (refer to section 5.2.1). For TCP/IP networks, the ksocklnd.ko module, also referred to as Sockets LND, implements the LNET driver. Its implementation differs from the presented LNET semantics and message types. The Sockets LND utilizes its own message type called `HELLO`, which is used to mimic traditional TCP/IP communication, and does not support any other message type.

For RDMA network technologies, the ko2iblnd.ko module, also referred to as o2ib LND, supports Infiniband, Omni-Path, and RoCE in conjunction with the OFED environment. The LND supports all of the described message types. Its implementation follows closely the described memory-oriented communication semantics. Its bulk data transmission relies on Scatter/Gather Elements (see section 6.4.2).

Besides these two main LNDs, several other LNDs have been introduced over time. Examples are the Gemini LND [173] and the Portals4 LND [174]. As indicated by its name, the Gemini LND provides support for Cray's Gemini interconnect technology. The Portals4 LND implements the LNET driver for the BXI interconnect [175] and relies on the Portals 4 API. Currently, it is not included in the official Lustre release.

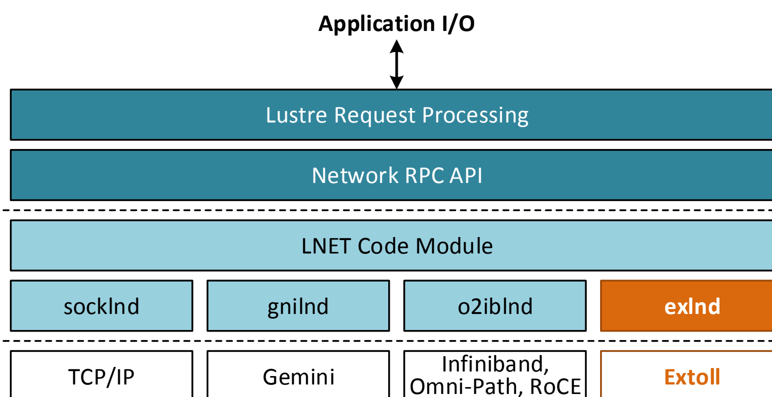


Figure 6.10: Lustre software environment with EXLND.

6.3 Design Challenges and Strategy

The main objective of EXLND is to provide an efficient mapping of Lustre's network protocol semantics onto the Extoll interconnect technology. Recapitulating the findings about LNET and its semantics from the previous section, EXLND needs to fulfill the following tasks:

- EXLND should support the `LNetGet()` and `LNetPut()` communication semantics and all LNET message types by efficiently mapping those to suited Extoll functional units and communication patterns. Candidates for the data transfers are both the VELO and RMA units.
- The solution should provide an efficient mechanism to support scatter/gather DMA operations. Recall that LNET messages can have a payload of up to 256 pages, which can be mapped onto scatter/gather lists.
- The general structure of an LND needs to be maintained so that EXLND can seamlessly blend into Lustre's and LNET's modular environment. Specifically, the expected sequence of API calls needs to be maintained.

Figure 6.10 displays the extended Lustre software environment with the targeted Extoll support. The design strategy for EXLND comprises of a twofold approach. First, the utilization of the ATU address translation scheme for scatter/gather lists is explored in section 6.4. The second part of the design focuses on the support of the LNET communication scheme by leveraging the RMA unit's innovative notification mechanism. The data transmission protocols and LNET message matching mechanism are presented in section 6.5. A prototype implementation of EXLND is presented in section 6.6.

6.4 Efficient RDMA with Vectored I/O Operations

Most modern NICs provide on-chip support for RDMA operations and address translation, which enables scatter/gather DMA. Among other things, those scatter/gather operations, also referred to as vectored I/O, have two major benefits: atomicity and efficiency. Atomicity means that one process can write into or read from a set of physical buffers, which can be scattered throughout the physical memory, without the risk that another process might perform I/O on the same data. In addition, this mechanism improves the efficiency since one vectored I/O operation can replace multiple ordinary reads or writes, and therefore, reduces the overhead. Scatter/gather DMA controllers provide the hardware support for scatter/gather I/O. To initiate such an operation, a controller needs the input modifier, also known as *scatter/gather list*, to offload the transfer to the NIC.

As described in section 6.2.2.3, the payload of an LNET message can consist of up to 256 pages. It is desirable to transfer the payload in one operation instead of multiple RDMA reads or writes. Other LNDs, such O2IB LND, implement vectored I/O by mapping the attached memory regions onto so called scatterlists. Within the kernel, a buffer to be used in a scatter/gather DMA operation is represented by an array of one or more scatterlist structures. As presented in section 3.2.5, the Extoll design features the address translation unit, which can be utilized to map page lists and contiguous kernel buffers into the Extoll address space. This functionality is used to provide scatter/gather DMA over Extoll.

The remainder of this section is organized as follows. First, the term physical buffer list is introduced followed by an overview of how Infiniband utilizes so called scatter/gather elements to support vectored I/O. The last part of this section focuses on the design and its limitations of scatter/gather DMA support for Extoll.

6.4.1 Memory Management

In the context of RDMA-enabled NICs, *memory regions* refer to continuous memory areas, which have been pinned in main memory and registered with a NIC. Such non-shared memory regions are also called physical buffer lists and consist of page or block lists, as depicted in Figure 6.11. NICs can access these physical buffers by using their physical addresses. Another important characteristic is that they cannot be swapped out of the main memory, which enables scatter/gather transfers.

For page lists, the page size has to be an integral power of two and all pages have to have the same size. The data can start at an offset into the first page, referred

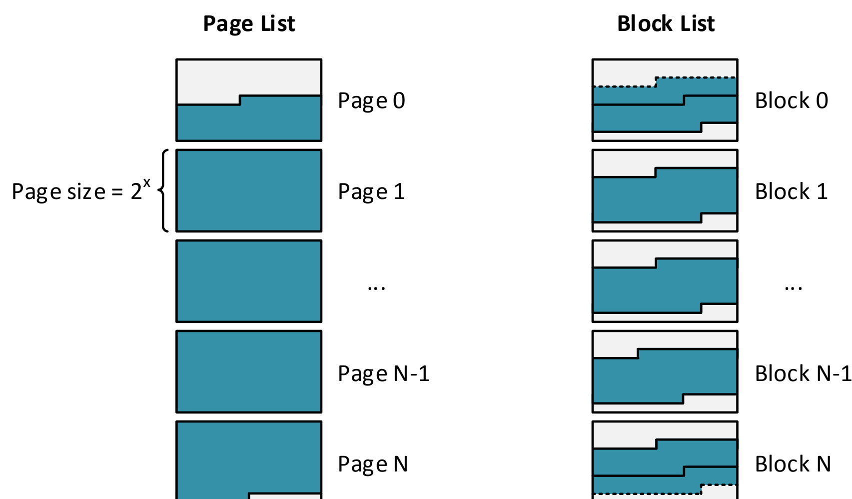


Figure 6.11: Physical buffer lists.

to as *first byte offset*, and can end on a non-page boundary, which means that the last page can be partially filled. Pages do not have to be continuous in memory. To perform scatter/gather I/O with page lists, the following input modifiers are needed: the page size, the first byte offset, the length, and the address list of the pages.

For block lists, the pages where the blocks are residing on need to have the same size. The block size itself is arbitrary and depends on the sizes supported by the NIC. As for the data boundaries, the same rules as for page lists apply. The data can start at an offset into the first block and can end at an offset into the last block. The dashed lines in Figure 6.11 outline the block versus data boundaries. The following modifiers are needed for scatter/gather transfers: the block size, the first byte offset, the length, and the address list of the blocks.

Depending on the underlying NIC technology, two different types of address translations can be distinguished: *onloading* and *offloading*. In case of onloading, the address translation is moved to the CPU and handled by the driver software. In case of offloading, the hardware typically has a dedicated controller that is able to gather data out of the memory onto the wire. Offloading bypasses the operating system and reduces the load on the CPUs.

6.4.2 Infiniband Verbs and Scatter/Gather Elements

As previously described in section 3.4.1 about Infiniband, work requests are placed onto a queue pair and can be categorized in send and receive work requests. When the request processing is completed, a *work completion* (WC) entry can optionally be placed onto a *completion queue* (CQ), which is associated with the work queue.

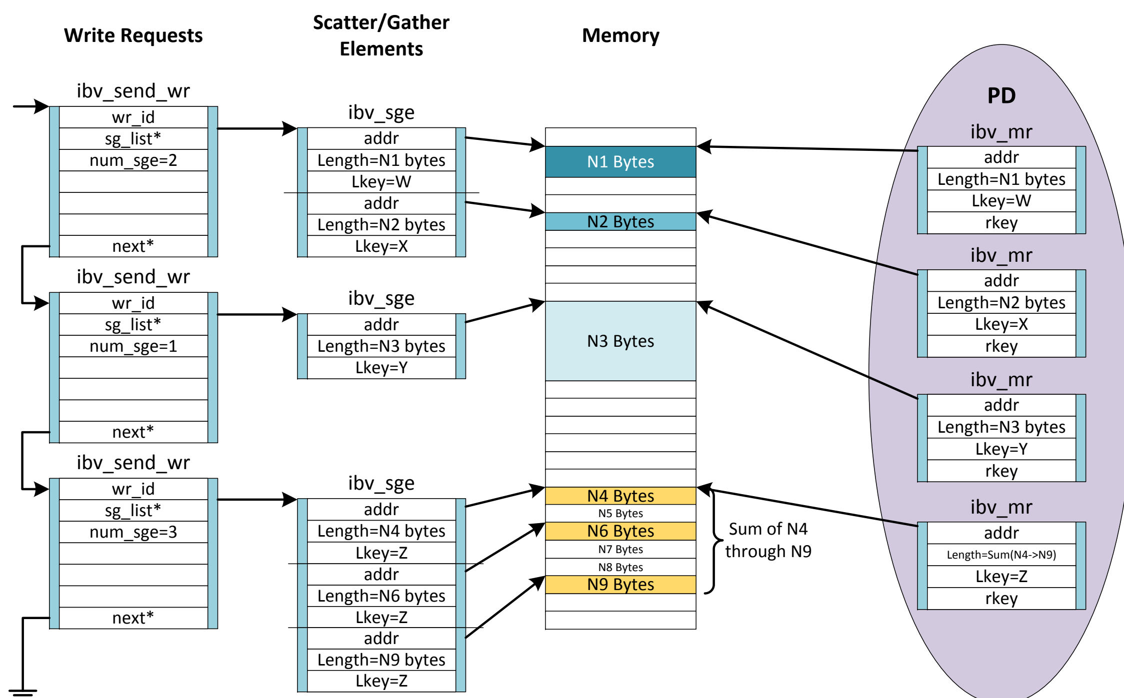


Figure 6.12: Relation of work requests, scatter/gather elements, main memory, memory regions and protection domain [176].

Scatter/Gather Elements (SGE) are used to define the memory address to write to or read from and are associated with a work request. An SGE is a pointer to a memory region, which has been pinned through a protection domain (PD) and can be accessed by an HCA for read and write operations. Typically, a memory region is a contiguous set of memory buffers, which have been registered with an HCA. The registration of a memory region causes the OS to provide the HCA with the virtual-to-physical mapping of that particular region, but also pins the memory, which means that the OS cannot swap the memory out onto secondary storage. The successful memory registration among other things returns two objects called `Lkey` and `Rkey`, which need to be used when accessing memory regions. The key pair provides authentication means. The `Lkey` (local key) can be used to access local memory regions, while the `Rkey` (remote key) needs to be sent to remote peers, so that they can directly access a local memory region through RDMA operations. As already mentioned, a memory region belongs to a protection domain, which provides an effective bonding between QPs and memory regions. PDs can be seen as an aggregating entity. Figure 6.12 presents a detailed overview of the relation between work requests, SGEs, main memory, and protection domains.

HCAs have an on-chip scatter/gather DMA controller that enables the gathering of data (page lists and block lists) out of the memory onto the wire in a single DMA

transaction. This means that scatter/gather I/O can be completely offloaded to the HCA. This feature is utilized by the O2IB LND for bulk data transmissions.

6.4.3 Scatter/Gather DMA Operation Support for Extoll

Recall from section 3.2.5 that the ATU acts as an MMU for the Extoll NIC, especially for the RMA unit, and therefore, is suitable to provide scatter/gather support for RDMA operations through the RMA unit. By default, the ATU kernel module allocates 128 GATs with an NLP size of 4 KB. Each GAT can map up to 2^{18} NLPs, which translates to $2^{18} * 4$ KB = 1 GB of mappable main memory per GAT. The ATU provides address translation offloading for two types of memory regions: pages lists and continuously allocated kernel virtual buffers.

The payload of an LNET message is limited by the LNET MTU, which is 1 MB per transmission, and can comprise of up to 256 pages with a page size of 4 KB. The payload is described by a memory descriptor, which points to an associated buffer that is allocated utilizing GFP (get free pages) flags. The GFP flags ensure that the returned buffer consists of pages, but do not pin the buffer into the main memory. The buffer either consists of an array of pages (array of `lnet_kiov_ts`) or a continuously allocated kernel virtual buffer of type `struct kvec`, which can be translated into a scatterlist consisting of pages. This means that the LNET design aligns well with the capabilities of the ATU design, which indicates that for Lustre the address translation can be completely offloaded to the Extoll NIC.

Within the scope of this work, the Extoll kernel API has been extended to provide ATU memory registration services for kernel modules such as LNDs. There are two functions available for memory registration, one for scatterlists and one for page lists. Both return a software NLA, which can be used to build RMA software descriptors. In addition, a de-registration function has been implemented, which expects the software NLA, the corresponding VPID, and the number of mapped pages as parameters. For example for page lists, the ATU expects the following input modifiers in order to perform a correct address translation: the VPID of the kernel process, the pointer to the first entry in page list of type `struct page`, the number of list entries, and the number of bytes (also called stride) that need to be added to reach the next entry of type `struct page` in the list.

When using the Extoll kernel API to register memory regions, the requesting process needs to make sure that the provided buffer is pinned into main memory so that the buffer can not be exempt by the OS and be swapped out onto secondary storage. This is a necessary requirement when working with physical addresses.

When memory is swapped out before the RMA operation gets completed, the address translation results in a general protection error, which ultimately leads to a compute node failure.

In general besides the support for page lists and continuous buffer space, it is desirable to provide scatter/gather DMA operations for block lists. However, the current ATU design does not support address translation offloading for such physical buffer lists. It requires an additional piece of kernel code, which handles such buffer types. One idea is to copy memory blocks into continuously allocated buffer space, e.g., for small fragments, and then, map this buffer to an NLA.

6.5 Support for LNET Protocol Semantics

The mapping of the LNET protocol semantics onto the Extoll technology comprises of two components: the data transmission protocols themselves and a message matching mechanism on both the initiating and the target side. The following sections provide the specification of the data transmission protocols followed by the description of how incoming messages are matched with their corresponding LNET messages.

6.5.1 Data Transmission Protocols

EXLND distinguishes two types of transmission protocols: *immediate sends* and *bulk data transfers*. The transmission protocol is chosen depending on the message type, but also the size of the payload.

6.5.1.1 Immediate Send

Similar to the eager protocol of EXT-Eth presented in section 5.4.1.2, the immediate send splits the payload of an LNET message in 120 byte-sized fragments and sends the data directly through multiple VELO sends. The immediate send is used for messages of type `ACK`, which typically have no payload attached, and fit in one VELO message. It can also be used for messages of type `PUT` or `GET` for small payload sizes. A user-tunable threshold, similar to EXT-Eth, defines the upper limit for the immediate send path. On the receiving side, the VELO message containing the last fragment of the payload notifies EXLND that all data has been received, re-assembles the payload, and then, passes it to the upper layers.

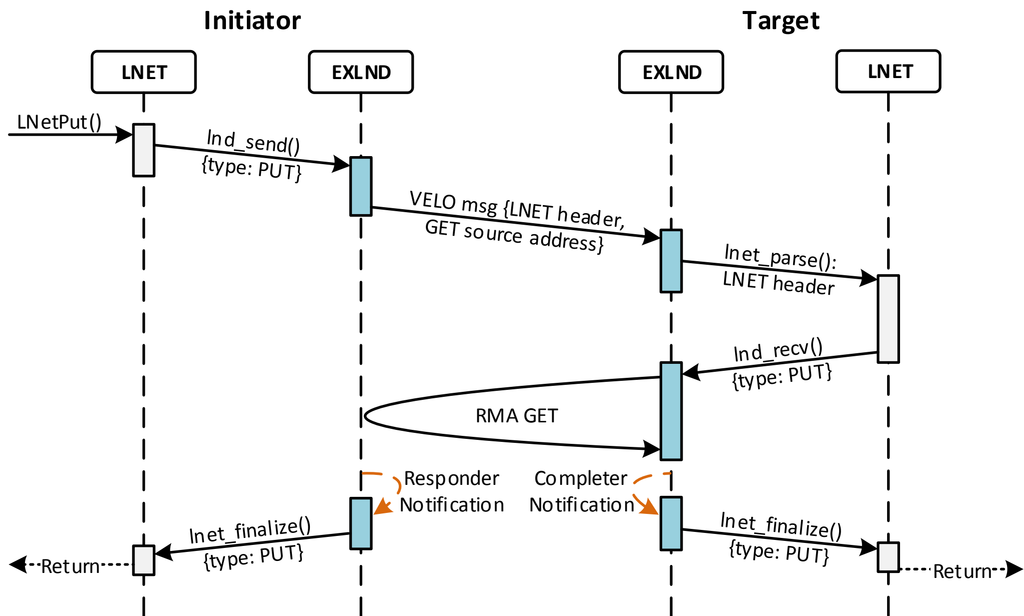


Figure 6.13: Overview of LNetPut() communication sequence over Extoll.

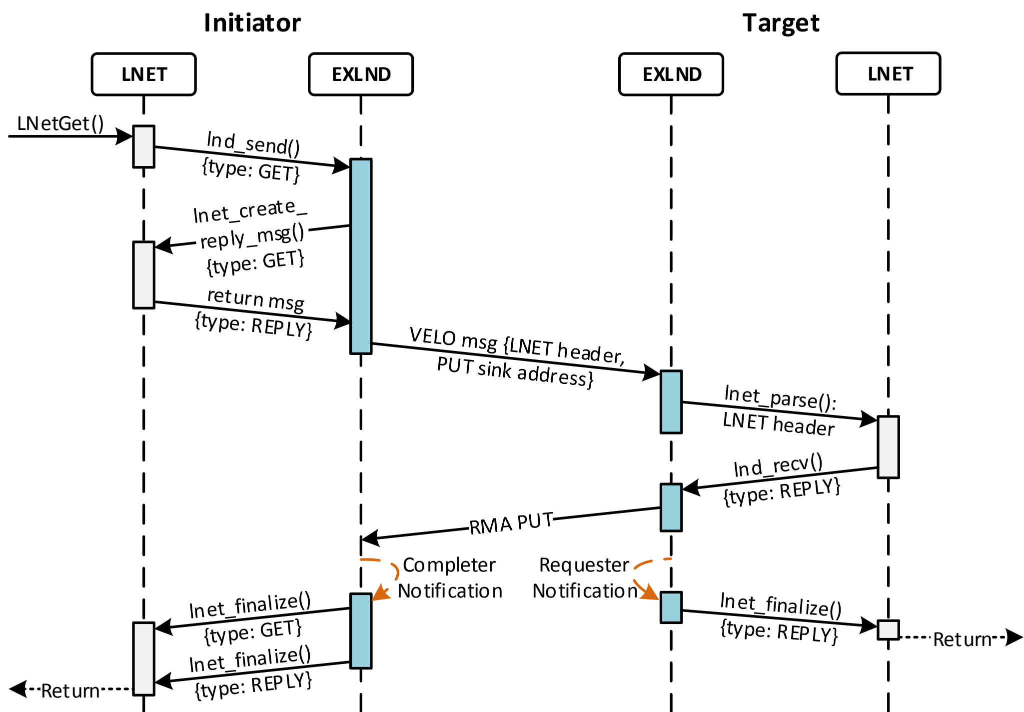


Figure 6.14: Overview of LNetGet() communication sequence over Extoll.

6.5.1.2 Bulk Data Transfer

In general, a send path can be divided in two steps: the setup and registration of the data buffers, and the transmission itself. One of the main design goals of EXLND is to minimize the number of LND-internal messages that need be exchanged in order to perform an `LNetPut()` or `LNetGet()`. This is achieved by internally interchanging the RMA operations performed by EXLND. In case of a `PUT` message, EXLND performs a rendezvous GET in order to read the data from the initiating node and write it directly to the target node. In case of a `GET` message, EXLND performs a rendezvous PUT, which writes the payload directly to the data sink on the initiating node. In this context, rendezvous means that the RMA operation is advertised through a VELO message from the initiating node.

Rendezvous GET Protocol

As described in section 6.2.2, an `LNetPut()` triggers the transmission of a `PUT` message, while the `ACK` message is optional. The `PUT` message has a memory descriptor attached, which describes the data to be sent over the network. Typically, a client needs to send a PUT request to the destination node. Once the target has allocated the PUT sink, the target informs the initiator of the availability and the PUT operation can be performed. For Extoll, EXLND maps the `PUT` source buffer to an NLA. If the payload is provided in form of a page list, EXLND only needs to make sure that the buffer is pinned, and then, can directly utilize the kernel API to map the buffer to RMA-addressable memory. For a buffer of type `kvec`, EXLND needs to translate the buffer into an array of pinned pages, before it registers the memory.

A `PUT` message is transmitted through the rendezvous GET protocol. This means that the payload of the LNET message on the initiating node acts as a GET source buffer for the target side. First, the GET source address and LNET header are sent to the target node by encapsulating the information into a VELO message. On the target side, `lnet_parse()` calls EXLND's receive function, which in turn initiates the RMA GET operation. The beauty of this approach is that there is no further communication required between the participating peers. Upon completion of the GET operation, the RMA unit generates two notifications, one on the initiating and one on the target node. On the initiating side, an RMA responder notification notifies EXLND that the send operation has succeeded, which results in the finalizing of the LNET message. On the target side, an RMA completer notification informs EXLND of the successful data transmission and the LNET message can be finalized. Figure 6.13 outlines the complete `PUT` sequence.

Rendezvous PUT Protocol

When the send function is called by `LNetGet()`, EXLND receives a message of type `GET` from LNET. `GET` messages have no data attached, but encode the information about the GET sink buffer. EXLND maps this buffer to an NLA, and the rendezvous PUT protocol is used to transmit the data to the target node. This means that when the target parses the incoming header of the `GET` message, EXLND's receive function receives a `REPLY` message, which provides the data source information for the transmission. The source buffer is also mapped to an NLA and the receive function triggers an RMA PUT operation to the initiating node.

Once the transmission has successfully completed, two RMA notifications are written. On the initiating side, a completer notification informs EXLND that the data has been received. On the target node, a requester notification informs EXLND that the PUT operation has been performed. On both sides, the notification is used to find the corresponding LNET message. The NLAs are de-registered and the LNET messages can be finalized.

6.5.2 Message Matching and Descriptor Queues

Upon the completion of a bulk data transmission, LNET has to be notified about the completed transfer by finalizing the corresponding LNET message through `lnet_finalize()`. As previously described, EXLND utilizes Extoll's RMA notification mechanism. These notifications can be associated with different EXLND events on the initiating and target nodes, and used to implement a message matching mechanism. In accordance with Figures 6.13 and 6.14, the following EXLND events can be distinguished:

LNetPut Done – Initiator: The payload of the `PUT` message has been sent and the initiating node can de-register the associated software NLA. Since the data transfer has been performed through an RMA GET from the target node, a *responder notification* is generated on the initiating node.

LNetPut Done – Target: The payload of the `PUT` message has been received and the target node can de-register the associated software NLA. Since the data transfer has been performed through an RMA GET from the target node, a *completer notification* is generated.

LNetGet Done – Initiator: The data has been received from the target node and the associated software NLA can be de-registered. Since the data transfer

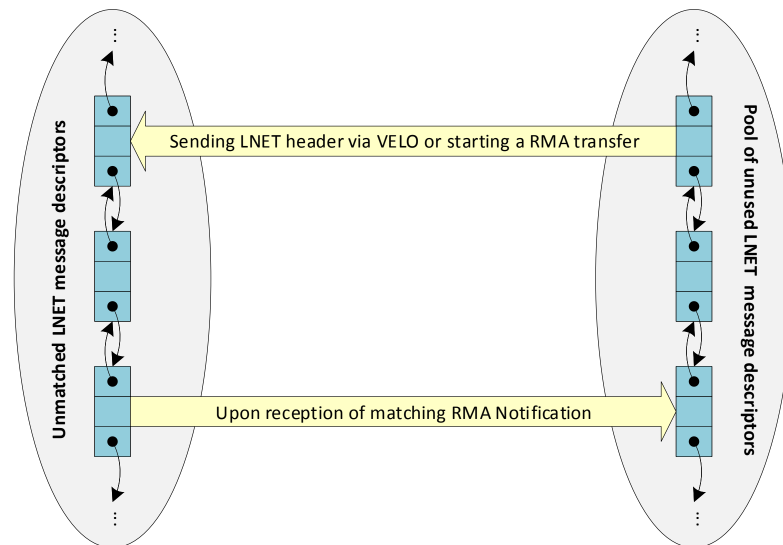


Figure 6.15: The movement of list elements representing unfinished transfers.

has been performed through an RMA PUT from the target node, a *completer notification* is generated on the initiating node of the `GET` message.

LNetGet Done – Target: The data has been written to the initiating node of the `GET` message and the target node can de-register the associated software NLA. Since the data transfer has been performed through an RMA PUT from the target node, a *requester notification* is generated.

To match incoming RMA notifications with pending LNET messages, EXLND defines three descriptor queues, with each of them corresponding to one of the three notifications types. Each peer (clients and servers) implements these queues.

Recall from section 5.4.2.1 that RMA notifications contain a field with the start address (physical or NLA) of the last read or write performed by the RMA unit. Extoll divides RMA transfers larger than the RMA's MTU in multiple frames of the size of the RMA MTU. This fragmentation is performed by the requester, which means that the requester knows the original start address of a read or write operation. The other two RMA subunits, responder and completer, only know where the read or write of the last frame has started, which means that the corresponding notification contains the start address of the last frame. This address needs to be calculated on each node perform initiating the bulk data transmission. Depending on the expected EXLND event, this either happens in EXLND's send or receive function.

Since EXLND knows all possible event types and their expected RMA notifications, EXLND adds the pending LNET message to the corresponding descriptor queue and connects it with the address that is expected to be seen in the RMA notification.

```
1 # add to /etc/modprobe.d/lustre.conf
2 options lnet networks="ex(ex0)"
```

Listing 6.4: Lustre configuration for EXLND.

For the address calculation, the equation presented in section 5.4.2.2 can be reused. Afterwards, EXLND initiates the data transfer by sending the header of the LNET message through VELO or performing an RMA operation. When EXLND receives a new RMA notification, the corresponding descriptor queue is searched. When the address can be matched, EXLND has found the corresponding LNET message and the message is finalized it through `lnet_finalize()`.

The descriptor queues are realized as double linked lists. At module startup time, EXLND initializes a pool of empty message descriptors. When a new transfer is started, EXLND's send and receive functions search for an empty entry, initialize its values, and put it into the correct descriptor queue, depending on the expected RMA notification. After an LNET message has been freed, the corresponding list item is moved back into the pool of free descriptors. This is done to avoid allocating and freeing memory for every send and receive, which would be rather time-consuming compared. Figure 6.15 illustrates the movement of list elements.

6.6 EXLND: Extoll Lustre Network Driver

Within the scope of this work, a prototype implementation of the *Extoll Lustre Network Driver* (EXLND) is realized in form of the kernel module `kexlnd.ko`, which utilizes Extoll's kernel API to interface the Extoll NIC. The prototype is compatible with Lustre versions 2.8 and 2.10, which are currently the most widely deployed Lustre releases. EXLND implements the mandatory function pointers, as described in section 6.2.1, and register the LND type `EXLND` in LNET's subsystem.

The current version of EXLND implements the immediate send protocol for messages of `ACK`. For all other message transmission, the module relies on the rendezvous PUT and rendezvous GET protocols, as presented in section 6.5.1.2. To support efficient bulk data transfers through scatter/gather DMA, EXLND utilizes the address translation mechanism introduced in section 6.4.3 and maps the payload buffers to software NLAs. The default LNET credit configuration is as follows: TX credits = 256, peer credits = 8, and peer router credits = 0. This is the recommended standard configuration for LNET credits. To configure EXLND as an LNET interface, a Lustre configuration file needs to be created in the normal `modprobe` configuration

options directory (typically `/etc/modprobe.d/`), which contains an entry for EXLND. Listing 6.4 displays the line that configures EXLND as an LNET interface with the name `ex`. The current state of the prototype is able to configure and mount a small scale Lustre file system.

6.7 Preliminary Performance Results

This section provides a preliminary performance evaluation of the prototype implementation of EXLND. The LNET self-test facility is leveraged to simulate Lustre metadata and I/O traffic over EXLND.

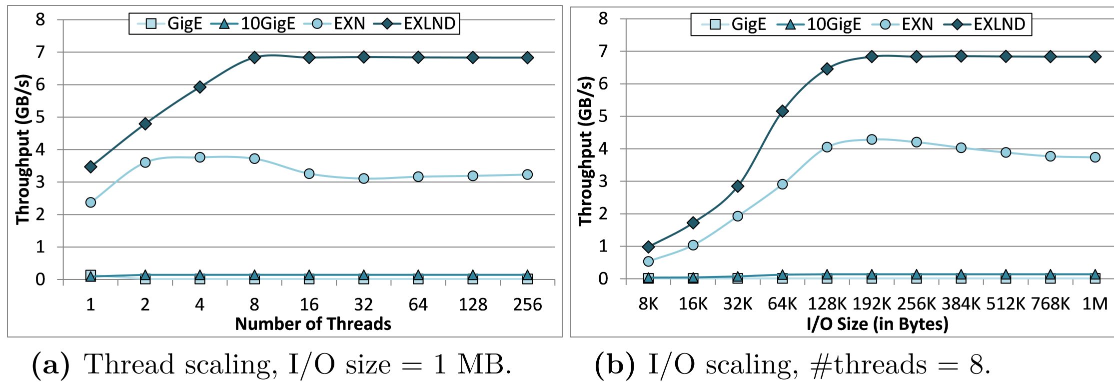
6.7.1 System Setup and Methodology

The Haswell and Skylake test systems presented in section 5.6.2 are utilized for an initial evaluation of EXLND. The systems run CentOS Linux version 7.3.1611 with kernel release 3.10.0-514.26.2.el7. In order to compile EXLND, a self-compiled Lustre version 2.10.1 is installed on the system nodes.

Testing a self-written LND is not an easy task. The main reason for this is that an LND lives inside of LNET. The functionality of an LND and LNET are tightly coupled; almost every function implemented by an LND interfaces with the upper LNET layer. Therefore, it cannot be simply loaded and tested like a normal kernel module. For evaluation purposes, LNET provides its own testing facility called *LNET self-test* [168]. The LNET self-test is implemented as a kernel module, which runs on top of LNET and the Lustre network drivers. Its main purposes are to (1) test the connection ability, (2) run regression tests, and (3) test the performance for a given Lustre network. The self-test provides the following two benchmarks, which measure the bandwidth and the count of remote procedure calls (RPCs) per second for a configured LNET interface:

ping The ping benchmark generates short request messages, which in turn generate short response messages. It can be used to measure the latency and overhead for small messages, but also to simulate the Lustre metadata traffic.

brw The brw (bulk read write) benchmark can be used to transfer data from the target to the source (read benchmark) or from the source to the target (write benchmark) in bulk data transfers. The payload size can be configured through the `size` parameter. This test is used to measure the network bandwidth for a given LND and simulates Lustre I/O traffic of a mounted file system.



(a) Thread scaling, I/O size = 1 MB. **(b)** I/O scaling, #threads = 8.

Figure 6.16: LNET self-test: bulk read test simulating Lustre I/O traffic.

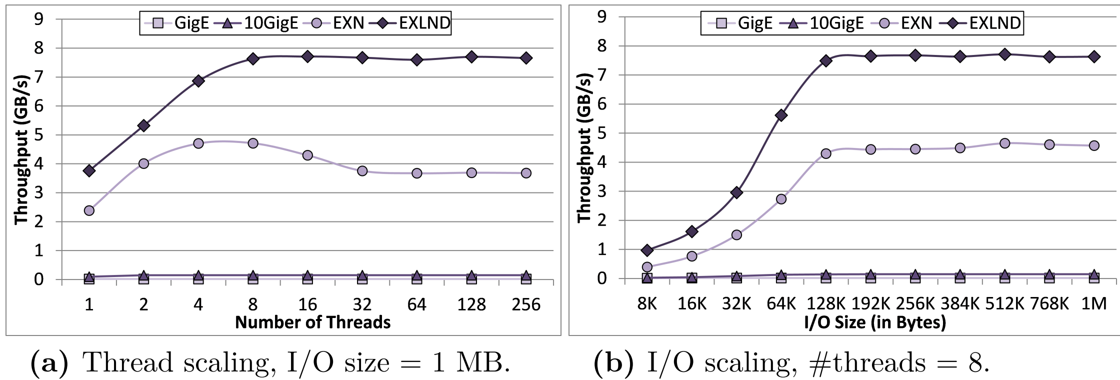
In general, LNET is not interested in the normal network latency. Recall that LNET was derived from the Portals API, and therefore, relies on the abstraction of remote procedure calls for communication purposes. Instead of sending a message, LNET calls a procedure on another node and the node responds in the same way. Therefore, LNET measures how many RPCs can be performed per second. However, the equivalent to an RPC is the count of ping packets that can be sent per second. Consequently, the inverse of the RPCs per second count can be interpreted as the half round trip latency through the LNET stack with latency $= \frac{1}{\text{RPCs}}$.

LNET provides an LND for TCP/IP networks. Therefore, it can be used out-of-the box for Ethernet networks, i.e., EXN can be used to configure a Lustre network. To facilitate a baseline, all benchmarks were performed over Gigabit Ethernet (GigE), 10 Gigabit Ethernet (10GigE), EXN, and EXLND.

6.7.2 LNET Self-Test Results

Figure 6.16 presents the results of the brw read benchmark. This benchmark simulates a client server communication where the client wants to read data from the server, and therefore, sends a read request. Internally, this means that the client sends a `PUT` message to the server, which in turn initiates the bulk data transfer from the server in form of a `PUT` message. Considering the design of the rendezvous GET protocol, which is used to transmit messages of type `PUT`, and the related communication overhead, EXLND performs surprisingly well, since EXLND has to perform two RMA GET operations to perform the communication sequence.

In Figure 6.16a, the payload size is fixed to 1 MB and the number of concurrently reading threads is scaled up to 256 threads. Starting with eight concurrent threads, the network is saturated and provides 6.8 GB/s of read performance. Figure 6.16b,



(a) Thread scaling, I/O size = 1 MB. **(b)** I/O scaling, #threads = 8.

Figure 6.17: LNET self-test: bulk write test simulating Lustre I/O traffic.

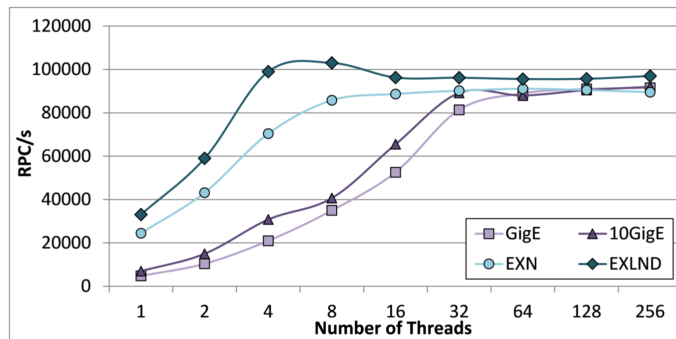


Figure 6.18: LNET self-test: ping test simulating Lustre metadata traffic.

on the other hand, presents the I/O scaling results. The number of concurrently reading threads is fixed to 16 and the size of the payload is scaled up to 1 MB. The limit of 1 MB is inflicted by the LNET MTU. Starting with a payload size of 192 KB, the network is saturated with a peak bandwidth of 6.8 GB/s.

Figure 6.17 shows the results of the brw write benchmark. In contrast to the read benchmark, the write benchmark simulates a client server communication where the client wants to write data to the server, and therefore, sends a write request in form of a `PUT` message to the server. In turn, the server initiates the bulk read transfer by sending a `GET` message to the client. In comparison to the read sequence, the write sequence comprises of an RMA GET and an RMA PUT operation, which introduces less overhead then two RMA GET operations. Figures 6.17a and 6.17b present the thread scaling and I/O scaling results for the write benchmark respectively. It can be seen that the peak bandwidth of about 7.7 GB/s is reached for eight concurrent threads starting with a payload size of 128 KB.

Figure 6.18 displays the results of the ping benchmark. As mentioned before, the ping benchmark measures how the number of RPCs that can be performed per second. While the payload size is fixed, the number of concurrent threads can be

configured for the benchmark. For EXLND, the maximum bandwidth is reached starting with four concurrent threads. An interesting observation can be made for one thread. As mentioned before, the the inverse of the RPCs per second can be used to calculate the half round trip latency for LNET. EXN achieves 245000 RPCs/s for one thread, which results in a latency of approximately 4.1 us. EXLND, on the other hand, performs 33100 RPC/s for one thread. This translates to a latency of about 1.5 us. Considering the microbenchmark results presented in section 5.6.3, these results align well with previously measured network latencies.

In addition to the benchmark results presented in this section, the same set of benchmarks has been performed on the Skylake system. As previously presented for EXN, the same performance degradation effects have been measured. Like EXN, EXLND riles on the spinlock mechanism of the Linux operating system, which introduces the latency of the PAUSE instruction every time a descriptor is en- or dequeued in one of the descriptor queues.

6.8 EXLND Summary

The contributions of this chapter were twofold. The first part focused on the analysis and summary of the Lustre networking protocol, which is an essential prerequisite to develop a custom LND. Even though Lustre is one of the most popular parallel file systems, its developer community is very small and most features are barely documented, especially from a developer's point of view. Therefore, providing a comprehensive overview is a valuable contribution to the entire HPC community.

The second contribution comprised the design and implementation of EXLND, which leverages the capabilities of Extoll to provide efficient storage connectivity in large-scale HPC deployments. One of the key characteristics of EXLND is the minimized communication overhead needed to transfer messages between networking peers. The innovative notification mechanism of the RMA unit can be used to reduce unnecessary RDMA sink or source availability advertisements, but also eliminates the need to send a completion message after successfully reading or writing data. In addition, the support of scatter/gather DMA operations has been added to the Extoll software stack, which enables efficient bulk data transmissions. This feature is needed to support the LNET message payloads of up to 1 MB in size.

The evaluation of the EXLND prototype implementation has demonstrated good performance characteristics. It can be seen that read requests initiated by a client perform slightly worse than write requests to a server. This is due to the protocol

overhead introduced by the nature of LNET. Another metric evaluated for EXLND is the RPCs per second count. The results indicate that a single thread has a half-round trip latency of 1.5 us. Given that this benchmark measures the latency including the complete LNET stack, this result is rather promising.

Parallel file systems provide great performance and scalability. However, projecting these to the user applications can be a challenge, mainly due to load imbalances and resource contentions of network and storage system components. The following chapter leaves the world of the Extoll interconnect and presents two user space tools, which attempt to resolve I/O contention in busy HPC environments, where multiple, concurrent applications compete for resources. The target system is the Titan supercomputer, which utilizes the Cray Gemini interconnect in a 3D torus topology. Therefore, it is expected that the presented results can be directly transfered to a large-scale system utilizing the Extoll technology, which also employs a 3D torus topology.

Chapter 7

Resource Contention Mitigation at Scale

While the previous chapters have evolved around the Extoll technology and focused on the improvement of network communication, this chapter discusses transparent load balancing and resource contention mitigation techniques from the client side. This research project was made possible through a collaboration between the Oak Ridge National Laboratory and the Computer Architecture Group.

Proportional to the scale increases in HPC systems, many scientific applications are becoming increasingly data-intensive, and parallel I/O has become one of the dominant factors impacting the large-scale application performance. The lack of a global workload coordination coupled with the shared nature of storage systems cause load imbalance and resource contention over the end-to-end I/O paths, which results in severe performance degradation. Efficient use of extreme-scale computing resources often requires extensive application tuning and code modification. This results in a steep learning curve for scientific application developers to understand the complex I/O subsystem, but also to address the I/O load imbalance and contention issues. Therefore, it is a major hurdle for applications to adopt and take advantage of any underlying improvements.

To ease this transition in the most transparent way, this chapter introduces two easy-to-use load balancing frameworks: *Aequilibro* and *TAPP-IO*. Both solutions rely on the topology-aware *Balanced Placement I/O* (BPIO) algorithm [177], which transparently balances data across all available storage system components. Aequilibro translates the benefits of BPIO into the platform-neutral middleware ADIOS. With this unification, ADIOS-enabled applications can effortlessly take advantage of BPIO's performance benefits without any further modifications. TAPP-IO (Trans-

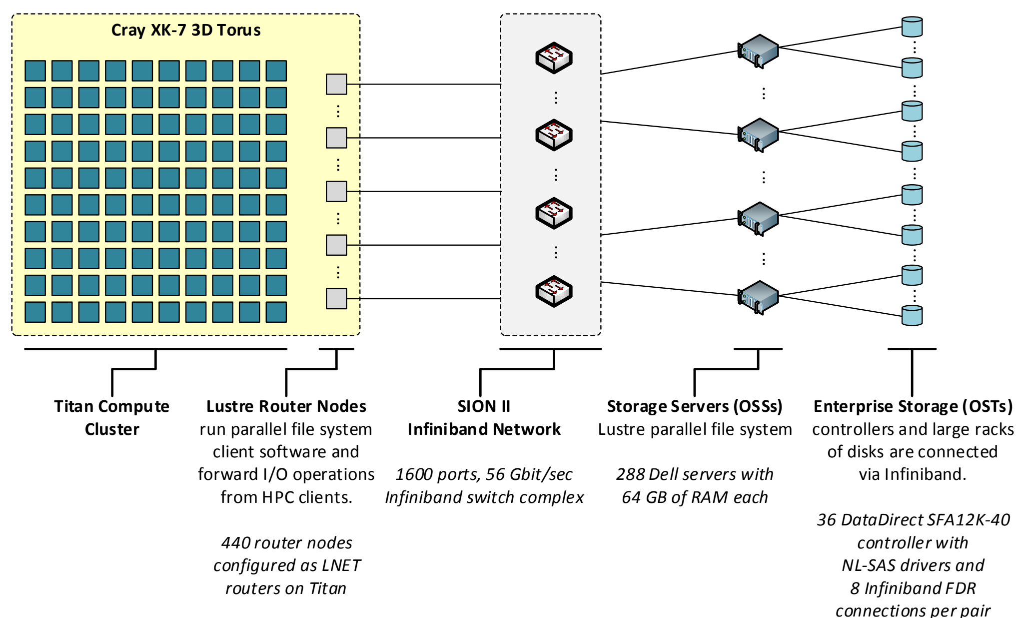


Figure 7.1: Infrastructure and I/O path between Titan and Spider II [177].

parent Automatic Placement of Parallel I/O) is a dynamic, shared load balancing framework, which transparently intercepts file creation calls during runtime to balance the workload over all available storage targets. The usage of TAPP-IO requires no application source code modifications and is independent from any I/O middleware.

The reminder of the chapter is organized as follows. The first part introduces the target and evaluation platform Titan, and provides some background information about observing load imbalance at large scales and performance tuning methodologies for center-wide parallel file systems. The main part introduces the I/O middleware Aequilibro and the high-level I/O library TAPP-IO. The last section concludes the chapter by presenting benchmark and evaluation results. This chapter summarizes and extends various contributions to international conferences [17, 19, 18].

7.1 Spider II – A Leadership-Class File System

This section presents an overview of the Titan supercomputer [178, 179] and Spider II [180] in order to facilitate an understanding of the target and evaluation platform. Titan is a hybrid-architecture Cray XK7 system with 18,688 compute nodes and 710 TB of total system memory. Each node contains both 16-core AMD Opteron CPUs and NVIDIA K20X Kepler GPUs. The high capability compute machine is backed by the center-wide parallel file system Spider II, which is based on the Lustre

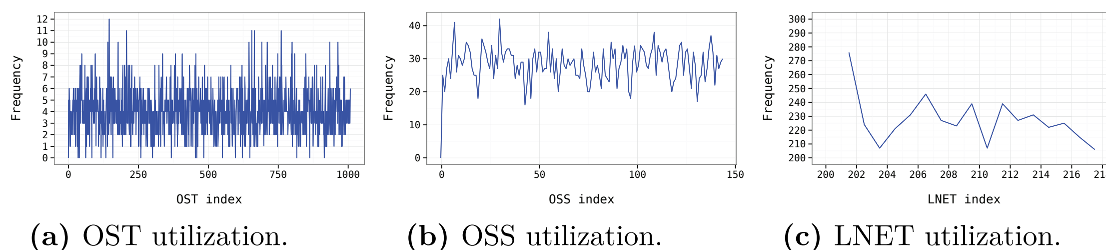
technology. Spider II is one of fastest and largest POSIX-complaint parallel file systems and is designed to serve write-heavy I/O workloads. Figure 7.1 shows the topology diagram, and in particular, the multi-layered end-to-end I/O path.

On the back-end storage side, Spider II has 20,160 Near-Line SAS disks organized in RAID 6 arrays. Each of these RAID arrays acts as an OST. The OSTs are connected to the OSSs over direct InfiniBand (IB) FDR links. At the time of this writing, a patched version of Lustre 2.8 was running on the I/O servers. The storage system is split into two distinct, non-overlapping sections, *atlas1* and *atlas2*, and each is formatted as a separate name space to increase reliability, availability, and overall metadata performance. Each file system has 144 OSSs and 1,008 OSTs. Each OSS is connected to a 36-port IB FDR top-of-the-rack (TOR) switch and two DDN controllers. Each TOR switch (36 in total) is connected with a total of 8 OSSs. Each switch also connects to two 108-port aggregation switches. The aggregation switches provide connectivity for the Lustre metadata and management servers.

On the front-end at the compute side, there are two different types of nodes on Titan: compute and Lustre I/O router, also known as LNET router, nodes. Both types of nodes are part of the Gemini network [129] connected in a 3D torus topology. Each node has a unique network ID (NID) for addressing purposes. 440 XK7 service nodes are configured as Lustre LNET router nodes. Of these, 432 are used for file I/O and 8 are used for metadata operations. Titan's I/O routers are connected to the Spider II TOR switches via InfiniBand FDR links. Note that the Spider II TOR switches enable these I/O routers to reach to the back-end storage system (OSSs and OSTs). By default, Lustre uses a round-robin algorithm to pick routers. The first alive router on top of the list will be picked to route the message and then will be placed at the end of the list for the next round to provide a load balance among multiple routers. Each Spider II TOR switch is assigned an LNET route.

7.2 The Need for Balanced Resource Usage

To cope with system failures, most scientific applications periodically write out memory states. These bursty writes, i.e., checkpointing data, can cause resource contention leading to hotspots which are detrimental to parallel application performance. Hotspots lead to variations in completion times across processes, and therefore, to a blocking behavior and wasted computational capacity. The rule of thumb for checkpointing is that it should not take more than 10% of the application run time per hour. As the HPC systems grow in scale, the cumulative memory size also grows,



(a) OST utilization. **(b)** OSS utilization. **(c)** LNET utilization.

Figure 7.2: Default resource usage distribution on Spider II [177].

resulting in larger amounts of data to be written out during each checkpoint window. This pushes the limits and capabilities of parallel file and storage systems; the growth trajectories for these two are not always the same. Most scientific HPC applications make no difference between defensive I/O and scientific output. In fact, they do I/O operations at regular pre-defined intervals and expect a minimal amount of time to be spent in I/O subroutines.

As can be seen in section 7.1, parallel I/O systems are inherently complex, particularly in the context of end-to-end I/O paths. For example at the starting point of a typical I/O path, an application can use a high-level I/O library, such as HDF5 [50], for various reasons including portability, improved data management, and enhanced metadata capabilities. HDF5 is implemented on top of MPI-IO [181] which, in turn, performs POSIX I/O calls against a parallel file system such as Lustre [166]. Furthermore before an I/O request reaches its eventual storage target, it may have to traverse through the compute fabric (e.g., a 3D torus or Dragonfly, or a plain fat-tree), and a large-scale storage network fabric (e.g., InfiniBand). Finally, the request reaches the backend storage that provides a block interface, though the parallel file system sits on top and across these backend storage devices. In large-scale HPC deployments, file systems are often shared among applications concurrently running on a single system and sometimes among applications running on multiple systems (e.g., a center-wide file system). This often results in file system and network contention. The observed I/O bandwidth at the application level is much lower than the theoretical peak bandwidth of the underlying storage system.

Resource contention has a negative impact on the performance and scalability in a large-scale file and storage system such as Spider II. A simple experiment [177] can be used to illustrate the need for balanced resource utilization. An application with 4096 processes is launched on Titan with each of the processes creating and performing a single write operation to an individual file on the parallel file system. Afterwards,

the striping pattern and file traces for every created file are evaluated, which can be used to determine the utilized OSTs, OSSs and LNET routers. Figure 7.2 presents the default Lustre resource usage distribution for OSTs, OSSs, and LNET routers, respectively. It can be observed that there is a significant variation in usage across different Lustre components, even though Lustre employs a round-robin resource allocation scheme. This variation and the resulting contention is due to the lack of a system-wide I/O load organizer on modern distributed large-scale HPC systems. Parallel file and storage systems supporting these large-scale systems only have a partial view of the overall I/O activity and can only try to optimize the resource usage at one end of the I/O path. Also, Xie et al. [182] observed that most I/O load imbalances are the result of improper resource allocation within a given scientific application, which makes this work particularly relevant.

7.3 Related Work

While moving towards the exascale era, resource contention and performance variability in HPC systems remain a major challenge. I/O workload imbalance paired with the increasing gap between compute capabilities of HPC systems and the underlying storage system are known issues in the HPC domain [183, 184].

Several research studies have addressed these problems to provide better I/O techniques. For example, Gainaru et al. [185] introduce a global scheduler that minimizes congestion caused by I/O interference by considering the application's past behaviors when scheduling I/O requests. Herbein et al. [186] present a job scheduling technique that reduces contention by integrating I/O-awareness into scheduling policies. As shown by Yildiz et al. [184], scheduler-level solutions not always lead to improved performance even though it helps to control the level of interferences.

Some research efforts consider network contention as the major contributor to I/O load imbalance. Luo et al. [187] introduce a preemptive, core-stateless optimization approach based on open loop end-point throttling. Jiang et al. [188] introduce new endpoint congestion-control protocols to address the differentiation between network and endpoint congestion more properly. Li et al. [189] present *ASCAR*, a storage traffic management system for improving the bandwidth utilization and fairness of resource allocation. But, these techniques cannot be adopted for scientific HPC systems like Titan. There are too many applications and the I/O patterns change drastically between different job runs.

Another area takes a file and storage-system centric view. Zhu et al. [190] present *CEFT-PVFS*, a modification of the PVFS file system [191], to achieve a better workload balance. Singh et al. [192] address the problem of load imbalance in the setting of cloud data centers. Congestion and load imbalance can still occur at large scale as demonstrated by Dillow et al. [193]. Luu et al. [194] analyze the problem of low I/O performance on leading HPC systems. They use Darshan [195] logs of over a million jobs representing a combined total of six years of I/O. Even though the platforms' file systems have a peak throughput of hundreds of GB/s, only few applications experience high I/O throughput. Lofstead et al. [196] introduce *Adaptive IO*, a set of dynamic and proactive methods for managing I/O interference. The methods are bundled in a new ADIOS transport method that dynamically shifts work from heavily used areas of the storage system to areas that are more lightly loaded. The design is limited on how quickly a coordinator can react to storage load dynamics. Liu et al. [197] introduce an ADIOS transport method that attempts to re-route I/O to less loaded storage areas while applying a throttling technique that limits how much data can be re-routed during writing.

A third area of related work is introduced by commercial data centers where load imbalance and *Quality of Service* (QoS) are also major concerns for multi-tenant systems. Such methods and techniques are exemplified by *Pulsar* [198], *Baraat* [199], and *Corral* [200]. Pulsar consists of a logically centralized controller with full visibility of the data center topology and a rate enforcer inside the hypervisor at each compute node. It offers tenants their own dedicated virtual data centers (VDC) to ensure end-to-end throughput guarantees. Baraat is a decentralized, task-aware scheduling system. It schedules tasks in a FIFO order, and avoids head-of-line blocking by dynamically changing the level of multiplexing in the network. Based on data from past data center studies, the application behavior is characterized in order to apply the task-aware scheduling heuristic. Corral is based on the assumption that a large fraction of production jobs are recurring with predictable characteristics. It uses characteristics of future workloads to determine an offline schedule which coordinates the placement of data and tasks.

In the scientific HPC context though, the application requirement and expectation, the highly specialized workload, and the architectural design and integration workflow present some unique challenges on leveraging techniques developed in cloud computing and data center settings [201]. Many of the assumptions made for commercial data center performance optimizations are not applicable to large-scale scientific simulation systems. For example, scientific HPC systems do not have a global view

on all available system resources and allocations. Also, storage and I/O systems have been shifting from a machine-exclusive paradigm to a data-centric paradigm where the mixed workload becomes a norm and much less predictable than before [202]. The scientific workloads itself differ from commercial workloads, e.g. search queries, data analytics jobs, and social news-feeds. Commercial and scientific applications have different requirements [201]. While commercial codes can be classified as high-throughput computing, scientific workloads are categorized as latency-sensitive, large-scale, and tightly coupled computations. They assume the presence of a high-bandwidth, low-latency interconnect, a parallel distributed file system shared between compute nodes, and a head node that can submit MPI jobs to all worker nodes.

7.4 Observations and Best Practices for File I/O

When working with parallel file systems such as Lustre, several different research studies and papers have addressed the question of how to optimize the file striping pattern. The following provides an overview of best practices and key observations.

(1) The *stripe alignment* [42, 203] of data can be critical. When using the file-per-process access pattern with a large number of files/processes, it is best to set the stripe count to one (i.e., no striping is used). For the single-shared-file access pattern, the stripe count should equal the number of writing processes (i.e., aggregators). This limits the OST contention and decreases the possibility that multiple processes communicate with the same OST at the same time.

(2) Besides stripe alignment, the size and location of I/O operations should be carefully managed. By mapping writing processes to specific servers, the number of concurrent operations on a single server can be reduced. This is especially helpful when file locks are handed out on a per-server basis (e.g., Lustre) to minimize the *file locking contention* [42, 204].

(3) The *I/O request size* (i.e., the transfer size) should be large [42], i.e., a full stripe width or greater. In general, large files, large transfers, and a larger number of I/O clients results in a larger aggregate bandwidth.

(4) When working with large data sets, it is recommended to *use high-level I/O libraries or middleware* such as ADIOS, HDF5, or SIONlib [42, 205]. They map application abstractions onto storage abstractions and provide portability, but also organize accesses from many processes. Also, MPI, not the file system, should be used for communication.

(5) Besides the file access organization from the application side, the compute system itself can play a key factor in limiting the overall performance, e.g., network congestion can lead to a suboptimal I/O performance. By identifying and eliminating hotspots, providing better *routing algorithms* and careful *placement of I/O routers*, network congestion can be mitigated [206].

(6) Even though a data-centric file system may have a high theoretical peak bandwidth, the real performance is what the users observe under file system congestion. For advanced users, it is a good practice to expose low-level infrastructure details in a programmable fashion, e.g., through a user library, to achieve better performance [177, 206].

Recapitulating the observations and best practices, the rule of thumb for Lustre file striping, in particular the stripe count, can be summarized as follows [207]:

- #files ≥ #OSTs → reduce the Lustre contention and OST file locking:

$$\text{stripe count} = 1 \tag{7.1}$$

- #files < #OSTs → utilize as many OSTs as possible:

$$\text{stripe count} = \left\lfloor \frac{\#\text{OSTs}}{\#\text{files}} \right\rfloor \tag{7.2}$$

- #files = 1 → maximize the parallel access performance:

$$\text{stripe count} = \begin{cases} \#\text{Aggregators}, & \text{if } \#\text{Aggregators} \leq \#\text{OSTs}; \\ \#\text{OSTs}, & \text{otherwise.} \end{cases} \tag{7.3}$$

7.5 End-to-End Performance Tuning

Over the past decade, the Oak Ridge Leadership Computing Facility has presented several different strategies to address system performance variability [206]. The following sections present two performance tuning methodologies, which address observations (5) and (6) introduced in section 7.4.

7.5.1 Fine-Grained Routing Methodology

The *fine-grained routing* (FGR) congestion avoidance methodology [193, 208] was first introduced with the Jaguar supercomputing system [209]. The system utilized two

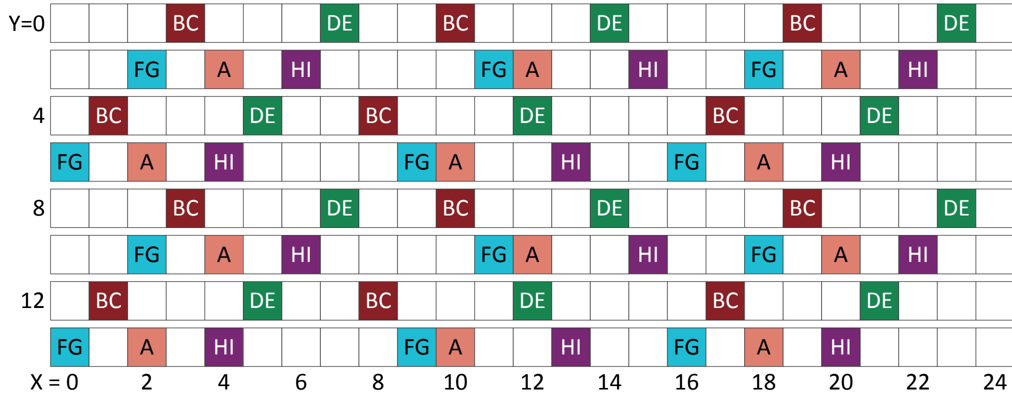


Figure 7.3: Topological XY representation of Titan's I/O routers [211].

different network technologies, the Cray SeaStar [210] and InfiniBand, which where target to network congestion leading to severely limited aggregate I/O performance. FGR employed two components to mitigate network contention. First, it paired topologically close clients and I/O servers, which avoided link saturation by reducing the load on common SeaStar torus links. Second, a new LNET router configuration was introduced, which assigned weights to different LNET routes based on the client to I/O server pairings. With the system upgrade from Jaguar to Titan, two key system components changed. The file system was upgraded to Spider II and the Cray SeaStar interconnect was exchanged with the Cray Gemini. The FGR method has been updated to better match the new system and to provide additional optimizations. For Titan and the Spider II file system, the fine-grained routing method comprises of two steps [211]. The first step is to place LNET router nodes as equidistant from each other as possible. This guarantees a fair I/O router distribution across the machine. The second step is to pair clients with their optimal LNET routers, which intends to minimize the end-to-end hop counts and congestion.

Recall that there are 440 LNET routers and 36 InfiniBand TOR switches. The switches are named based on their location in Spider II (row (*1-4*) and index within row (*a-i*)): *atlas-ibsw{1-4}{a-i}*. Out of the 440 LNET routers, 432 are used for file I/O and 8 are used for metadata operations. The I/O routers are equally distributed among the TOR switches with each switch providing 12 connections to Titan. The router naming scheme is based on the connected switch and index: *rtr{1-4}{a-i}-{1-12}*. Four I/O routers are grouped in a module which aligns well with the four rows of Spider II. Each of the nodes within a given module connects to the same "letter" (*A-I*) switch in each row. For the final router placement, the Spider II file system is further subdivided into four sections each serving two rows of Titan. This

Algorithm 2 Fine-grained routing algorithm [211]

1: **procedure** ROUTE SELECTION ALGORITHM (R^G, C)
2:
3: Divide R^G into 4 sub-groups: $R^G(1) \ldots R^G(4)$.
4: **for all** sub-groups $R^G(i)$ **do** ▷ i ranges 1 to 4
5: $C[y] \leftarrow$ y coordinate of C
6: $R_1^G(i) \leftarrow$ first router module in the ith sub-group
7: $R_1^G(i)[y] \leftarrow$ y coordinate of $R_i^G(S)$
8: **if** $(C[y] == R_1^G(i)[y] - 1)$
9: or $(C[y] == R_1^G(i)[y])$
10: or $(C[y] == R_1^G(i)[y] + 1)$
11: or $(C[y] == R_1^G(i)[y] + 2)$ **then**
12: break with sub-group i selected
13: **end if**
14: **end for**
15:
16: $i \leftarrow$ index of selected sub-group
17: $r_1, r_2, r_3 \leftarrow$ first, second, and third router module of selected sub-group i
18: $d_{min} \leftarrow \infty$
19: $Index_{primary} \leftarrow \infty$
20:
21: **for** j in $1, \ldots, 3$ **do**
22: $d_{current} \leftarrow \text{dist}(C[x], r_j[x])$ ▷ distance along X dimension
23: **if** $d_{current} < d_{min}$ **then**
24: $Index_{primary} \leftarrow j$
25: $d_{min} \leftarrow d_{current}$
26: **end if**
27: **end for**
28: primary router module $\leftarrow R^G_{Index_{primary}}(i)$
29: backup router modules $\leftarrow$ two other modules in the ith sub-group
30:
31: **return** <primary and backup router modules>
32:
33: **end procedure**

is necessary to limit the hops in Y-direction due to Gemini's limited bandwidth in this direction. For a given router group, e.g., *rtr{1-4}{a}-{1-12}*, 12 subgroups are assigned to the sections with three of each subgroup per section. The positions for router A are set manually while the subsequent routers are placed approximately $\frac{1}{3}$ around the X- and Z-dimensions. Figure 7.3 displays the final I/O router placement in Titan. Each box in the figure represents a cabinet, where X and Y denote the dimensions in Titan's 3D torus topology. Colored boxes represent cabinets containing

at least one I/O module [206]. Similar colors correspond to identical router groups. For a detailed description of the placement heuristic refer to Ezell et al. [211].

Algorithm 2 describes the second step of FGR, namely how a client chooses the optimal I/O router module for a given router group. In the FGR algorithm, $R_i^G(S)$ denotes the ith I/O router module in the Gth router group and S^{th} sub-group, with $i \in \{1,\ldots,4\}$, $G \in \{1,\ldots,9\}$, and $S \in \{1,\ldots,3\}$. The fine-grained routing algorithm needs two input parameters, the coordinates C, presented as (X,Y,Z), in the 3D torus of the client and the destination router group (R^G). It can be divided in two parts. First, the sub-group whose Y-coordinates are in close proximity with the input client (lines 4–14) is chosen. Second, the routers of the selected sub-group are returned. The one with the shortest distance to the input client is made the primary router, the other two serve as backups. The X-direction crosses cabinet boundaries. Therefore, it is desirable to minimize the hop count in the X-direction. For both the LNET router placement and the client-to-LNET router module assignment algorithm, various scripts exists to ensure that all nodes are cabled correctly and can communicate with each other. These scripts can also be used to generate mapping files containing the topology and routing information in X-, Y-, and Z-directions. These mapping files can be used to initialize the balanced placement I/O strategy.

7.5.2 Balanced Placement I/O Strategy

There are several possible approaches to address the end-to-end resource contention problem. One possibility would be the improvement of the Lustre resource allocation scheme. But, this would only address one end of the problem, and also, it would introduce a significant amount of LNET configuration overhead since every storage system comes with its own system design. Therefore, the possibility was disregarded.

Another way to achieve balanced resource utilization is the deployment of a centralized, system-wide I/O coordination and control mechanism. For example, *Fastpass* [212] is a network framework that aims for high utilization with zero queuing for data centers. For large-scale scientific HPC systems, this approach is not feasible. Multiple applications are running concurrently, with a variety of different I/O patterns and workloads. A system-wide I/O organizer requires the coordination between different subsystems (often designed and provided by different vendors), as well as scientific application code changes to participate in system-wide coordinated I/O. Therefore, even though it is possible to design a system-wide I/O coordinator, it is practically prohibited to be deployed. The system-wide I/O request coordination would lead to a tremendous communication overhead, and therefore, it would likely

Algorithm 3 Balanced placement algorithm [177]

```
 1: procedure BALANCED PLACEMENT (List of NIDs, List of OSTs)
 2:
 3:     lnet_freq ← 0, rtr_freq ← 0, oss_freq ← 0, ost_freq ← 0
 4:     random_offset ← randomly selected reachable LNETs
 5:
 6:     for all NIDs in the input NID list do
 7:         lnet ← random_offset
 8:
 9:         for all reachable OSTs do
10:             cost ← MAX
11:             oss ← ost2oss()                                  ▷ map OST to OSS
12:             mycost ← placement_cost(lnet_freq, rtr_freq, oss_freq, ost_freq)
13:             if (mycost < cost) then
14:                 mycost ← cost
15:                 picked_ost ← ost
16:                 picked_oss ← oss
17:             end if
18:         end for
19:
20:         record NID and the selected OST
21:         increment lnet_freq, rtr_freq, oss_freq, ost_freq
22:     end for
23:
24: end procedure
```

lead to a sub-optimal utilization of the available computational resources. The third approach is to balance the I/O workload on an end-to-end and per job basis.

This technique is adopted by the *Balanced Placement I/O* (BPIO) library [177], which uses a lightweight placement cost function that takes a weighted average of how frequently different file and storage system resources have been traversed by previous I/O requests issued by the same application. The most general case is defined by

$$C = w_1 R_1 + w_2 R_2 + ... + w_n R_n = \sum_{i=1}^{n} w_i R_i \tag{7.4}$$

where C is the cost of an I/O path, R_i is a resource component, and w_i is the relative weight factor with $\sum_{i=1}^{n} w_i = 1$. For Lustre, possible resource components are logical I/O routes (i.e., LNETs), or actual file system and networking resources (i.e., Lustre I/O routers, OSSs, and OSTs). BPIO intelligently allocates I/O paths for a parallel file system and employs a placement strategy that provides a binding between an I/O client (compute node or MPI rank ID) and a storage target for a given I/O

phase while aiming at an even I/O traffic distribution across resource components to avoid points of contention. This is ensured by taking the topology and resource dependencies into consideration. The library utilizes the mapping files generated by the previously described fine-grained routing congestion avoidance method. FGR organizes I/O paths to minimize end-to-end hop counts and congestion. This is done by pairing clients with their closest possible, and in the case of Titan, optimal LNET router. When the BPIO library is initialized, it parses the mapping files to retrieve a list of all available file system and storage components, but also extracts the routing information between different system components. For Lustre based systems, the placement cost function C is defined as shown in Equation 7.5.

$$C = w_1 \times \mathrm{rtr_freq} + w_2 \times \mathrm{lnet_freq} + w_3 \times \mathrm{oss_freq} + w_4 \times \mathrm{ost_freq} \tag{7.5}$$

rtr_freq, lnet_freq, oss_freq, and ost_freq are the usage frequency of I/O routers, LNETs, OSSs, and OSTs, respectively, and w_1, w_2, w_3, and w_4 are the relative weight factors. For the Spider II system, the weight factors are set to the following values: $w_1 = w_2 = w_3 = 0.2$, $w_4 = 0.4$. These values are chosen based on the observation that storage targets are typically more imbalanced than other storage components.

For a given I/O client, the BPIO library employs Algorithm 3, which loops over all reachable storage targets to choose one with the lowest placement cost per compute node. This is repeated for all active I/O processes of an application *only once* before it enters the I/O write phase. An initial performance evaluation of the library was performed on the Titan supercomputer [178].

7.6 Design Objectives and Strategy

Recapitulating the last section, it can be said that the BPIO library provides a lightweight, tunable placement mechanism, which utilizes a placement cost function that can be tuned to meet the target system. The introduced overhead is kept to a minimum since the balancing function is called only once per I/O phase and the internal data structures are kept as simple as possible. But, it is important to understand that the BPIO library simply returns a list containing the optimal binding between an I/O client and an OST for the next I/O phase. It does not provide a mechanism for the actual data placement nor file layout. The application developer has to modify the scientific code and needs to know how to set Lustre file properties in a C/C++ program using the *llapi* [168]. This is particularly difficult when taking different I/O interfaces and workload patterns into consideration.

In order to increase the application adaptability and transition, this chapter contributes two easy-to-use load balancing frameworks: *Aequilibro* and *TAPP-IO*. Both integrations rely on the topology-aware BPIO algorithm and combine the placement information with the key observations for optimal file striping patterns, as described in section 7.4. Also, they support three major I/O interfaces: POSIX I/O, MPI-IO, and parallel HDF5. This ensures a broad application compatibility, since the interfaces are widely used on current HPC systems. As pointed out by a recent evaluation of HPC system workloads [194], between 50% and 95% of jobs use the POSIX I/O API. The remaining jobs use either MPI-IO directly or high-level libraries built atop MPI-IO such as HDF5.

Aequilibro Aequilibro [17, 19] (derived from Latin: "keep in a state of equilibrium/balance") aims to unify two key approaches to cope with the imbalanced use of I/O resources. It utilizes the topology-aware BPIO method for mitigating resource contention and combines it with the advantages of the platform-neutral I/O middleware ADIOS, which provides a flexible I/O mechanism for scientific applications. But, Aequilibro does not support single shared file I/O. The design and implementation is described in section 7.7.

TAPP-IO TAPP-IO (*Transparent Automatic Placement of Parallel I/O*) [19, 18] is designed to work with parallel file systems and does not require any modifications to application or I/O library source code. It is implemented as a user space library and transparently intercepts file creation calls during runtime to balance the workload over all available storage targets.

The TAPP-IO framework works with both dynamically and statically linked applications, and proposes a new placement strategy to support not only file-per-process I/O, but also single shared files. This opens the door to a new class of scientific applications that can leverage the BPIO placement library for improved performance. An overview of the design and implemented is presented in section 7.8.

7.7 Aequilibro – An I/O Middleware Integration

Aequilibro is implemented as an optional feature in the file creation and write phase for selected ADIOS transport methods, and makes the following two contributions. First, ADIOS-enabled applications can effortlessly take advantage of BPIO's balancing strategy without any further modifications. Second, ADIOS provides a simple way to use multiple different I/O interfaces without the need for the user to explicitly

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <adios-config host-language="C">
3     <adios-group name="example">
4         <var name="var_test" type="integer"/>
5     </adios-group>
6     <method group="example" method="MPI_AGGREGATE">
7         num_aggregators=64; num_ost=512
8     </method>
9     <buffer max-size-MB="32" allocate-time="now"/>
10 </adios-config>
```

Listing 7.1: Example ADIOS XML configuration file.

adopt and integrate BPIO library calls. Therefore, it can be used as a simple, yet efficient evaluation framework for BPIO.

7.7.1 Transport Methods

ADIOS provides a mechanism to externally describe an application's I/O requirements by providing an XML-based configuration file. This results in a runtime selectable technique, which replaces I/O interface calls with basic ADIOS operations such as `adios_open()` and `adios_write()`. Depending on the specified transport method, a different application I/O pattern and I/O interface is used to access files. Listing 7.1 shows an example XML configuration file for the ADIOS framework.

Two example transport methods are `POSIX` and `MPI_AGGREGATE`. `POSIX` takes advantage of the concurrency of parallel file systems and implements the simplest parallel I/O pattern, known as file-per-process. Each writing process is responsible for writing data to its own output file. One of the processes is managing an index file. Overall, the transport method creates an index file along with a subdirectory containing all of the files written individually by the application's processes.

`MPI_AGGREGATE` is a sophisticated, MPI-IO-based technique derived from the `MPI` transport methods. Instead of using the default data aggregation model of the local MPI installation, it provides a user-tunable method, which aggregates data from multiple MPI processes into larger chunks of data before writing them out to the file system. A subset of application processes acts as an aggregator for a subset of peers. This pattern is also known as single shared file I/O with collective buffering. The number of aggregators (i.e., writing MPI ranks) and the number of OSTs are tunable, and can be set by the user at runtime. The method creates a collection of individual files, with one file corresponding to each of the aggregators. In addition, the user can provide Lustre-specific striping information, e.g., stripe count and size.

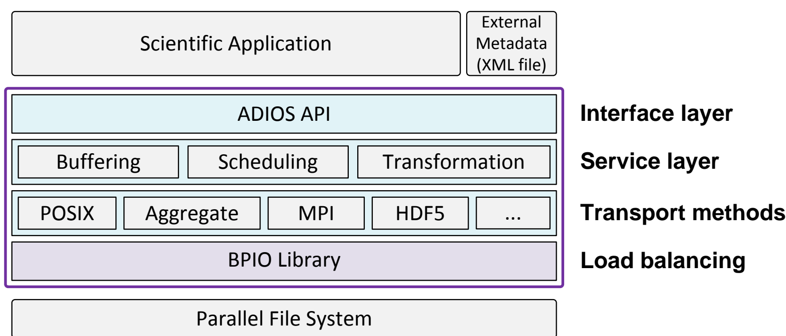


Figure 7.4: Aequilibro software stack.

7.7.2 Software Design and Implementation

Aequilibro aims to resolve the resource contention problem by integrating BPIO into ADIOS. The software architecture of ADIOS comprises of several layers, as depicted in Figure 7.4. The best way to combine BPIO and ADIOS is to select well-suited transport methods and implement the BPIO library calls directly into these methods. In doing so, the BPIO library acts as a shim layer between the transport methods and the parallel file system, as depicted by Figure 7.4.

The load balancing library is integrated in the transport methods `POSIX`, `MPI_AGGREGATE`, and `PHDF5`. There main reason for this decision is that when Aequilibro was initially designed, Spider II was still running Lustre version 2.4. Therefore, Aequilibro inherited the file placement and striping limitations of this release. Lustre 2.4 lacked the ability to provide fine-grained control of object placement. With Lustre release 2.7, a new feature [213] was introduced, which provides the user with the possibility to define a striping pattern for single shared files. This feature is leveraged by TAPP-IO (see section 7.8).

7.7.2.1 File Creation Interfaces

Listing 7.2 presents the standard POSIX calls to open and possibly create a file. Given a `pathname`, `open()` as well as `creat()` return a file descriptor, which is a small, nonnegative integer. The parameter `flags` describes the access mode and must include `O_RDONLY`, `O_WRONLY`, or `O_RDWR`, which either opens the file in read-only, write-only, or read/write mode, respectively. The parameter `mode` is optional and is only needed when the flag `O_CREAT` is specified. It describes the file access permissions. The Lustre user library *llapi* offers a similar API to open and create new files. Listing 7.3 displays the call syntax for `llapi_file_create()`

```
1 /* open, creat - open and possibly create a file */
2 int open(const char *pathname, int flags);
3 int open(const char *pathname, int flags, mode_t mode);
4
5 int creat(const char *pathname, mode_t mode);
```

Listing 7.2: Overview of file create and open calls.

```
1 /* llapi_file_open, llapi_file_create - open and possibly
2 create a file or a device on a Lustre filesystem */
3 int llapi_file_open(const char *name, int flags, int mode,
4 unsigned long long stripe_size, int stripe_offset,
5 int stripe_count, int stripe_pattern);
6
7 int llapi_file_create(const char *name,
8 unsigned long long stripe_size, int stripe_offset,
9 int stripe_count, int stripe_pattern);
```

Listing 7.3: *llapi* open and create calls.

and `llapi_file_open()`. `flags` and `mode` are similar to the POSIX API. The main difference between the calls is that llapi calls can be used to pre-create a Lustre file descriptor for a file with the provided striping information. The library allocates the file objects on the specified OSTs and creates a Lustre inode on the MDS before forwarding the call to `open()`.

7.7.2.2 Integration of BPIO with ADIOS

The ADIOS framework is initialized by a user application by calling `adios_init()`. The initialization of BPIO library is integrated in this call, which triggers the configuration of BPIO with the system-specific, FGR-based mapping files. Also, the list of allotted I/O clients is created. At the entry point of an I/O write phase, the BPIO library is invoked by ADIOS with the list of allotted I/O clients. Depending on the selected transport method, this takes place in the `adios_open()` or the `adios_groupsize()` call. The BPIO library returns a list with the I/O-client-to-OST assignment. Aequilibro uses this information to create the file descriptor, as described in Algorithm 4.

For POSIX I/O, a file is created with Lustre's *llapi* library. Internally, the stripe size, stripe count, and start OST of a file are set via the logical object volume (LOV) layer and stored in the data structure `lov_user_md`. LOV handles the client access to OSTs. Following observation (1) from section 7.4, the stripe count is set to one, which limits the OST contention. In the end, the striping information is applied by

Algorithm 4 Data placement algorithm: file-per-process pattern

```
1:  stripe_count ← 1, stripe_size ← 1 MB, fd ← 0
2:  stripe_pattern ← LOV_PATTERN_RAID0
3:
4:  clients ← Balanced Placement (NIDs, OSTs)        ▷ Update NID/OST binding
5:  ost_offset ← number of available OSTs
6:
7:  if (clients[my_rank] ≥ ost_offset) then
8:      ost ← clients[my_rank] − ost_offset
9:  else
10:     ost ← clients[my_rank]
11: end if
12:
13: fd ← llapi_file_open(filename, stripe_size, ost, stripe_count, stripe_pattern)
14: if (transport_method == MPI_AGGREGATE) then
15:     close(fd)
16:     Invoke MPI_File_open()
17: else if (transport_method == PHDF5) then
18:     close(fd)
19:     Invoke H5Fcreate()
20: else
21:     Invoke open()
22: end if
```

calling the function `llapi_file_open()`. For MPI-IO and HDF5, the file creation is divided in two steps. A file is pre-created as described for the POSIX case, and afterwards, it is opened with `MPI_File_open()` or `H5Fcreate()`, respectively.

As described by Wang et al. [177], the algorithm used by BPIO is sensitive to the size of the possible resources and routing paths. When the number of I/O requests is small and tightly packed in close proximity, a set of suboptimal OSTs might be used.

7.8 TAPP-IO Framework

TAPP-IO provides users with a transparent auto-tuning framework that takes full advantage of the optimizations done at the interconnect level, i.e., by utilizing FGR, and the load balancing done at the file system level through BPIO. The framework transparently intercepts file I/O calls (metadata operations) during runtime to distribute the workload evenly over all available storage targets. TAPP-IO's design incorporates all of the observations presented in section 7.4. The following sections describe the design and implementation of the runtime environment.

7.8.1 Parallel I/O Support

The TAPP-IO framework proposes a file placement strategy that supports both file-per-process and single-shared-file I/O access patterns, as described in section 2.4.4. File-per-process scales the single writer I/O pattern to encompass all processes instead of just one process. Each process performs I/O operations on their own separate file. Thus, for an application run of N processes, N or more files are created. In a single-shared-file application I/O pattern, multiple processes perform I/O operations either independently or concurrently to the same file. The possible HPC application I/O patterns can roughly be classified as follows (the ratio can be read as *writer count* : *file count*):

(1) $N : N$, stripe count ≥ 1;

(2) $N : M$, stripe count ≥ 1, $M > N$;

(3) $N : M$, stripe count ≥ 1, $M < N$;

(4) $N : 1$, stripe count > 1.

Case (1) and (2) describe the file-per-process I/O patterns, case (3) presents a strategy where the writing is aggregated in M shared files, and case (4) is the single-shared-file strategy where multiple clients write to multiple ranges within the same file. With a *file-per-process* I/O pattern, it is best to use no striping (stripe count of 1), as presented in section 7.4. This limits the storage target contention when dealing with a large number of files and processes. Therefore, case (2) is disregarded for the TAPP-IO framework. Case (3) is a special case of case (4) where multiple writers are aggregated in multiple shared files. TAPP-IO currently supports cases (1), (3), and (4) with the limitation that the balancing algorithm needs the expected file size for the shared files. For *single shared files*, TAPP-IO tries to minimize both the overhead associated with splitting an operation between storage targets (i.e., stripe alignment) and contention between writing processes over a single storage target (i.e., file locking contention). Figure 7.5 displays two possible shared file layouts. The *segmented file layout* keeps the data from a process in a contiguous block, while the *strided file layout* strides the data throughout the file.

When accessing a single shared file from many processes, the stripe count should equal the number of processes (refer to section 7.4). The size and location of I/O operations from the processes should be carefully managed to allow as much stripe alignment as possible resulting in each writing process accessing only a single storage target. Analogous to the file-per-process pattern, the algorithm follows the placement

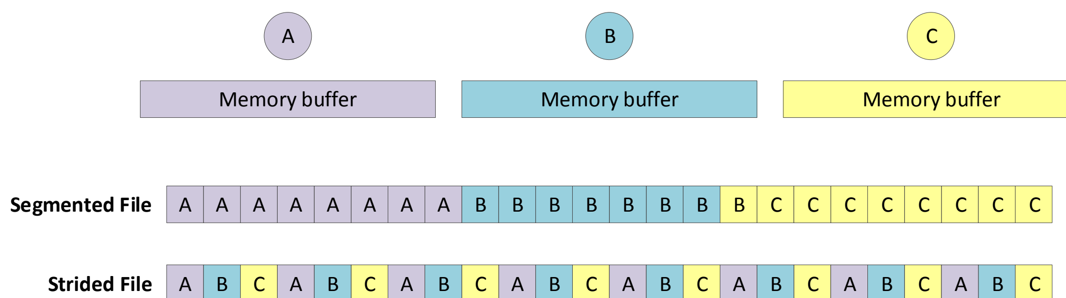


Figure 7.5: Shared file layout patterns.

strategy implied by the segmented file layout. A naive approach would be to set the stripe count to the number of writer processes and the stripe size to $\frac{\text{file size}}{\text{stripe count}}$. But for Lustre, the stripe size needs to be an even multiple of 65536 bytes. Equation 7.6 describes how the file size can be aligned to match an even multiple of 65536 bytes.

$$\text{aligned filesize} = \left\lceil \frac{\text{real filesize}}{65536 \times 2 \times \text{number of writers}} \right\rceil \text{ (in Bytes)} \tag{7.6}$$

With these parameters, the algorithm tries to achieve high levels of performance while mitigating storage targets contention at large process counts. The placement algorithm is invoked only once for every I/O write phase which adds minimal overhead. The optimal set of storage targets is determined similarly to the balanced placement algorithm introduced in Algortihm 3. For each stripe of a shared file, the optimal stripe-to-storage-target assignment is calculated by utilizing of the BPIO placement cost function.

Algorithm 5 displays a simplified version of the TAPP-IO balancing algorithm. The implementation of the TAPP-IO balancing framework has been deployed and tested on Titan's Spider II file system. In order to specify the striping information for the file-per-process strategy, it is sufficient to set a file descriptor's Lustre striping information via the *llapi* library before opening the file via the corresponding I/O interface (`MPI_File_open()` or `H5Fcreate()`). `llapi_file_create()` allows a client to specify the stripe size, stripe count, and OST offset of a file via the logical object volumn manager. When MPI-IO or HDF5 tries to create a file, the I/O layers transparently adopts the Lustre file descriptor that was previously created. The existing descriptor is used by Lustre when opening the corresponding file.

With Lustre 2.7 [213], a new feature was introduced that provides the user with the ability to explicitly specify the striping pattern via an ordered list of OSTs. The Lustre *llapi* can be utilized to specify the Lustre striping parameter struct

Algorithm 5 TAPP-IO Balancing Algorithm (simplified)

```
 1: /* I/O call, e.g. open(), triggers balancing */
 2: osts ← Balanced Placement (NIDs, OSTs)          ▷ Update NID/OST binding
 3:
 4: /* Determine placement parameters */
 5: if (File-per-process) then
 6:     ost ← osts[my_rank]
 7:     size ← 1 MB
 8:     count ← 1
 9:     pattern ← LOV_PATTERN_RAID0
10: else if (Single-shared-file) then
11:     struct striping_info param ← 0
12:     param->stripe_count ← number of writing processes
13:     param->stripe_size ← aligned filesize / stripe count
14:     param->osts ← osts                    ▷ Array with stripe_count entries
15:     param->stripe_offset ← param->osts[0]
16:     param->striping_is_specific ← true
17:     param->stripe_pattern ← 0
18: end if
19:
20: /* Initialize Lustre file descriptor via llapi */
21: fd ← 0
22: if (File-per-process) then
23:     fd ← llapi_file_create(filename, size, ost, count, pattern)
24: else if (Single-shared-File) && (my_rank == 0) then
25:     fd ← llapi_file_open_param(filename, flags, mode, param)
26: end if
27:
28: close(fd)
29: return ___real_function()
```

`llapi_stripe_param` (denoted as struct striping_info in Algorithm 5) where a list of OSTs can be passed to the LOV manager. Unlike the file-per-process strategy in this example, `llapi_file_open_param()` is called by MPI rank 0 to create the Lustre file descriptor. This is a necessary restriction, since only one process can apply the striping pattern to create the Lustre inode. If the inode has already been created, the API returns a Lustre error. The TAPP-IO library returns a list of MPI rank ID to OST assignments which is used to specify the striping pattern. Currently, the balancing algorithm for single shared files needs the expected file size from the application in order to match the stripes with the writing processes. Via `MPI_Info_set()`, the application can forward the file size by specifying a value pair `(fileSize,value)`. To utilize TAPP-IO for multiple-shared files, the framework also

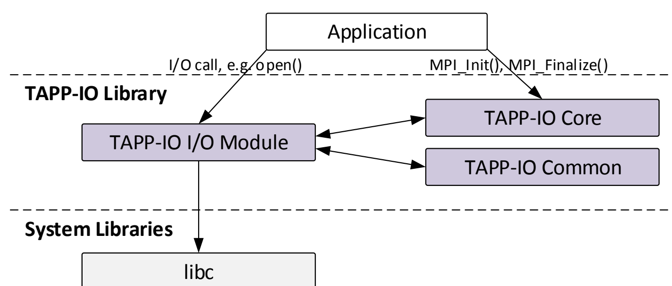


Figure 7.6: TAPP-IO runtime environment.

needs the number of writing tasks per file and the corresponding MPI communicator. TAPP-IO extracts the information from the info object and calculates the stripe size. The stripe size is matched to keep data from a process in a contiguous block. Processes can concurrently access a single shared file. Still, this feature lacks the flexibility to dynamically re-size a file.

For the time being, dynamic re-striping or re-sizing of a file comes with an enormous overhead. The basic idea is to re-create the file with the new striping pattern. But, this involves the copying (i.e, reading into memory and then writing out of the memory) of the file to the client and back to the parallel file system. This procedure is resource consuming, and therefore, not feasible. It is expected that the introduction of the progressive file layouts [214], which is based on composite layouts, with Lustre 2.10 will provide the means to efficiently enhance the balancing algorithm for single shared files.

7.8.2 Runtime Environment

TAPP-IO uses function interposition, similar to Recorder [215] and Darshan [195], to prioritize itself over standard library calls. For dynamically linked applications, the framework is built as a shared library. Once TAPP-IO is specified as the preloading library via `LD_PRELOAD`, it intercepts POSIX I/O, MPI-IO, and HDF5 file creation calls issued by the application and reroutes them to the balancing framework. For statically linked applications, the library requires no source code modifications, but has to be added transparently during the link phase of MPI compiler scripts such as `mpicc` or `mpif90`. This approach is a compromise in that existing binaries must be recompiled (or relinked) in order to use TAPP-IO. POSIX routines are intercepted by inserting wrapper functions via the GNU linker's `--wrap` argument.

After rerouting the function calls to the TAPP-IO framework, the library evenly places the data on the available storage targets. This balancing approach is trans-

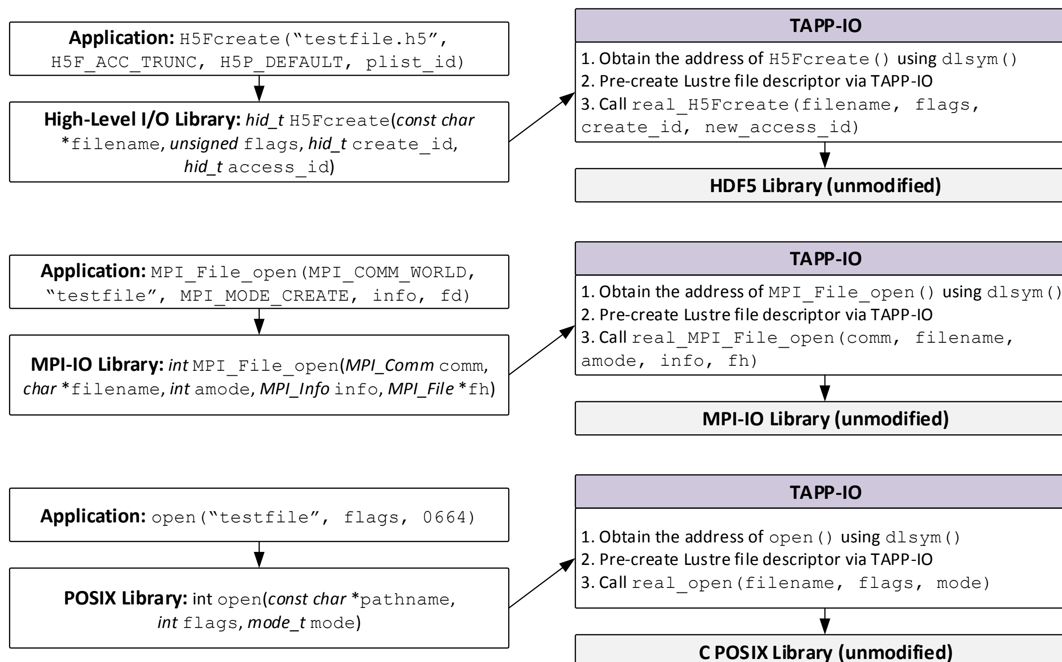


Figure 7.7: Dynamic interception of I/O functions at runtime.

parent to the user because alterations are made without changes to application and library source code.

For both dynamically and statically linked applications, TAPP-IO intercepts MPI-IO routines using the profiling interface to MPI (PMPI). Figure 7.6 illustrates the TAPP-IO runtime environment. The framework consists of three main components: *TAPP-IO Core*, *TAPP-IO Common*, and *TAPP-IO I/O Modules*. The core of the framework handles the initialization and clean up of the library. Before any I/O call can be rerouted to TAPP-IO, the internal data structures need to be initialized. This happens during the call to `MPI_Init()`. The common module hosts the balancing algorithm and helper functions to maintain module specific I/O characterization data. In addition, there is an I/O module for every supported I/O interface. The I/O modules implement the wrapper functions. Figure 7.7 displays the dynamic interception of I/O routines at runtime. The following sequence illustrates the mode of operation of TAPP-IO for HDF5:

(1) TAPP-IO intercepts and reroutes `H5Fcreate()` to the HDF5 I/O module.

(2) TAPP-IO Common provides a list of NID/OST bindings.

(3) A Lustre file descriptor is allocated with the balancing information.

(4) The function returns by calling `real_H5Fcreate()`.

The mechanism is the same for the MPI-IO and POSIX I/O. It offers per job and end-to-end I/O performance improvement in the most transparent way. Currently, the framework supports the following I/O calls: `open[64]()`, `creat[64]()`, `MPI_File_open()`, and `H5Fcreate()`. These mechanisms have been tested with the MPICH MPI implementation for both GNU and Cray C, C++, and Fortran compilers. It also works correctly for both static and dynamic compilation, requires no additional supporting infrastructure for instrumentation, and is compatible with other MPI implementations and compilers.

7.9 Data Collection and Analysis

This section presents the analysis results of Aequilibro and TAPP-IO. First, the benchmarking methodology and test setup are introduced followed by the experimental evaluation results obtained from the Spider II system.

7.9.1 Benchmarking Methodology

The benchmarking methodology utilizes two benchmarks, the synthetic IOR benchmark and the real-world HPC workload S3D. They are used to evaluate to overall performance improvement, bandwidth, execution time, and I/O interference.

7.9.1.1 IOR Benchmark

The well-known IOR [216] synthetic benchmark tool is adopted to exploit the strengths and weaknesses of Aequilibro and TAPP-IO. IOR provides a flexible way of measuring I/O performance under different read or write sizes, concurrencies, file formats, and access pattern strategies. It supports different I/O interfaces ranging from traditional POSIX I/O to advanced high-level I/O libraries like MPI-IO and HDF5, and differentiates parallel I/O strategies between file-per-process and single-shared-file approaches. Shan et al. [217] have demonstrated that IOR can be used to characterize and predict the I/O performance on large-scale HPC systems. IOR is utilized to evaluate the direct exploitation of the BPIO library in comparison to Aequilibro and TAPP-IO transparently handling I/O for an application.

Table 7.1 displays the IOR benchmark variants used with the *file-per-process* strategy. The evaluation is divided into three different I/O performance comparisons. First, the original version of BPIO is directly used for the data placement by modifying the IOR source code, referred to as *IOR BPIO*. Before creating a file

Table 7.1: IOR benchmark variants.

Index	Variant	Description
(I)	Default	The original IOR benchmark without any modification.
(II)	BPIO	A modified version of the IOR tool that utilizes the BPIO library for balanced data placement.
(III)	ADIOS	An IOR benchmark where all I/O calls are replaced with the ADIOS API for I/O handling.
(IV)	Aequilibro	Same code base as IOR ADIOS, but utilizes the BPIO library for balanced data placement.
(V)	TAPP-IO	Unmodified IOR benchmark utilizing TAPP-IO via `LD_PRELOAD`.

with Lustre's *llapi*, the BPIO library is used to determine the compute node (NID) to OST assignment. The results are compared to the unmodified IOR benchmark *IOR Default*. Second, all I/O interface calls are replaced by the ADIOS API. IOR is used as a workload generator to drive the ADIOS framework, denoted as *IOR ADIOS*. Using ADIOS with IOR provides an easy way to stress the file system while handling file I/O with the ADIOS API. Essentially, IOR is used as a workload generator to drive the ADIOS framework. A side benefit is that Aequilibro can be tested without any additional code modification. The third part of the evaluation provides the comparison of IOR Default and *IOR TAPP-IO*. For the *single shared file* (SSF) I/O strategy, three different variants of the IOR benchmark setups are used: (I) *IOR Default SSF*, (II) *IOR Optimized SSF*, and (III) *IOR TAPP-IO SSF*. Variant (I) employs the default Lustre striping configuration (stripe count = 4, stripe size = 1MB). Variant (II) uses an optimized striping setting (stripe count = numberOfWriters, stripe size = fileSize / numberOfWritingTask) and the default Lustre OST placement algorithm. Variant (III) uses the same stripe count and size as (II), but utilizes the TAPP-IO framework to determine the MPI process ID to OST binding. The list is used to configure the specific striping information transparently prior to file creation.

The metrics of interest include the overall execution time and the end-to-end I/O *performance improvement* gained by using either BPIO, Aequilibro or TAPP-IO. It is provided in percentage and calculated with the following equation:

$$\text{Performance Improvement (in \%)} = 100 * \left(\frac{\text{Bandwidth }_{balanced}}{\text{Bandwidth }_{default}} - 1 \right) \tag{7.7}$$

In order to accurately model a typical HPC workload behavior, the benchmark parameters need to be aligned with the desired workloads. The IOR benchmark provides a wide range of parameters including the *API*, *FilePerProc*, *WriteFile*, *NumTasks*, *BlockSize*, and *TransferSize*. On Titan, the memory size per node is 32 GB with 2 GB per processor. IOR is run with different block sizes to evaluate the impact of caching effects and a *TransferSize* of 1 MB. For POSIX I/O, the *fsync* and *useO_DIRECT* options are set. O_DIRECT bypasses I/O and file system buffers. For MPI-IO, the same effect can be achieved by enabling the *direct_io* MPI-IO optimization hint. For Lustre-specific settings, each file is created with a stripe size of 1 MB and in the case of file-per-process pattern, a *StripeCount* of 1 is used. The stripe size should be aligned with the *TransferSize* in order to get the best performance. *StripeCount* specifies the number of OSTs where the data is striped across while *StripeSize* defines the size of one stripe.

7.9.1.2 I/O Interference

Large-scale HPC systems waste a significant amount of computing capacity because of I/O interferences caused by multiple running applications concurrently accessing shared parallel file system and storage resources. Lofstead et al. [196] provide a definition of internal and external interferences to characterize the variability in I/O performance. The definition is used to evaluate the performance of TAPP-IO in terms of I/O interferences.

Internal interference When too many write processes from one specific application try to write to a single storage target at the same time, internal interference can occur. Write caches are exceeded which leads to a blocking behavior of the application until the buffers are cleared. The IOR benchmark is to evaluate the internal interference by writing data of different sizes to OSTs while scaling the number of nodes/writers. For simplicity reasons, the IOR benchmark is configured to use 1,008 OSTs and POSIX I/O.

External interference Even if an application would try to evenly use all available storage system resources, external interference can still occur. This is caused by the fact that a parallel file system is a shared resource where accesses are shared between all compute nodes and running applications. To demonstrate the I/O performance variability, hourly IOR tests with an 1,008 node allocation are performed, one process per node (one writer per node), and POSIX I/O with the file-per-process I/O strategy.

Another characterization metric is the *imbalance factor* of an I/O action as defined by Lofstead et al. [196]. It describes the ratio of the slowest ($wtime_{max}$) versus the fastest write ($wtime_{min}$) times across all writers:

$$\text{Imbalance factor} = \frac{wtime_{max}}{wtime_{min}} \tag{7.8}$$

The imbalance factor reflects the impact of the slowest writer on the overall performance. Therefore, the factor can be utilized to characterize the imbalance of an application.

7.9.1.3 HPC Workload

S3D [218] is a combustion code simulation that is employed on many different HPC systems. When executed, it generates a large amount of I/O requests. Verifying the I/O performance of S3D with TAPP-IO is a good indicator for the impact on other large-scale applications. Unfortunately, S3D is a proprietary code. *Genarray3D* [219] is a skeletal I/O application, which utilizes the ADIOS middleware's XML I/O specification with additional runtime parameters to emulate the I/O behavior of S3D. By default, the `MPI_AGGREGATE` transport method sets the striping information for a file. In order to run ADIOS with TAPP-IO, the part that specifies the striping information has to be removed.

In Genarray3D, three dimensions of a global array are partitioned among MPI processes along X-Y-Z dimensions in the same block-block-block fashion. Each process writes an N^3 partition. The size of each data element is 4 bytes, leading to the total data size of $N^3 * P * 4$ bytes, where P is the number of processes. One key difference between the IOR benchmark and Genarray3D is that all cores present on a compute node are utilized by default. This improves the computational efficiency of the simulation. On the other hand, Genarray3D generates pressure on single storage targets, because each compute node hosts its own operating system with a single mount point per file system.

7.9.2 Experimental Setup

All tests were performed on the Titan supercomputer. In order to get representative results, two major issues were addressed. First, all experiments were conducted in a busy production environment. No tests were run during the quiet maintenance mode. The results show that performance gains can be achieved in an active production environment. Second, a broad set of compute nodes were used instead of just a

certain subset of nodes. This demonstrates that independently from any specific node set on Titan, an application can readily benefit from the presented balancing framework. The application level placement scheduler (ALPS) on Titan returns a node allocation list where nodes tend to be logically close to each other. There are two attempts to get a higher node coverage. The first one is to submit scaling tests one after another independently, in the hope that a different set of compute nodes is covered with every run. The second attempt is to submit scaling runs in parallel to occupy a larger set of nodes. Both approaches are used to get a broader coverage. More than 30 scaled runs were obtained for IOR variants (I) to (IV) (see Table I), with each run ranging from 8 to 4,096 nodes. For each node allocation, three iterations are performed to obtain the average badwidth results.

All of the described experiments are conducted in a noisy, active production environment. Therefore, performance numbers may not always be conclusive. To cope with this issue and to draw consistent observation, multiple tests were performed with at least three repetitions per run. Iterations within the same run get the same node allocation. Each iteration executes IOR variants (I) to (IV). This is essential in order to average out the variance across the same set of allocated nodes. In addition, a large set of Titan compute nodes were covered throughout these tests. For the internal interference, more than 30 scaled IOR runs per node allocation have been performed. The number of allocated nodes ranges from 16 to 1,024 with 16 MPI processes per node which corresponds to a range from 128 to 16,384 processes in total. Each process writes a separate file. For the external interference, more than 100 samples were obtained.

7.9.3 Synthetic Benchmark Evaluation

The following sections present the IOR benchmark performance results, including the overall performance improvement, the application execution time, the achieved bandwidth, the standard deviation, and the interference results.

7.9.3.1 Performance Imrovement

The results of the scaling runs with a 4 GB file size per writing process and file-per-process strategy are summarized in Figure 7.8 for 64, 512, 2,048, and 4,096 nodes respectively. Over a period of four months, more than 30 scaled runs per node allocation size were obtained. Each sub-figure represents a particular node allocation. The X-axis represents the enumeration of runs with the same count of node allocation, but for *different* sets of nodes. The bandwidth performance of IOR Default and IOR

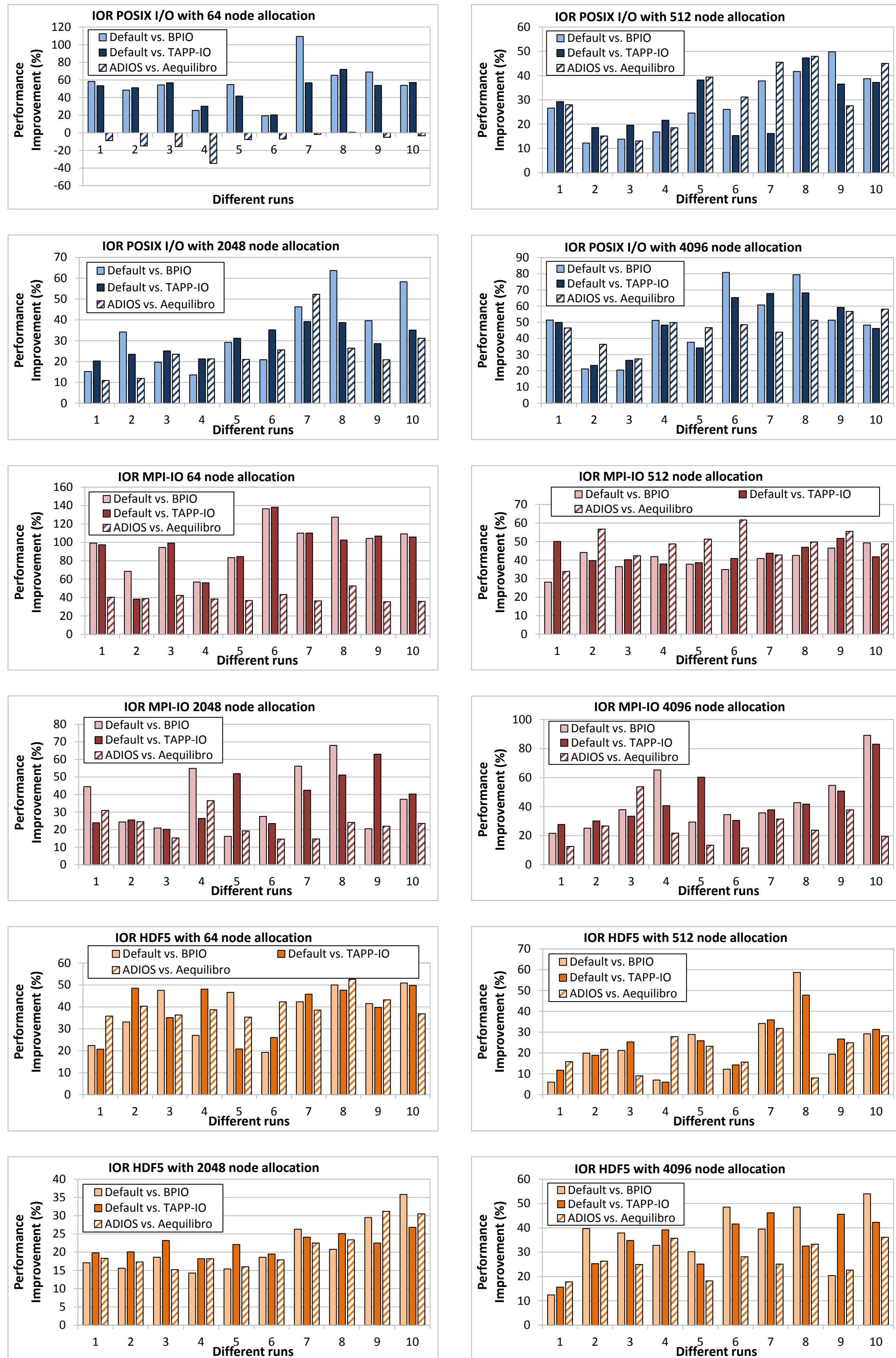


Figure 7.8: Performance improvements for IOR large-scale runs.

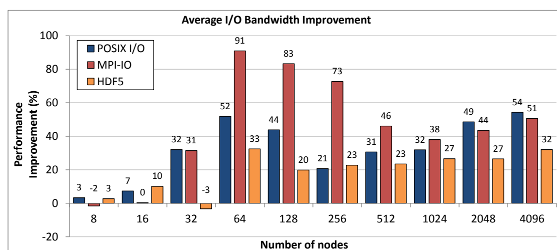


Figure 7.9: Average performance gains (in %): TAPP-IO vs. Default placement.

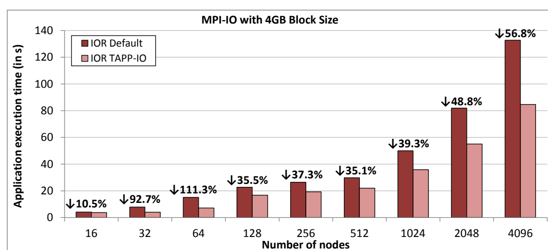


Figure 7.10: Average application execution time for MPI-IO with 4 GB block size.

BPIO (denoted as *Default vs. BPIO*), IOR Default and IOR TAPP-IO (denoted as *Default vs. TAPP-IO*), and IOR ADIOS and IOR Aequilibro (denoted as ADIOS vs. Aequilibro) utilizing Equation 7.7 are compared. In all cases, it can be seen that the balancing provides significant performance improvements for small-, medium-, and large-scale runs. An exception is the performance for Aequilibro at smaller scales for POSIX I/O. IOR BPIO and IOR TAPP-IO show similar performance improvement trends. For POSIX I/O, TAPP-IO provides about 40% of performance improvement for 2,048 nodes and about 50% for 4,096 nodes. Similar trends can be observed for HDF5 and MPI-IO at large-scale. For 4,096 nodes, TAPP-IO provides up to 89% of performance improvement for MPI-IO and 54% for HDF5. It is noteworthy that the performance improvement achieved by Aequilibro is inferior to BPIO and TAPP-IO. The additional overhead introduced by the I/O middleware framework lowers the overall I/O performance. While there are variations across different runs, it can be observed that the trend remains the same. There are consistent performance gains across multiple runs and iterations. Optimizing the overall I/O cost leads to a reduced application execution time (especially for large-scale runs) and therefore, to a reduced operational cost per executed application. Figure 7.9 shows the performance improvement averaged over all completed runs for file-per-process I/O and a 4 GB file size per writing process (see also Figure 7.8). It confirms that TAPP-IO consistently provides a higher throughput with the balanced placement algorithm. The only exceptions are MPI-IO for 8 nodes and HDF5 for 32 nodes.

7.9.3.2 Application Execution Time

Figure 7.10 presents the average application execution time of IOR Default and IOR TAPP-IO for MPI-IO with file-per-process. The percentage on top of the bars describes the time improvement. A similar trend can be observed for POSIX I/O and HDF5. From the results, it can be concluded that resolving resource contention at

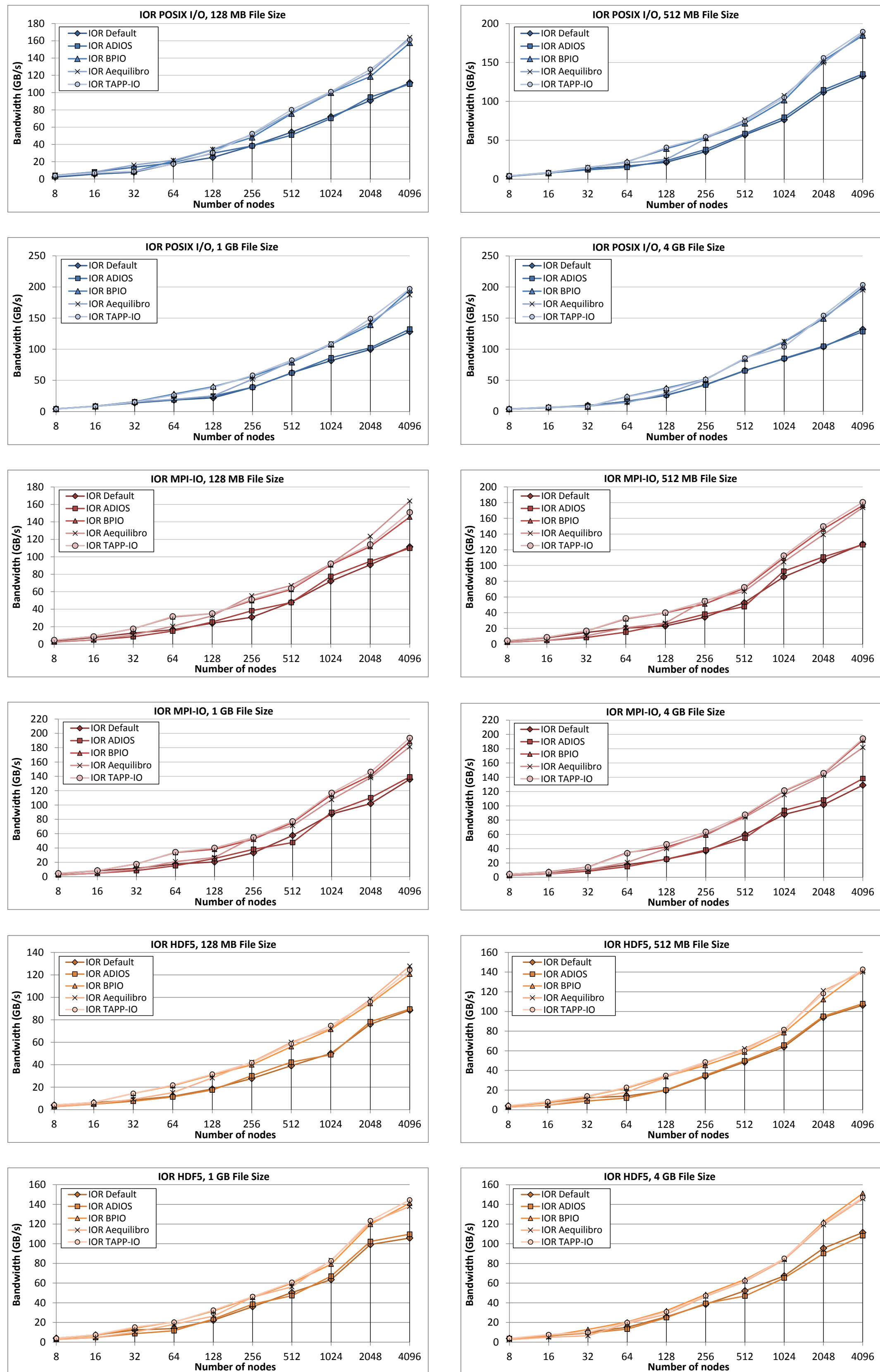


Figure 7.11: Bandwidth performance for IOR (I) to (V) for different file sizes.

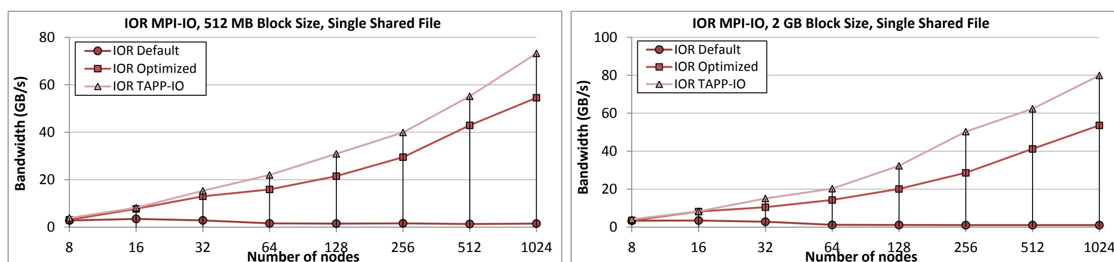


Figure 7.12: IOR bandwidth performance for MPI-IO and single shared file.

the storage system level directly impacts the overall execution time of the application. With the advent of big data and the increasing amount of defensive I/O in mind, it is expected that the balancing mechanism will have a tremendous effect on an application's performance.

7.9.3.3 Bandwidth Results

Figure 7.11 summarizes the IOR bandwidth results for different file sizes scaling from 8 to 4,096 nodes for file-per-process. The results illustrate the average bandwidth per second from over more than 40 scaled runs with at least three repetitions per IOR variant (refer to Table I) per node allocation within one run. The results were collected over the period of four months. From the throughput results, the following observations can be made. First, starting from small-scale runs with at least 16 nodes, our load balancing framework TAPP-IO and the BPIO library both provide consistent bandwidth improvements. Aequilibro does not provide any significant improvement for less than 128 nodes. Second, with an increasing number of I/O processes and allocated computing nodes, POSIX, MPI-IO, and HDF5 benefit from utilizing TAPP-IO with IOR. For example, IOR TAPP-IO with POSIX I/O achieves up to 202.7 GB/s on average for a 4,096 node allocation and a 4 GB file size per writing process. This can be translated to 54% performance improvement compared to the default data placement with IOR Default. For smaller file sizes, the maximum bandwidth is less, but the average performance improvement trend remains the same compared with the default data placement. This consistent improvement can be observed for MPI-IO and HDF5 as well. Another noteworthy aspect is that the expected caching effects for smaller file sizes were non-existent. In summary, it can be said that all IOR variants utilizing a data balancing algorithm are able to provide similar performance results. But, TAPP-IO makes the application independent from the need to actively adopt the BPIO mechanism by integrating it into the application or using an I/O framework such as Aequilibro.

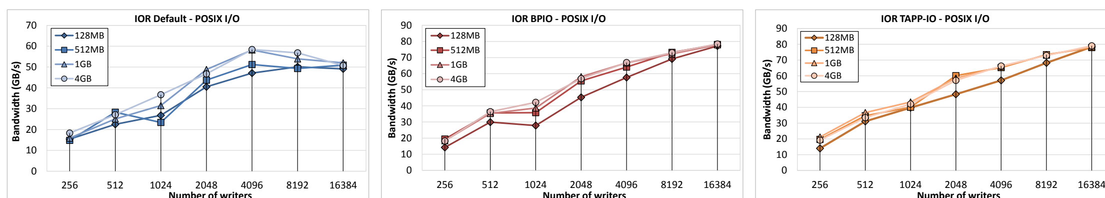


Figure 7.13: Scaling of aggregate write bandwidth.

Figure 7.12 displays initial performance results of TAPP-IO with the single shared file balancing algorithm for IOR with MPI-IO for different block sizes. The result present the average bandwidth out of 30 scaled runs with node allocations ranging from 8 to 1,024 nodes with one writing process per allocated node. Similar to the file-per-process mode, TAPP-IO provides significant performance improvement starting with a node allocation as minimal as 32 nodes. The default Lustre striping pattern throttles the throughput tremendously. The default striping pattern distributes the file over 4 OSTs with a striping size of 1. Multiple writing processes try to access the same OST at the same time. The optimized IOR version provides an increasing bandwidth compared to the default variant, but still utilizes the Lustre default OST placement like the file-per-process results. The observed performance gain by distributing stripes of the same file evenly among available storage targets is consistent with the observations made for the file-per-process I/O pattern. For example, TAPP-IO provides a performance improvement of about 75.8% compared to the optimized placement for 256 nodes. Another consistent observation that should be noted is that the standard deviation results obtained from different iterations within the same run was relatively small for TAPP-IO compared to the results obtained with the Lustre default data placement.

7.9.3.4 Standard Deviation

The IOR benchmark output provides the standard deviation and mean calculated from the performance results of the executed repetitions per run. While evaluating the collected benchmarks results, the following consistent observation for all IOR variants utilizing a balancing algorithm could be made, independently from the I/O interface. For both Aequilibro and TAPP-IO, the standard deviation is tremendously lower for all tested file sizes. The standard deviation is a measure to quantify the amount of variation of a set of data values. In other words, when utilizing BPIO, the achieved bandwidth of each repetition is relatively close while using the Lustre default data placement leads to a huge variation among repetitions.

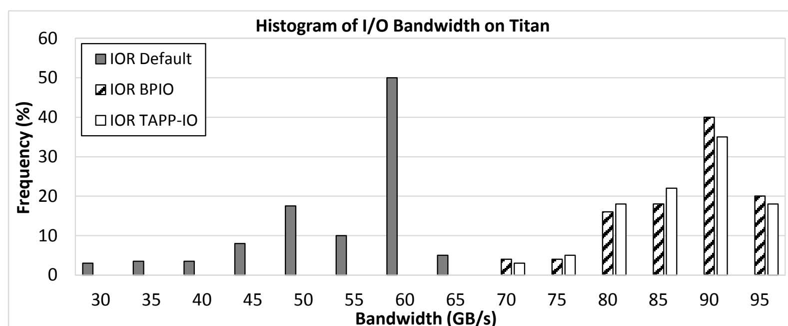


Figure 7.14: I/O performance variability due to external interference.

7.9.3.5 Interference Results

Figure 7.13 presents the results of the scaled internal interference tests, as introduced in Section 7.9.1.2. The results represent the average write bandwidth on Titan for IOR with POSIX I/O. With an increasing number of nodes and writers, it can be observed that in the case of IOR Default and IOR ADIOS the effects of internal interferences consistently decrease the average bandwidth with an increasing number of writers. For IOR BPIO/TAPP-IO and IOR Aequilibro, the internal interference effects can still be observed, but they have less impact on the overall bandwidth performance due to a balanced data distribution over all available storage targets.

Figure 7.14 displays the histograms of I/O bandwidth based on the external interference tests data collected in over 100 runs. It can be seen that in busy production environments like Titan, there is a substantial I/O variability between different runs. Note, BPIO/TAPP-IO and Aequilibro are solely used for the test runs. There are multiple other applications scheduled at the same time. In addition, the imbalance factor is calculated based on Equation 7.8. For IOR Default, the imbalance factor is about 6.9 in average, while for IOR BPIO/TAPP-IO it ranges in between 1.3 and 1.9 leading to an improvement by a factor of 3.5. For both IOR ADIOS and IOR Aequilibro, the imbalance factor ranges in between 1.1 and 1.2. ADIOS offers synchronous write methods. The imbalance factor does not provide any information about the overall performance.

7.9.4 HPC Workload Evaluation

Scaled runs with 128, 256, 512, 1,024, 2,048, and 4,096 nodes are performed, which correspond to 2,048, 4,096, 8,192, 16,384, 32,768, and 65,536 MPI processes, respectively. Weak scaling of the problem size grid is used such that each process generates an 8 MB output/checkpoint file periodically (10 checkpoints in each run).

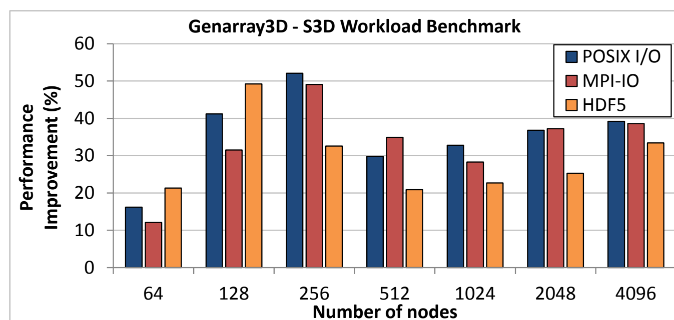


Figure 7.15: Average I/O bandwidth improvement for S3D workload.

The I/O bandwidth measurement is performed for default (ADIOS) and balanced data placement (ADIOS with TAPP-IO) by running three Genarray3D simulations within the same allocation.

Figure 7.15 displays the summary of the I/O bandwidth improvements observed for S3D-IO. The improvements are averaged over ten runs for each configuration. It can be observed that even for small node count runs the performance can be improved. For large-scale runs, we observe that TAPP-IO significantly improves the I/O bandwidth. This is consistent with the IOR synthetic benchmark performance results. For large node/processor counts, applications can directly benefit from TAPP-IO without any additional code changes.

7.10 Summary

This chapter has introduced Aequilibro, a transparent unification of BPIO and ADIOS, and TAPP-IO, a dynamic, shared data placement framework, which transparently intercepts file creation calls. Both tools provide the user with a transparent solution that takes full advantage of the optimizations done at the interconnect level and the load balancing done at the file system level. Large-scale applications can directly benefit from the BPIO end-to-end and per job I/O performance improvements, and also leverage the optimization presented through the fine-grained routing methodology. Also, both solutions tune the file layout in accordance with the presented best practices for large-scale file I/O. While Aequilibro only supports the file-per-process access pattern, TAPP-IO also provides a balancing mechanism for single shared file accesses.

The effectiveness of both Aequilibro and TAPP-IO has been evaluated on the Titan supercomputer. IOR, a synthetic benchmark, and S3D, a real-world HPC workload, have been utilized to analyze the performance for POSIX I/O, MPI-IO, and HDF5 in terms of bandwidth, application execution time, and interferences. The evaluation

has been performed over an extended period of time (i.e., over three months) in order to provide conclusive results. The evalation results have demonstrated that both Aequilibro and TAPP-IO translate the benefits of BPIO transparently into an application, while providing consistent performance improvements for different node allocations. For example, POSIX I/O and MPI-IO can be improved by up to 50% for large-scale runs (i.e., more than 1024 allocated nodes), while HDF5 shows performance improvements of up to 32%.

Chapter 8

Conclusion

Today's supercomputing systems comprise heterogeneous system architectures. Their software environments require to be carefully tuned to excel in reliability, programmability, and usability, but also to cope with the extreme concurrency and heterogeneity of today's and future systems. One key observation has been that the performance of such systems is dominated by the time spent in communication and I/O, which highly depends on the capabilities of the interconnection network and its software ecosystem. The aim of this work was to resolve the I/O, communication, and concurrency gaps by designing, implementing, and evaluating intermediate software layers for the Extoll network technology. In addition, the work has been complemented by a research study targeting the Titan supercomputing system in order to gain valuable insights on the impact of the I/O gap at larger scales.

The first contribution has presented the *network-attached accelerator* (NAA) approach, which proposes a novel communication architecture for scaling the number of accelerators and compute nodes independently. The NAA design leverages two innovative features of the Extoll technology, the SMFU and remote register file accesses, which allow the remote configuration and operation of accelerator devices over the network. This work has focused on the design and the implementation of an intermediate software layer, which allows the disaggregation of accelerator devices from their PCIe hosts, and introduces a unique technique to virtualize remote accelerator devices. NAA enables accelerator-to-accelerator direct communication without local host interaction and dynamic compute resources allocation, which makes this work particularly interesting for the design of future exascale systems. Two prototype systems have been evaluated for two different classes of accelerator devices:

Intel Xeon Phi coprocessors and NVIDIA GPUs. The performance evaluation has demonstrated that the communication time for direct accelerator communication can be significantly reduced. Also, the NAA design has been utilized to enable the booster cluster, which has been deployed in the DEEP prototype system.

The second contribution has evolved around the need to provide high-performance SANs with the support for legacy codes and protocols. Even today, the TCP/IP protocol suite, together with the Sockets interface, is one of the most dominant standards for end-to-end network communication. This work has introduced two protocols, which target the acceleration of traditional TCP/IP communication over Extoll. First, *Ethernet over Extoll* (EXT-Eth) has been presented. The protocol provides IP addressing and address resolution, but also leverages Extoll's high performance capabilities through two-sided, asynchronous RDMA read operations. The second protocol is *Direct Sockets over Extoll* (EXT-DS), which provides kernel bypass data transfers for TCP point-to-point communication. For both protocols, the evaluation of the prototype implementations has shown promising results. In addition, EXN, the network driver implementing EXT-Eth, has actively been used in the DEEP and DEEP-ER prototype systems. In DEEP, EXN has been used to provide the SSH login channel between the booster nodes. In DEEP-ER, BeeGFS has been run on top of EXN to provide the storage connectivity [220].

Another important building block of large-scale HPC facilities is the parallel file and storage system. The third contribution of this work has focused on the Lustre file system and its network protocol semantics. Even though Lustre is one of the most popular file system solutions in the world, its implementation details are barely documented. Therefore, the first part of this work has focused on the comprehensive analysis of the Lustre network protocol (LNET) semantics. These insights have been used to efficiently map the LNET protocol onto the Extoll technology. One particular goal was the minimization of the required communication overhead. The prototype implementation, called EXLND, leverages the innovative notification mechanism of Extoll's RMA unit, which reduces the synchronization overhead to a bare minimum. An initial performance evaluation has shown that the bandwidth and message rates are on par with the theoretical peak performance. The documentation of LNET in combination with the development cycle of an RDMA-capable LND provides invaluable insights for the entire open science HPC community.

The last but not least contribution of this work has evolved around the Titan supercomputer and its storage system Spider II, which is based on the Lustre technology. Load imbalance and resource contention are common problems in large-

scale HPC systems. This contribution focuses on the mitigation of these problems and has introduced the design, implementation, and evaluation of two user-friendly load balancing frameworks, which combine the best practices recommended for large-scale file I/O with the topology-aware BPIO algorithm. The resulting solutions distribute the I/O workload across all available storage system components in an equal manner. The first solution is called *Aequilibro* and translates the benefits of BPIO into the platform-neutral I/O middleware ADIOS. The second framework is called *TAPP-IO* and transparently intercepts file creation calls, i.e., metadata operations, to balance the workload among all available storage targets, without the need to modify the application code. Both frameworks have been evaluated over extended period of time. The results obtained through these test runs have confirmed the effectiveness of the presented solutions. For example, the performance of POSIX I/O and MPI-IO have been improved by up to 50% for large-scale application runs, while HDF5 has demonstrated performance improvements of up to 32%.

In summary, it can be said that the challenges that have been phrased at the beginning of this work have been successfully addressed. By designing, implementing, and evaluating a variety of intermediate software layers for the Extoll interconnection technology and the Titan supercomputing system, this work contributes a comprehensive study of network communication and I/O through the entire software ecosystem in respect to the interconnection network, which makes this a valuable contribution to both system architects and researchers. In addition, this is the first work that efficiently leverages all hardware capabilities of the Extoll NIC, thus, proving the universal applicability of the Extoll research project to the HPC domain.

8.1 Outlook

For future work addressing the Extoll software environment with respect to the support of different communication protocols, but also the optimization of the presented load balancing techniques, several possible improvements can be identified:

Network-Attached Accelerators and VPCI

While the booster implementation has been successful and is rather robust, VPCI is currently limited to one host. This limitation is imposed by the fact that Extoll maps the accelerator's device memory over the local SMFU BAR window. In addition, VPCI adds the accelerator as a virtual PCI device to the local PCI hierarchy. Future work needs to explore how devices such as GPUs can be efficiently accessed by all

compute nodes in a local network. One possible solution is the adoption of the *Global GPU Address Space* architecture [99]. However, this would require a design shift in the SMFU configuration. Instead of utilizing the limited interval setup, the address mask approach could be used to provide a distributed shared address space.

TCP/IP Protocol Support

The current design of EXT-Eth and its network interface EXN does not support IP multicast groups. To cope with this limitation, a combination of hardware-backed routing table configuration and additional software is needed to emulate the promiscuous mode. In addition, the current implementation of `AF_EXTL` does not support RMA operations yet, which are required for both the accelerated BCopy mode and the true zero-copy data transfer mode.

Lustre Networking Support through EXLND

The present-day implementation of EXLND utilizes the immediate send transmission protocol only for messages of type `ACK`. By extending the usage of immediate send to small `PUT` and `GET` messages, it is expected that the overall network performance is improved, but also that the associated overhead can be reduced. In addition, EXLND does not support the multi-rail routing feature of recent Lustre releases yet. This feature allows LNET routers to be added as peers with multiple interfaces and ensures equal traffic distribution among all routers.

Resource Contention Mitigation Techniques

Aequilibro and TAPP-IO both utilize a load balancing heuristic based on a weighted cost function, but do not consider current system statistics or the filling level of OSTs. The future version of TAPP-IO will be backed by an "end-to-end control plane". The balancing framework will be complemented by a global coordinator, which gathers statistics from the MDSs and OSSs, collects feedback, and supplies placement results based on a prediction model back to the clients. On the client side, TAPP-IO will interact with a MySQL database, which contains the predicted placement information, i.e., the OSTs and stripe size.

List of Abbreviations

ACL .... Access Control List
ADIOS .... Adaptable I/O System
AN .... Accelerator Node
API .... Application Programming Interface
APIC .... Advanced Programmable Interrupt Controller
ARP .... Address Resolution Protocol
ASIC .... Application Specific Integrated Circuit
ATU .... Address Translation Unit
AZ-SDP ...... Asynchronous Zero-Copy SDP
BAR .... Base Address Register
BCopy .... Buffered Copy
BI .... Booster Interface
BIC .... Booster Interface Card
BIOS .... Basic Input/Output System
BN .... Booster Node
BNC .... Booster Node Card
BPIO .... Balanced Placement I/O
CN .... Compute Node
CPU .... Central Processing Unit
CQ .... Completion Queue
CRC .... Cyclic Redundancy Check
CUDA .... Compute Unified Device Architecture
DARPA .... Defense Advanced Research Projects Agency
DDP .... Direct Data Placement
DEEP .... Dynamical Exascale Entry Platform
DEEP-ER .... DEEP – Extended Reach
DEEP-EST ... DEEP – Extreme Scale Technologies
DHCP .... Dynamic Host Configuration Protocol
DMA .... Direct Memory Access

DNE .... Lustre Distributed Namespace
EMP .... Extoll Management Program
EOP .... End of Packet
EXLND .... Extoll Lustre Network Driver
EXT-DS ...... Direct Sockets over Extoll
EXT-Eth ...... Ethernet over Extoll
FGR .... Fine-Grained Routing
FID .... File Identifier
FIFO .... First In First Out
GAT .... Global Address Table
GATe .... Global Address Table Entry
GGAS .... Global GPU Address Spaces
GPGPU ...... General Purpose Graphics Processing Unit
GPU .... Graphics Processing Unit
HBM2 .... High Bandwidth Memory 2
HCA .... Host Channel Adapter
HDF5 .... Hierarchical Data Format v5
HPC .... High Performance Computing
HPL .... High-Performance Linpack
HT .... HyperTransport
HToC .... HyperTransport over CAG
I/O .... Input/Output
IBTA .... InfiniBand Trade Association
ICMPv6 ...... Internet Control Message Protocol for IPv6
IGMP .... Internet Group Management Protocol
ION .... I/O Forwarding Nodes
IP .... Internet Protocol
IPv4 .... Internet Protocol Version 4
IPv6 .... Internet Protocol Version 6
IRQ .... Interrupt Request
ITR .... Interrupt Throttling Rate
iWARP .... Internet Wide Area RDMA Protocol
LAA .... Locally Administered Address
LAN .... Local Area Network
LDLM .... Lustre Distributed Lock Manager
LMV .... Logical Metadata Volume

LND .... Lustre Network Driver
LNET .... Lustre Network Protocol
LOV .... Logical Object Volume
MAC Address Media Access Control Address
MD .... Memory Descriptor
MDC .... Metadata Client
MDS .... Metadata Server
MDT .... Metadata Target
ME .... Match Entry
MGC .... Management Client
MGS .... Management Server
MGT .... Management Target
MIC .... Many Integrated Core
MMIO .... Memory-Mapped I/O
MMU .... Memory Management Unit
MPA .... Marker PDU Aligned
MPI .... Message Passing Interface
MPP .... Massively Parallel Multi-Processor
MSI .... Message Signaled Interrupt
MTL .... Matching Transport Layer
MTT .... Message Type Tag
MTU .... Maximum Transmission Unit
NAA .... Network-Attached Accelerators
NIC .... Network Interface Controller
NID .... Network Identifier
NLA .... Network Logical Address
NLP .... Network Logical Page
NLPA .... Network Logical Page Address
NPD .... Neighbor Discovery Protocol
NUMA .... Non-Uniform Memory Access
OFED .... OpenFabrics Enterprise Distribution
OID .... Object Identifier
OMB .... OSU Micro-Benchmarks
OS .... Operating System
OSC .... Object Storage Client
OSS .... Object Storage Server

OST .... Object Storage Target
OUI .... Organizationally Unique Identifier
PCI .... Peripheral Component Interconnect
PCIe .... Peripheral Component Interconnect Express
PD .... Protection Domain
PDU .... Protocol Data Unit
PGAS .... Partitioned Global Address Space
PIC .... Function-independent Code
PIO .... Programmed Input/Output
PMPI .... Profiling interface to MPI
POSIX .... Portable Operating System Interface
QP .... Queue Pair
RAM .... Random Access Memory
RC .... Reliable Connected
RDMA .... Remote Direct Memory Access
RDMAP ...... RDMA Protocol
RMA .... Remote Memory Access
RoCE .... RDMA over Converged Ethernet
RPC .... Remote Procedure Call
RRA .... Remote Register File Access
SAN .... System Area Network
SCI .... Scalable Coherent Interface
SCIF .... Symmetric Communication Interface
SCTP .... Stream Control Transmission Protocol
SDP .... Sockets Direct Protocol
SGE .... Scatter/Gather Elements
SHOC .... Scalable Heterogeneous Computing
SIONlib .... Scalable I/O library
SMFU .... Shared Memory Functional Unit
SMP .... Symmetric Multiprocessor
SMX .... Streaming Multiprocessor Unit
SOP .... Start of Packet
SPMD .... Single Program Multiple Data
SSF .... Single Shared File
TAPP-IO ..... Transparent Automatic Placement of Parallel I/O
TCP .... Transmission Control Protocol

TLB .... Translation Look-aside Buffer
TOR .... Top-of-the-Rack
UAA .... Universally Administered Address
UD .... Unreliable Datagram
UDP .... User Datagram Protocol
UEFI .... Unified Extensible Firmware Interface
UMA .... Uniform Memory Access
UPC .... Universal Parallel C
VELO .... Virtualized Engine for Low Overhead
VFS .... Virtual File System
VPID .... Virtual Process Identifier
WC .... Work Completion
ZCopy .... Zero-Copy
ZFS .... Zettabyte File System

List of Figures

1.1 TOP1 system performance and power consumption development. . . 2
1.2 Key factors influencing the total application execution time. 4

2.1 Shared memory system architectures. 10
2.2 Distributed memory architecture. 10
2.3 Communication architecture abstraction following Culler et al. 11
2.4 Communication models in distributed memory systems. 15
2.5 Basic send/receive sequence. 17
2.6 Remote Direct Memory Access operations. 17
2.7 Remote load and store operations. 17
2.8 MPI eager protocol. 20
2.9 MPI rendezvous protocol. 21
2.10 Abstract model of HPC system with external file system. 24
2.11 Typical parallel I/O architecture for data-intensive sciences. 25
2.12 Schemes of storage systems attached to a HPC system. 28
2.13 Single writer I/O pattern. 32
2.14 File-per-process I/O pattern. 32
2.15 Single shared file I/O patterns. 33

3.1 Overview of the Extoll NIC top-level diagram. 36
3.2 Overview of notification mechanism for Extoll PUT and GET. 38
3.3 Sized PUT/GET software descriptor format. 39
3.4 VELO message software descriptor format. 40
3.5 SMFU's address space layout for the interval configuration. 41
3.6 ATU address translation overview. 43
3.7 Overview of the PCIe Bridge unit. 45
3.8 Overview of the Extoll software environment. 47
3.9 Overview of Infiniband HCA system integration. 49
3.10 PCIe topology. 51

3.11 MPI performance overview. 53

4.1 Accelerator development trend in the TOP500 since 2011. 56
4.2 The DEEP Cluster-Booster concept. 58
4.3 PCI Express configuration space header types. 60
4.4 Base address register. 61
4.5 MSI Capability register set for 64-bit address size. 62
4.6 Linux PCI Express enumeration example. 63
4.7 Simplified architectural overview of an Intel Xeon Phi Coprocessor. . 65
4.8 Nvidia's Kepler GK110 architecture + DRAM 66
4.9 NAA architecture diagram: system versus user view. 70
4.10 PCI Express tree as seen by the Linux operating system with one network-attached accelerator device. 71
4.11 Abstract software stack view. 73
4.12 Memory mapping between a cluster node and two accelerators. 75
4.13 Interrupt handling within the booster. 77
4.14 System and user view. 79
4.15 Hardware components. 80
4.16 BIC software stack. 81
4.17 Communication paths. 84
4.18 Micro-benchmarks performance of internode MIC-to-MIC communication using the Extoll interconnect. 86
4.19 Half round-trip latency performance of internode MIC-to-MIC communication using MPI. 87
4.20 Bandwidth and bidirectional bandwidth performance of internode MIC-to-MIC communication using MPI. 87
4.21 Communication time for the bead-spring polymer melt benchmark. . 89
4.22 Overall application time for 64 threads, 32 Threads/MIC. 89
4.23 Production-ready GreenICE cube. 90
4.24 Control flow of PCI configuration space accesses. 94

5.1 Comparison of the OSI and TCP/IP reference models. 102
5.2 Overview of the TCP/IP data transmission path. 106
5.3 Overview of the TCP/IP data reception path. 107
5.4 Schematic overview of the NAPI functionality. 108
5.5 Overview of interconnects and protocols in the OpenFabrics stack. . . 112
5.6 Overview of the Extoll software stack with TCP/IP extensions. . . . 116

5.7 Ethernet over Extoll software frame format. 117
5.8 Transmission of large messages via eager protocol. 118
5.9 Transmission of large messages via rendezvous protocol. 119
5.10 RMA standard notification format. 120
5.11 Payload layout after RMA MTU fragmentation. 121
5.12 MAC address layout universal versus local. 123
5.13 Extoll MAC address layout. 124
5.14 Internet Protocol (IPv4) over Ethernet ARP packet. 125
5.15 Path of an incoming packet in NAPI mode. 128
5.16 Interrupt throttling rate state transitions of the *ixgbe* driver. 130
5.17 Overview of system calls used in stream sockets connection. 133
5.18 Relation between socket handles, ports, and virtualized hardware. . . 134
5.19 BCopy transmission flow. 137
5.20 RMA transfer sequence and buffer copies for BCopy mode. 137
5.21 ZCopy read transmission flow. 138
5.22 ZCopy write transmission flow. 139
5.23 TCP state transition diagram. 140
5.24 Overview of the socket software stack. 142
5.25 Overview of the Kernel FIFO usage with different message types. . . 144
5.26 Netperf bandwidth performance for TCP streaming connections. . . . 148
5.27 Netperf bandwidth performance for EXN with different TCP/IP tuning configurations. 149
5.28 TCP/IP PingPong benchmark results for TCP connections. 150
5.29 Host CPU utilization during netperf bandwidth test. 151
5.30 OMB latency performance comparing Skylake and Haswell. 152
5.31 OMB bandwidth performance comparing Skylake and Haswell. 152
5.32 OMB message rate performance comparing Skylake and Haswell. . . . 152

6.1 Lustre file system components in a basic setup. 157
6.2 An overview of the internal Lustre architecture. 158
6.3 Overview of the LNET software architecture. 159
6.4 Illustration of the Lustre LNET addressing scheme. 160
6.5 Overview of Lustre I/O operations. 162
6.6 Lustre file layout overview. 163
6.7 LNET API naming conventions. 164
6.8 Overview of LNetPut() communication sequence. 168
6.9 Overview of LNetGet() communication sequence. 168

6.10 Lustre software environment with EXLND. 172
6.11 Physical buffer lists. 174
6.12 Relation of work requests, scatter/gather elements, main memory, memory regions and protection domain. 175
6.13 Overview of LNetPut() communication sequence over Extoll. 178
6.14 Overview of LNetGet() communication sequence over Extoll. 178
6.15 The movement of list elements representing unfinished transfers. . . . 181
6.16 LNET self-test: bulk read test simulating Lustre I/O traffic. 184
6.17 LNET self-test: bulk write test simulating Lustre I/O traffic. 185
6.18 LNET self-test: ping test simulating Lustre metadata traffic. 185

7.1 Infrastructure and I/O path between Titan and Spider II. 190
7.2 Default resource usage distribution on Spider II. 192
7.3 Topological XY representation of Titan's I/O routers. 197
7.4 Aequilibro software stack. 204
7.5 Shared file layout patterns. 208
7.6 TAPP-IO runtime environment. 210
7.7 Dynamic interception of I/O functions at runtime. 211
7.8 Performance improvements for IOR large-scale runs. 217
7.9 Average performance gains (in %): TAPP-IO vs. Default placement. . 218
7.10 Average application execution time for MPI-IO with 4 GB block size. 218
7.11 Bandwidth performance for IOR (I) to (V) for different file sizes. . . . 219
7.12 IOR bandwidth performance for MPI-IO and single shared file. 220
7.13 Scaling of aggregate write bandwidth. 221
7.14 I/O performance variability due to external interference. 222
7.15 Average I/O bandwidth improvement for S3D workload. 223

List of Tables

3.1 Overview of possible RMA notification type combinations. 37
3.2 Overview of PCIe backdoor registers needed for configuration. 46
3.4 Infiniband versus Ethernet performance comparison. 50

4.1 Description of LAMMPS timings output. 88
4.3 Overview of the benchmark results. 98

5.1 TCP/IP protocol overhead for Gigabit Ethernet under Linux 2.6.18. . 110
5.3 Overview of VELO user tags. 135

6.1 Overview of LNET message types. 167

7.1 IOR benchmark variants. 213

Listings

4.1 Overview of the VPCI device structure. 92
4.2 Generic PCI function pointers defined in <include/linux/pci.h>. . . . 93

5.1 Example output of ifconfig for Extoll network interface exn0. 129
5.2 TCP host kernel parameters configuration via sysctl. 146

6.1 The LND struct. 165
6.2 A memory descriptor fragment with a kernel virtual address. 170
6.3 A page-based fragment of a memory descriptor. 170
6.4 Lustre configuration for EXLND. 182

7.1 Example ADIOS XML configuration file. 203
7.2 Overview of file create and open calls. 205
7.3 *llapi* open and create calls. 205

References

[1] Moshe Y. Vardi. 'Science Has Only Two Legs'. In: *Communications of the ACM* 53.9 (Sept. 2010), pp. 5–5. ISSN: 0001-0782. DOI: 10.1145/1810891.1810892.

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. 'Design of ion-implanted MOSFET's with very small physical dimensions'. In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268. ISSN: 0018-9200. DOI: 10.1109/JSSC.1974.1050511.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. 'Dark silicon and the end of multicore scaling'. In: *38th Annual International Symposium on Computer Architecture (ISCA)*. June 2011, pp. 365–376.

[4] *TOP500 Supercomputer Sites*. URL: http://www.top500.org/ (visited on 10/01/2018).

[5] E. Strohmaier, H. W. Meuer, J. Dongarra, and H. D. Simon. 'The TOP500 List and Progress in High-Performance Computing'. In: *Computer* 48.11 (Nov. 2015), pp. 42–49. ISSN: 0018-9162. DOI: 10.1109/MC.2015.338.

[6] Benjamin Klenk and Holger Fröning. 'An Overview of MPI Characteristics of Exascale Proxy Applications'. In: *High Performance Computing*. Ed. by Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes. Cham: Springer International Publishing, 2017, pp. 217–236. ISBN: 978-3-319-58667-0.

[7] ASCR Advisory Committee. *The Opportunities and Challenges of Exascale Computing*. Exascale Advisory Committee Report. 2010. URL: https://science.energy.gov/ascr/ascac/ (visited on 10/07/2018).

[8] Peter Kogge et al. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*. http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf. Sept. 2008.

[9] Sarah Neuwirth, Dirk Frey, Mondrian Nuessle, and Ulrich Bruening. ‘Scalable Communication Architecture for Network-Attached Accelerators’. In: *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2015, pp. 627–638. DOI: 10.1109/HPCA.2015.7056068.

[10] Sarah Neuwirth, Dirk Frey, and Ulrich Bruening. ‘Communication Models for Distributed Intel Xeon Phi Coprocessors’. In: *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. Dec. 2015, pp. 499–506. DOI: 10.1109/ICPADS.2015.69.

[11] Sarah Neuwirth, Dirk Frey, and Ulrich Bruening. ‘Network-Attached Accelerators: Host-independent Accelerators for Future HPC Systems’. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC15)*. Poster paper. 2015.

[12] Ian Cutress. *Host-Independent PCIe Compute: Where We’re Going, We Don’t Need Nodes*. Dec. 2015. URL: https://www.anandtech.com/show/9851/hostindependent-pcie-compute-where-were-going-we-dont-need-nodes (visited on 10/04/2018).

[13] Sarah Neuwirth, Ulrich Bruening, and Mondrian Nuessle. ‘Measurements with an Extreme Low-Latency Interconnect on HP Servers’. In: *High Performance - Consortium for Advanced Scientific and Technical Computing User Group Meeting (HP-CAST 25)*. Nov. 2015.

[14] Sarah Neuwirth, Mondrian Nuessle, and Ulrich Bruening. ‘Alternative Low-latency Interconnect Options’. In: *High Performance - Consortium for Advanced Scientific and Technical Computing User Group Meeting (HP-CAST 27)*. Nov. 2016.

[15] Sarah Neuwirth and Tobias Groschup. ‘Evaluation of Lustre RDMA Performance over Extoll’. In: *ISC High Performance Conference 2018 (ISC18), Exhibitor Forum*. June 2018.

[16] Sarah Neuwirth, Tobias Groschup, and Ulrich Bruening. ‘Extoll Lustre Network Driver – Overview and Preliminary Results’. In: *Lustre Administrator and Developer Workshop 2018 (LAD’18)*. Sept. 2018.

[17] Sarah Neuwirth, Feiyi Wang, Sarp Oral, Sudharshan Vazhkudai, James Rogers, and Ulrich Bruening. ‘Using Balanced Data Placement to Address I/O Contention in Production Environments’. In: *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Oct. 2016, pp. 9–17. DOI: 10.1109/SBAC-PAD.2016.10.

[18] Sarah Neuwirth, Feiyi Wang, Sarp Oral, and Ulrich Bruening. 'Automatic and Transparent Resource Contention Mitigation for Improving Large-scale Parallel File System Performance'. In: *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. Dec. 2017, pp. 604–613. DOI: 10.1109/ICPADS.2017.00084.

[19] Sarah Neuwirth, Feiyi Wang, Sarp Oral, and Ulrich Bruening. 'An I/O Load Balancing Framework for Large-scale Applications (BPIO 2.0)'. In: *International Conference for High Performance Computing, Networking, Storage and Analysis (SC16)*. Poster paper. 2016.

[20] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN: 978-0080573076.

[21] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003. ISBN: 978-0122007514.

[22] Tse-yun Feng. 'A Survey of Interconnection Networks'. In: *Computer* 14.12 (Dec. 1981), pp. 12–27. ISSN: 0018-9162. DOI: 10.1109/C-M.1981.220290.

[23] Ulrich Bruening. *Parallel Computer Architecture*. Lecture Notes. 2017.

[24] Jose Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: An Engineering Approach*. 1st. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997. ISBN: 978-0818678004.

[25] MPI Forum. *MPI: A Message-Passing Interface Standard – Version 1.3*. URL: https://www.mpi-forum.org/docs/ (visited on 07/08/2018).

[26] D. B. Gustavson. 'The Scalable Coherent Interface and related standards projects'. In: *IEEE Micro* 12.1 (Feb. 1992), pp. 10–22. ISSN: 0272-1732. DOI: 10.1109/40.124376.

[27] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Elsevier, 2011. ISBN: 978-8178672663.

[28] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 'LogP: Towards a Realistic Model of Parallel Computation'. In: *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '93. San Diego, California, USA: ACM, 1993, pp. 1–12. ISBN: 0-89791-589-5. DOI: 10.1145/155332.155333.

[29] *MPICH – High-Performance Portable MPI*. URL: https://www.mpich.org/ (visited on 07/20/2018).

[30] Network-Based Computing Laboratory. *MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE*. URL: http://mvapich.cse.ohio-state.edu/ (visited on 07/20/2018).

[31] Software in the Public Interest (SPI). *Open MPI: Open Source High Performance Computing*. URL: https://www.open-mpi.org/ (visited on 07/20/2018).

[32] Katherine Yelick et al. 'Productivity and Performance Using Partitioned Global Address Space Languages'. In: *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*. PASCO '07. London, Ontario, Canada: ACM, 2007, pp. 24–32. ISBN: 978-1-59593-741-4. DOI: 10.1145/1278177.1278183.

[33] UPC Consortium. *UPC Language Specifications Version 1.3*. URL: https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf (visited on 07/22/2018).

[34] Robert W. Numrich and John Reid. 'Co-array Fortran for Parallel Programming'. In: *SIGPLAN Fortran Forum* 17.2 (Aug. 1998), pp. 1–31. ISSN: 1061-7264. DOI: 10.1145/289918.289920.

[35] LBNL and U.C. Berkeley. *GASNet Specification, Version 1.8.1*. URL: https://gasnet.lbl.gov/dist/docs/gasnet.html (visited on 07/22/2018).

[36] High Performance Computing Tools group at the University of Houston and Extreme Scale Systems Center, Oak Ridge National Laboratory. *OpenSHMEM Application Programming Interface, Version 1.3*. URL: http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.3.pdf (visited on 07/22/2018).

[37] GASPI-Forum. *GASPI: Global Address Space Programming Interface – Specication of a PGAS API for communication, Version 17.1*. Feb. 2017.

[38] Andrew D. Birrell and Bruce Jay Nelson. 'Implementing Remote Procedure Calls'. In: *ACM Transactions on Computer Systems (TOCS)* 2.1 (Feb. 1984), pp. 39–59. ISSN: 0734-2071. DOI: 10.1145/2080.357392.

[39] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Version 0.92. Arpaci-Dusseau Books, 2015.

[40] Sandia National Laboratories (SNL). *The Portals 4.1 Network Programming Interface*. URL: `http://www.cs.sandia.gov/Portals/portals41.pdf` (visited on 07/22/2018).

[41] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API*. Addison-Wesley Professional Computing Series. Addison-Wesley, 2003. ISBN: 978-0131411555.

[42] Robert Latham, Robert Ross, Brent Welch, and Katie Antypas. *Parallel I/O in Practice*. Tutorial held in conjunction with SC12 International Conference for High Performance Computing, Networking, Storage and Analysis. Nov. 2012.

[43] Prabhat and Quincey Koziol. *High Performance Parallel I/O*. Ed. by Horst Simon. Chapman & Hall/CRC Computational Science. CRC Press, Oct. 2014. ISBN: 978-1466582347.

[44] NERSC. *Introduction to Scientific I/O*. URL: `http://www.nersc.gov/users/training/online-tutorials/introduction-to-scientific-i-o/` (visited on 07/09/2018).

[45] Benjamin Depardon, Gaël Le Mahec, and Cyril Séguin. *Analysis of Six Distributed File Systems*. Research Report. SysFera, Feb. 2013, p. 44. URL: `https://hal.inria.fr/hal-00789086`.

[46] Eliezer Levy and Abraham Silberschatz. 'Distributed File Systems: Concepts and Examples'. In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 321–374. ISSN: 0360-0300. DOI: `10.1145/98163.98169`.

[47] T. D. Thanh, S. Mohan, E. Choi, S. Kim, and P. Kim. 'A Taxonomy and Survey on Distributed File Systems'. In: *2008 Fourth International Conference on Networked Computing and Advanced Information Management*. Vol. 1. Sept. 2008, pp. 144–149. DOI: `10.1109/NCM.2008.162`.

[48] IEEE and The Open Group. *POSIX.1-2017: The Open Group Base Specifications Issue 7, 2018 edition, IEEE Standard 1003.1-2017*. URL: `http://pubs.opengroup.org/onlinepubs/9699919799/` (visited on 07/18/2018).

[49] Argonne National Laboratory. *ROMIO: A High-Performance, Portable MPI-IO Implementation*. URL: `http://www.mcs.anl.gov/projects/romio/` (visited on 07/18/2018).

[50] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. 'An Overview of the HDF5 Technology Suite and Its Applications'. In: *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. AD '11. Uppsala, Sweden: ACM, 2011, pp. 36–47. ISBN: 978-1-4503-0614-0. DOI: 10.1145/1966895.1966900.

[51] The HDF Group. *HDF5 File Format Specification Version 3.0*. URL: https://support.hdfgroup.org/HDF5/doc/H5.format.html (visited on 07/17/2018).

[52] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. 'Flexible IO and Integration for Scientific Codes Through The Adaptable IO System (ADIOS)'. In: *6th International Workshop on Challenges of Large Applications in Distributed Environments*. 2008, pp. 15–24. DOI: 10.1145/1383529.1383533.

[53] Oak Ridge National Laboratory (ORNL). *The Adaptable IO System (ADIOS)*. URL: https://www.olcf.ornl.gov/center-projects/adios/ (visited on 03/01/2018).

[54] Wolfgang Frings, Felix Wolf, and Ventsislav Petkov. 'Scalable Massively Parallel I/O to Task-local Files'. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC09)*. SC '09. Portland, Oregon: ACM, 2009, 17:1–17:11. ISBN: 978-1-60558-744-8. DOI: 10.1145/1654059.1654077.

[55] Mondrian B. Nuessle. 'Acceleration of the hardware-software interface of a communication device for parallel system'. PhD thesis. Universität Mannheim, 2009.

[56] M. Nuessle, B. Geib, H. Froening, and U. Bruening. 'An FPGA-Based Custom High Performance Interconnection Network'. In: *2009 International Conference on Reconfigurable Computing and FPGAs*. Dec. 2009, pp. 113–118. DOI: 10.1109/ReConFig.2009.23.

[57] Holger Froening, Mondrian Nuessle, Heiner Litz, Christian Leber, and Ulrich Bruening. 'On Achieving High Message Rates'. In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. May 2013, pp. 498–505. DOI: 10.1109/CCGrid.2013.43.

[58] Juri Schmidt. 'Accelerating Checkpoint/Restart Application Performance in Large-Scale Systems with Network Attached Memory'. PhD thesis. University of Heidelberg, Dec. 2017. DOI: 10.11588/heidok.00023800.

[59] Mondrian Nuessle, Patrick Scherer, and Ulrich Bruening. ‘A resource optimized remote-memory-access architecture for low-latency communication’. In: *The 38th International Conference on Parallel Processing (ICCP-09)*. Sept. 2009. DOI: 10.1109/ICPP.2009.62.

[60] Alexander Giese, Benjamin Kalisch, and Mondrian Nuessle. *RMA2 Specification, Revision 2.0.5*. CAG Confidential.

[61] Heiner Litz, Mondrian Nuessle, Holger Froening, and Ulrich Bruening. ‘VELO: A Novel Communication Engine for Ultra-low Latency Message Transfers’. In: *The 37th International Conference on Parallel Processing (ICCP-08)*. Sept. 2008. DOI: 10.1109/ICPP.2008.85.

[62] H. Froening and H. Litz. ‘Efficient Hardware Support for the Partitioned Global Address Space’. In: *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*. Apr. 2010, pp. 1–6. DOI: 10.1109/IPDPSW.2010.5470851.

[63] David Slogsnat, Alexander Giese, Mondrian Nüssle, and Ulrich Brüning. ‘An Open-source HyperTransport Core’. In: *ACM Trans. Reconfigurable Technol. Syst.* 1.3 (Sept. 2008), 14:1–14:21. ISSN: 1936-7406. DOI: 10.1145/1391732.1391734.

[64] Christian Leber. ‘Efficient Hardware for Low Latency Applications’. PhD thesis. Universität Mannheim, 2012.

[65] Dirk Frey. *Verification and Implementation of PCI Express Endpoint Remote Configuration using EXTOLL*. Diploma Thesis. Universität Mannheim. Oct. 2012.

[66] Heiner Litz. *HyperTransport On-Chip (HTOC) Protocol Specification*. Version 1.6, Computer Architecture Group, University of Heidelberg.

[67] Gregory F. Pfister. ‘An Introduction to the InfiniBand Architecture’. In: *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. Ed. by Rajkumar Buyya, Toni Cortes, and Hai Jin. Wiley-IEEE Press, 2002. Chap. 42, pp. 617–632. ISBN: 9780470544839. DOI: 10.1109/9780470544839.ch42.

[68] HPC Advisory Council. *Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing*. URL: http://www.hpcadvisorycouncil.com/pdf/IB_and_10GigE_in_HPC.pdf (visited on 08/04/2018).

[69] Alex Netes. 'EDR Infiniband'. In: *2015 User Group Workshop*. `https://www.openfabrics.org/images/eventpresos/workshops2015/UGWorkshop/Friday/friday_01.pdf`. Mar. 2015. (Visited on 09/23/2018).

[70] Jeff Hilland, Paul Culley, Jim Pinkerton, and Renato Recio. *RDMA Protocol Verbs Specification Version 1.0*. 2003. URL: `https://tools.ietf.org/html/draft-hilland-rddp-verbs-00` (visited on 08/28/2018).

[71] *RDMA Core Userspace Libraries and Daemons*. URL: `https://github.com/linux-rdma/rdma-core` (visited on 08/24/2018).

[72] PCI-SIG. *PCI Express Base Specification Revision 3.0*. Nov. 2010.

[73] MindShare Inc, Ravi Budruk, Don Anderson, and Tom Shanley. *PCI Express System Architecture*. Ed. by Joe Winkles. Addison-Wesley Developer's Press, 2003. ISBN: 978-0321156303.

[74] Network-Based Computing Laboratory (NBCL), The Ohio State University. *OSU Micro-Benchmarks (OMB)*. URL: `http://mvapich.cse.ohio-state.edu/benchmarks/` (visited on 08/02/2018).

[75] S. Prabhakaran, M. Neumann, S. Rinke, F. Wolf, A. Gupta, and L. V. Kale. 'A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications'. In: *2015 IEEE International Parallel and Distributed Processing Symposium*. May 2015, pp. 429–438. DOI: `10.1109/IPDPS.2015.34`.

[76] *European DEEP Project*. URL: `http://www.deep-project.eu/` (visited on 07/05/2018).

[77] Gene Amdahl. 'Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities'. In: *AFIPS Conference Proceedings*. Vol. 30. 1967, pp. 483–485. DOI: `10.1145/1465482.1465560`.

[78] *DEEP Projects*. URL: `https://www.deep-projects.eu/` (visited on 07/05/2018).

[79] Norbert Eicker and Thomas Lippert. 'An accelerated Cluster-Architecture for the Exascale'. In: *PARS '11, PARS-Mitteilungen, Mitteilungen - Gesellschaft für Informatik e.V., Parallel-Algorithmen und Rechnerstrukturen* 28 (Oct. 2011), pp. 110–119.

[80] Norbert Eicker, Thomas Lippert, Thomas Moschny, and Estela Suarez. ‘The DEEP project: Pursuing cluster-computing in the many-core era’. In: *Proceedings of the 42nd International Conference on Parallel Processing Workshops (ICPPW) 2013, Workshop on Heterogeneous and Unconventional Cluster Architectures and Applications*. 2013, pp. 885–892. DOI: 10.1109/ICPP.2013.105.

[81] Damian A. Mallon, Norbert Eicker, Maria E. Innocenti, Giovanni Lapenta, Thomas Lippert, and Estela Suarez. ‘On the Scalability of the Clusters-booster Concept: A Critical Assessment of the DEEP Architecture’. In: *Proceedings of the Future HPC Systems: The Challenges of Power-Constrained Performance*. 2012. DOI: 10.1145/2322156.2322159.

[82] A. Kreuzer, N. Eicker, J. Amaya, and E. Suarez. ‘Application Performance on a Cluster-Booster System’. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2018, pp. 69–78. DOI: 10.1109/IPDPSW.2018.00019.

[83] Estela Suarez, Norbert Eicker, and Thomas Lippert. ‘Supercomputing Evolution at JSC’. In: vol. 49. Publication Series of the John von Neumann Institute for Computing (NIC) NIC Series. NIC Symposium 2018, Jülich (Germany), 22 Feb 2018 - 23 Feb 2018. John von Neumann Institute for Computing, Feb. 2018, pp. 1–12. URL: http://hdl.handle.net/2128/17546.

[84] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. Ed. by Andy Oram. O’Reilly Media, Feb. 2009. ISBN: 978-0596005900.

[85] David A. Rusling. *The Linux Kernel*. Online Book. 1999. URL: https://www.tldp.org/LDP/tlk/tlk.html (visited on 07/07/2018).

[86] George Chrysos. *Intel Xeon Phi X100 Family Coprocessor - the Architecture*. URL: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner (visited on 07/07/2018).

[87] *Intel Xeon Phi Coprocessor System Software Developers Guide*. Intel Corporation. Mar. 2014.

[88] NVIDIA Corporation. *NVIDIA’s Next Generation CUDA Compute Archiecture: Kepler GK100*. Whitepaper. URL: https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (visited on 09/05/2018).

[89] Min Si and Yutaka Ishikawa. 'Design of Direct Communication Facility for Many-Core Based Accelerators'. In: *IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012*. May 2012, pp. 924–929. DOI: 10.1109/IPDPSW.2012.113.

[90] Min Si, Yutaka Ishikawa, and Masamichi Tatagi. 'Direct MPI Library for Intel Xeon Phi Co-Processors'. In: *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*. May 2013, pp. 816–824. DOI: 10.1109/IPDPSW.2013.179.

[91] Sreeram Potluri, Devendar Bureddy, Khaled Hamidouche, Akshay Venkatesh, Krishna Kandalla, Hari Subramoni, and Dhabaleswar K. Panda. 'MVAPICH-PRISM: A Proxy-based Communication Framework Using InfiniBand and SCIF for Intel MIC Clusters'. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC13)*. SC '13. Denver, Colorado: ACM, 2013, 54:1–54:11. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503288.

[92] Sreeram Potluri, Khaled Hamidouche, Devendar Bureddy, and Dhabaleswar K. Panda. 'MVAPICH2-MIC: A High Performance MPI Library for Xeon Phi Clusters with InfiniBand'. In: *Extreme Scaling Workshop (XSW), 2013*. Aug. 2013, pp. 25–32. DOI: 10.1109/XSW.2013.8.

[93] Matthias Noack. *HAM-Heterogenous Active Messages for Efficient Offloading on the Intel Xeon Phi*. Tech. rep. Konrad-Zuse-Zentrum für Informationstechnik Berlin, June 2014.

[94] Matthias Noack, Florian Wende, Thomas Steinke, and Frank Cordes. 'A unified programming model for intra-and inter-node offloading on Xeon Phi clusters'. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC14)*. IEEE. 2014, pp. 203–214. DOI: 10.1109/SC.2014.22.

[95] Antonio J. Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S. Quintana-Ortí, and José Duato. 'A complete and efficient CUDA-sharing solution for HPC clusters'. In: *Parallel Computing* 40.10 (2014), pp. 574–588. ISSN: 0167-8191. DOI: 10.1016/j.parco.2014.09.011.

[96] Amnon Barak and Amnon Shiloh. *The VirtualCL (VCL) cluster platform*. 2013.

[97] Alex Herrera. 'NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation'. In: *Nvidia Corp* (2014).

[98] NVIDIA Corporation. *Developing a Linux Kernel Module using RDMA for GPUDirect*. URL: http://docs.nvidia.com/cuda/gpudirect-rdma/index.html (visited on 07/03/2018).

[99] Lena Oden and Holger Froning. 'GGAS: Global GPU address spaces for efficient communication in heterogeneous clusters'. In: *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2013, pp. 1–8. DOI: 10.1109/CLUSTER.2013.6702638.

[100] H. Baier et al. 'QPACE: power-efficient parallel architecture based on IBM PowerXCell 8i'. English. In: *Computer Science - Research and Development* 25.3-4 (2010), pp. 149–154. ISSN: 1865-2034. DOI: 10.1007/s00450-010-0122-4.

[101] *The Green500 List*. URL: http://www.green500.org/ (visited on 07/05/2018).

[102] Ahmed Bu-Khamsin. 'Socket Direct Protocol over PCI Express Interconnect: Design, Implementation and Evaluation'. MA thesis. Simon Fraser University, Canada, 2012.

[103] Akber Kazmi. 'PCI Express and Non-Transparent Bridging support High Availability'. In: *Embedded Computing Design* (Nov. 2004). URL: http://embedded-computing.com/pdfs/PLXTech.Win04.pdf.

[104] David Mayhew and Venkata Krishnan. 'PCI Express and Advanced Switching: Evolutionary Path to Building Next Generation Interconnects'. In: *Proceedings of the 11th Symposium on High Performance Interconnects (HOTI'03)*. 2003. DOI: 0-7695-2012-X/03.

[105] NVIDIA Corporation. *NVIDIA NVSwitch: The World's Highest-Bandwidth On-Node Switch*. Technical Overview. URL: https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf (visited on 09/05/2018).

[106] NVIDIA Corporation. *NVIDIA DGX-2: The World's Most Powerful Deep Learning System for the Most Complex AI Challenges*. Data sheet. 2018.

[107] *Galibier Overview*. EXTOLL GmbH. URL: http://extoll.de/index.php/productsoverview/galibier (visited on 03/01/2018).

[108] OpenFabrics Alliance. *OpenFabrics Enterprise Distribution (OFED) / OpenFabrics Software Overview*. URL: https://www.openfabrics.org/ofed-for-linux/ (visited on 08/22/2018).

[109] *Intel MPI Benchmarks User Guide and Methodoly Description*. Intel Corporation. 2016.

[110] Sandia National Laboratory (SNL). *LAMMPS Molecular Dynamics Simulator.* URL: `http://lammps.sandia.gov/` (visited on 07/05/2018).

[111] Ulrich Bruening, Mondrian Nuessle, and Dirk Frey. 'An Immersive Cooled Implementation of a DEEP Booster'. In: *Intel European Exascale Labs Annual Report 2014* (July 2015).

[112] Kilian Polyak. *GPUNAA – Network-Attached Accelerators Architecture with GPGPUs.* University of Mannheim. Diploma thesis (supervised by Sarah Neuwirth and Dirk Frey). June 2016.

[113] Sudhanshu Goswami. 'An Introduction to KProbes'. In: *LWN.net* (Apr. 2005).

[114] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 'The Scalable Heterogeneous Computing (SHOC) Benchmark Suite'. In: *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units.* GPGPU-3. ACM, 2010, pp. 63–74. DOI: `10.1145/1735688.1735702`.

[115] Future Technologies Group @ ORNL. *The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite.* URL: `https://github.com/vetter/shoc` (visited on 07/11/2018).

[116] Lena Oden. 'Direct Communication Models for Distributed GPUs'. PhD thesis. Heidelberg University, Feb. 2015.

[117] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda. 'Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs'. In: *2013 42nd International Conference on Parallel Processing.* Oct. 2013, pp. 80–89. DOI: `10.1109/ICPP.2013.17`.

[118] Michael Kerrisk. *The Linux Programming Interface – A Linux and UNIX System Programming Handbook.* Ed. by Riley Hoffman. No Starch Press, Inc., 2010. ISBN: 978-1593272203.

[119] S. Deering and R. Hinden. *Internet Protocol, Version 6.* RFC8200. July 2017. URL: `https://tools.ietf.org/html/rfc8200` (visited on 07/15/2018).

[120] Information Sciences Institute, University of Southern California. *Internet Protocol.* Ed. by Jon Postel. RFC791. Sept. 1981. URL: `https://tools.ietf.org/html/rfc791` (visited on 07/15/2018).

[121] Information Sciences Institute, University of Southern California. *Transmission Control Protocol.* Ed. by Jon Postel. RFC793. Sept. 1981. URL: `https://tools.ietf.org/html/rfc793` (visited on 07/15/2018).

[122] Jon Postel. *User Datagram Protocol*. RFC768. Aug. 1980. URL: `https://tools.ietf.org/html/rfc768` (visited on 07/16/2018).

[123] Hyeongyeop Kim. *Understanding TCP/IP Network Stack & Writing Network Apps*. Feb. 2013. URL: `http://www.cubrid.org/blog/dev-platform/understanding-tcp-ip-network-stack/` (visited on 07/18/2018).

[124] Jeffrey C. Mogul and K. K. Ramakrishnan. 'Eliminating Receive Livelock in an Interrupt-driven Kernel'. In: *ACM Transactions on Computer Systems (TOCS)* 15.3 (Aug. 1997), pp. 217–252. ISSN: 0734-2071. DOI: `10.1145/263326.263335`.

[125] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. 'Beyonf Softnet'. In: *5th Annual Linux Showcase and Conference*. USENIX Association. 2001, pp. 165–172.

[126] A. Beifuß, D. Raumer, P. Emmerich, T. M. Runge, F. Wohlfart, B. E. Wolfinger, and G. Carle. 'A Study of Networking Software Induced Latency'. In: *2015 International Conference and Workshops on Networked Systems (NetSys)*. Mar. 2015, pp. 1–8. DOI: `10.1109/NetSys.2015.7089065`.

[127] Steen Larsen, Parthasarathy Sarangam, Ram Huggahalli, and Siddharth Kulkarni. 'Architectural Breakdown of End-to-End Latency in a TCP/IP Network'. In: *International Journal of Parallel Programming* 37.6 (Dec. 2009), pp. 556–571. ISSN: 1573-7640. DOI: `10.1007/s10766-009-0109-6`.

[128] Nathan Hanford, Vishal Ahuja, Matthew K. Farrens, Brian Tierney, and Dipak Ghosal. 'A Survey of End-System Optimizations for High-Speed Networks'. In: *ACM Computing Surveys (CSUR)* 51.3 (July 2018), 54:1–54:36. ISSN: 0360-0300. DOI: `10.1145/3184899`.

[129] Robert Alverson, Duncan Roweth, and Larry Kaplan. 'The Gemini System Interconnect'. In: *18th IEEE Symposium on High Performance Interconnects (HOTI '10)*. 2010, pp. 83–87. DOI: `10.1109/HOTI.2010.23`.

[130] J. Chu and V. Kashyap. *Transmission of IP over InfiniBand (IPoIB)*. RFC4391. Apr. 2006. URL: `https://tools.ietf.org/html/rfc4391` (visited on 08/22/2018).

[131] V. Kashyap. *IP over InfiniBand (IPoIB) Architecture*. RFC4392. Apr. 2006. URL: `https://tools.ietf.org/html/rfc4392` (visited on 08/22/2018).

[132] V. Kashyap. *IP Over InfiniBand: Connected Mode*. RFC4755. Dec. 2006. URL: `https://tools.ietf.org/html/rfc4755` (visited on 08/31/2018).

[133] Tzahi Oved. 'User Mode Ethernet Verbs'. In: *12th Annual OpenFabrics Alliance Workshop*. Apr. 2016.

[134] Tzahi Oved and Alex Rosenbaum. 'User Space IPoIB Packet Processing'. In: *13th Annual OpenFabrics Alliance Workshop*. Mar. 2017.

[135] Tzahi Oved and Rony Efraim. 'IPoIB Acceleration'. In: *13th Annual OpenFabrics Alliance Workshop*. Mar. 2017.

[136] Evgenii Smirnov and Mikhail Sennikovsky. 'Ethernet over Infiniband'. In: *14th Annual OpenFabrics Alliance Workshop*. Apr. 2018.

[137] Infiniband Trade Associaton. *Supplement to InfiniBand Architecture Specication Volume 1, Release 1.2.1: Annex A4: Sockets Direct Protocol (SDP)*. Oct. 2011.

[138] D. Goldenberg, M. Kagan, R. Ravid, and M. S. Tsirkin. 'Zero copy sockets direct protocol over infiniband-preliminary implementation and performance analysis'. In: *13th Symposium on High Performance Interconnects (HOTI'05)*. Aug. 2005, pp. 128–137. DOI: 10.1109/CONECT.2005.35.

[139] P. Balaji, S. Bhagvat, H. W. Jin, and D. K. Panda. 'Asynchronous zero-copy communication for synchronous sockets in the sockets direct protocol (SDP) over InfiniBand'. In: *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*. Apr. 2006. DOI: 10.1109/IPDPS.2006.1639560.

[140] Sean Hefty. 'RSockets'. In: *2012 OFS Developers' Workshop*. 2012.

[141] Network Working Group. *Stream Control Transmission Protocol*. RFC4960. Sept. 2007. URL: https://tools.ietf.org/html/rfc4960 (visited on 08/22/2018).

[142] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. *A Remote Direct Memory Access Protocol Specification*. RFC5040. Oct. 2007. URL: https://tools.ietf.org/html/rfc5040 (visited on 08/20/2018).

[143] H. Shah, J. Pinkerton, R. Recio, and P. Culley. *Direct Data Placement over Reliable Transports*. RFC5041. Oct. 2007. URL: https://tools.ietf.org/html/rfc5041 (visited on 08/20/2018).

[144] P. Culley, U. Elzur, R. Recio, S. Bailey, and J. Carrier. *Marker PDU Aligned Framing for TCP Specification*. RFC5044. Oct. 2007. URL: https://tools.ietf.org/html/rfc5044 (visited on 08/20/2018).

[145] C. Bestler and R. Stewart. *Stream Control Transmission Protocol (SCTP) Direct Data Placement (DDP) Adaptation*. RFC5043. Oct. 2007. URL: https://tools.ietf.org/html/rfc5043 (visited on 08/20/2018).

[146] InfiniBand Trade Association. *Supplement to InfiniBand Architecture Specication Volume 1 Release 1.2.1, Annex A17: RoCEv2*. Sept. 2014.

[147] Markus Fischer. 'GMSOCKS - A Direct Sockets Implementation on Myrinet'. In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. Oct. 2001, pp. 204–211. DOI: 10.1109/CLUSTR.2001.959979.

[148] Galen Hunt and Doug Brubacher. 'Detours: Binaryinterception of Win32 functions'. In: *3rd Usenix Windows NT Symposium*. 1999.

[149] *Mellanox's Messaging Accelerator (VMA)*. URL: http://www.mellanox.com/vma (visited on 07/31/2018).

[150] Y. Lin, J. Han, J. Gao, and X. He. 'uStream: A User-Level Stream Protocol over Infiniband'. In: *2009 15th International Conference on Parallel and Distributed Systems*. Dec. 2009, pp. 65–71. DOI: 10.1109/ICPADS.2009.105.

[151] 'IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture'. In: *IEEE Std 802-2001 (Revision of IEEE Std 802-1990)* (Feb. 2002), pp. 1–48. DOI: 10.1109/IEEESTD.2002.93395.

[152] David C. Plummer. *Ethernet Address Resolution Protocol – or – Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*. RFC826. Nov. 1982. URL: https://tools.ietf.org/html/rfc826 (visited on 08/23/2018).

[153] Publich Technical Identifiers. *Internet Assigned Numbers Authority*. URL: https://www.iana.org/ (visited on 08/31/2018).

[154] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. *Neighbor Discovery for IP version 6 (IPv6)*. RFC4861. Sept. 2007. URL: https://tools.ietf.org/html/rfc4861 (visited on 08/25/2018).

[155] S. Deering. *Host Extensions for IP Multicasting*. RFC1112. Aug. 1989. URL: https://tools.ietf.org/html/rfc1112 (visited on 08/25/2018).

[156] Jonathan Corbet. 'Zero-Copy Networking'. In: *LWN.net* (July 2017).

[157] W. Richard Stevens and Gary R. Wright. *TCP/IP Illustrated (Vol. 2): The Implementation*. Addison-Wesley, 1995. ISBN: 0-201-63354-X.

[158] Timothy W Curry et al. 'Profiling and Tracing Dynamic Library Usage Via Interposition'. In: *USENIX Summer*. 1994, pp. 267–278.

[159] Linux Manual Page. *ld.so, ld-linux.so - dynamic linker/loader*. URL: http://man7.org/linux/man-pages/man8/ld.so.8.html (visited on 07/23/2018).

[160] Free Software Foundation, Inc. *GNU Binutils Version 2.31*. URL: https://sourceware.org/binutils/docs-2.31/ (visited on 07/23/2018).

[161] Hewlett-Packard. *Netperf*. URL: https://github.com/HewlettPackard/netperf (visited on 09/28/2018).

[162] Mellanox Technologies. *SockPerf Network Benchmarking Utility*. https://github.com/Mellanox/sockperf. Sept. 2018.

[163] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. June 2016.

[164] Alois Kraus. *Why Skylake CPUs Are Sometimes 50% Slower – How Intel Has Broken Existing Code*. URL: https://aloiskraus.wordpress.com/2018/06/16/why-skylakex-cpus-are-sometimes-50-slower-how-intel-has-broken-existing-code/ (visited on 09/24/2018).

[165] Lustre Community. *Introduction to Lustre Architecture*. Oct. 2017. URL: http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf (visited on 04/28/2018).

[166] Peter J. Braam et al. *The Lustre Storage Architecture*. 2004.

[167] Feiyi Wang, Sarp Oral, and Galen Shipman. *Understanding Lustre Filesystems Internals*. Tech. rep. Oak Ridge National Laboratory, Apr. 2009.

[168] *Lustre Operations Manual 2.x*. 2018.

[169] *Lustre Networking – High-Performance Features and Flexible Support for a Wide Array of Network*. Nov. 2008.

[170] Intel. *Intel Lustre System and Network Administration – Introduction to Lustre Network*. July 2015. URL: https://nci.org.au/wp-content/uploads/2015/08/02-Introduction-LNET.pdf (visited on 06/22/2018).

[171] Peter J. Braama, Phil Schwan, and Ron Brightwell. *Portals and Networking for the Lustre File System*.

[172] Tobias Groschup. *Implementation and Evaluation of a Parallel Distributed File System for the EXTOLL High-Performance Network*. University of Heidelberg. Master thesis (supervised by Sarah Neuwirth). Mar. 2014.

[173] James Simmons and John Lewis. 'Taking Advantage of Multi-cores for the Lustre Gemini LND Driver'. In: *Cray User Group Conference (CUG 2013)*. 2013.

[174] Gregoire Pichon. 'Portals4 LND Overview'. In: *Lustre Administrator and Developer Workshop 2017 (LAD'17)*. Oct. 2017.

[175] S. Derradji, T. Palfer-Sollier, J. Panziera, A. Poudes, and F. W. Atos. 'The BXI Interconnect Architecture'. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects (HOTI)*. Aug. 2015, pp. 18–25. DOI: 10.1109/HOTI.2015.15.

[176] David A. Deming. 'InfiniBand Software Architecture and RDMA'. In: *2013 Storage Developer Conference (SDC13)*. Sept. 2013.

[177] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan S. Vazhkudai. 'Improving Large-scale Storage System Performance via Topology-aware and Balanced Data Placement'. In: *20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 2014, pp. 656–663. DOI: 10.1109/PADSW.2014.7097866.

[178] Arthur S. Bland, Jack C. Wells, Otis E. Messer, Oscar R. Hernandez, and James H. Rogers. 'Titan: Early Experience with the Cray XK6 at Oak Ridge National Laboratory'. In: *Cray User Group Conference (CUG)*. 2012.

[179] Oak Ridge Leadership Computing Facility. *Titan, Cray XK7 Compute System*. URL: https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/ (visited on 08/29/2018).

[180] Sarp Oral, David A. Dillow, Douglas Fuller, Jason Hill, Dustin Leverman, Sudharshan S. Vazhkudai, Feiyi Wang, Youngjae Kim, James Rogers, James Simmons, et al. 'OLCF's 1 TB/s, Next-generation Lustre File System'. In: *Cray User Group Conference (CUG 2013)*. 2013.

[181] Rajeev Thakur, William Gropp, and Ewing Lusk. 'On Implementing MPI-IO Portably and with High Performance'. In: *6th Workshop on I/O in Parallel and Distributed Systems*. 1999, pp. 23–32. DOI: 10.1145/301816.301826.

[182] Bing Xie, Yezhou Huang, Jeffrey S. Chase, Jong Youl Choi, Scott Klasky, Jay Lofstead, and Sarp Oral. 'Predicting Output Performance of a Petascale Supercomputer'. In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. Washington, DC,

USA, 2017, pp. 181–192. ISBN: 978-1-4503-4699-3. DOI: 10.1145/3078597.3078614.

[183] Sean Ahern et al. *Scientific Discovery at the Exascale: Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis and Visualization*. Tech. rep. 2011. URL: http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Exascale-ASCR-Analysis.pdf (visited on 03/05/2018).

[184] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu. 'On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems'. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2016, pp. 750–759. DOI: 10.1109/IPDPS.2016.50.

[185] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. 'Scheduling the I/O of HPC Applications Under Congestion'. In: *2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2015, pp. 1013–1022. DOI: 10.1109/IPDPS.2015.116.

[186] Stephen Herbein, Dong H. Ahn, Don Lipari, Thomas R. W. Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. 'Scalable I/O-Aware Job Scheduling for Burst Buffer Enabled HPC Clusters'. In: *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '16. Kyoto, Japan: ACM, 2016, pp. 69–80. ISBN: 978-1-4503-4314-5. DOI: 10.1145/2907294.2907316.

[187] Miao Luo, Dhabaleswar K. Panda, Khaled Z. Ibrahim, and Costin Iancu. 'Congestion Avoidance on Manycore High Performance Computing Systems'. In: *26th ACM International Conference on Supercomputing (ICS '12)*. 2012, pp. 121–132. DOI: 10.1145/2304576.2304594.

[188] Nan Jiang, Larry Dennison, and William J. Dally. 'Network Endpoint Congestion Control for Fine-grained Communication'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC15)*. SC '15. Austin, Texas: ACM, 2015, 35:1–35:12. ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807600.

[189] Yan Li, Xiaoyuan Lu, Ethan Miller, and Darrell Long. 'ASCAR: Automating contention management for high-performance storage systems'. In: *31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2015, pp. 1–16. DOI: 10.1109/MSST.2015.7208287.

[190] Yifeng Zhu, Hong Jiang, Xiao Qin, Dan Feng, and David R. Swanson. 'Improved Read Performance in a Cost-effective, Fault-tolerant Parallel Virtual File System (CEFT-PVFS)'. In: *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*. 2003, pp. 730–735. DOI: 10.1109/CCGRID.2003.1199440.

[191] Robert B. Ross, Rajeev Thakur, et al. 'PVFS: A Parallel File System for Linux Clusters'. In: *4th Annual Linux Showcase and Conference*. 2000, pp. 391–430.

[192] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. 'Server-storage Virtualization: Integration and Load Balancing in Data Centers'. In: *2008 ACM/IEEE Conference on Supercomputing (SC08)*. 2008, 53:1–53:12. DOI: 10.1109/SC.2008.5222625.

[193] David Dillow, Galen M. Shipman, Sarp Oral, Zhe Zhang, Youngjae Kim, et al. 'Enhancing I/O Throughput via Efficient Routing and Placement for Large-scale Parallel File Systems'. In: *IEEE 30th International Performance Computing and Communications Conference (IPCCC)*. 2011, pp. 1–9. DOI: 10.1109/PCCC.2011.6108062.

[194] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. 'A Multiplatform Study of I/O Behavior on Petascale Supercomputers'. In: *24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*. 2015, pp. 33–44. DOI: 10.1145/2749246.2749269.

[195] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. 'Understanding and Improving Computational Science Storage Access through Continuous Characterization'. In: *ACM Transactions on Storage (TOS)* 7.3 (Oct. 2011), 8:1–8:26. DOI: 10.1145/2027066.2027068.

[196] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. 'Managing Variability in the IO Performance of Petascale Storage Systems'. In: *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*. 2010, pp. 1–12. ISBN: 978-1-4244-7559-9. DOI: 10.1109/SC.2010.32.

[197] Qing Liu, Norbert Podhorszki, Jeremy Logan, and Scott Klasky. 'Runtime I/O Re-Routing + Throttling on HPC Storage'. In: *5th USENIX Workshop*

on Hot Topics in Storage and File Systems (HotStorage '13). San Jose, CA, 2013.

[198] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 'End-to-end Performance Isolation Through Virtual Datacenters'. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. 2014, pp. 233–248.

[199] Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. 'Decentralized Task-aware Scheduling for Data Center Networks'. In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM, 2014, pp. 431–442. ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2626322.

[200] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. 'Network-aware Scheduling for Data-parallel Jobs: Plan When You Can'. In: *2015 ACM Conference on Special Interest Group on Data Communication*. 2015, pp. 407–420. DOI: 10.1145/2829988.2787488.

[201] Katherine Yelick et al. *The Magellan Report on Cloud Computing for Science*. Research rep. http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Magellan_Final_Report.pdf. 2011. (Visited on 08/29/2018).

[202] Galen Shipman, D. Dillow, Sarp Oral, and Feiyi Wang. 'The Spider Center Wide File System: From Concept to Reality'. In: *Cray User Group Conference (CUG 2009)*. 2009.

[203] The National Institute for Computational Sciences, University of Tennessee at Knoxville. *Lustre Striping Guide*. URL: https://www.nics.tennessee.edu/computing-resources/file-systems/lustre-striping-guide (visited on 08/31/2018).

[204] W. K. Liao and A. Choudhary. 'Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols'. In: *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2008, pp. 1–12. DOI: 10.1109/SC.2008.5222722.

[205] Sarp Oral. 'OLCF I/O Best Practices'. In: *OLCF User Group Meeting*. June 2015.

[206] Sarp Oral et al. 'Best Practices and Lessons Learned from Deploying and Operating Large-scale Data-centric Parallel File Systems'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. New Orleans, Louisana: IEEE Press, 2014, pp. 217–228. ISBN: 978-1-4799-5500-8. DOI: 10.1109/SC.2014.23.

[207] Tom Edwards. *An Introduction to the Lustre Parallel File System*. Sept. 2014.

[208] D. A. Dillow, G. M. Shipman, S. Oral, and Z. Zhang. 'I/O Congestion Avoidance via Routing and Object Placement'. In: *Cray User Group Conference (CUG 2011)*. 2011.

[209] Arthur S. Bland, Wayne Joubert, Ricky A. Kendall, Douglas B. Kothe, James H. Rogers, and Galen M. Shipman. 'Jaguar: The World's Most Powerful Computer System – An Update'. In: *Cray User Group Conference (CUG 2010)*. 2010.

[210] R. Brightwell, K. Pedretti, and K. D. Underwood. 'Initial Performance Evaluation of the Cray SeaStar Interconnect'. In: *13th Symposium on High Performance Interconnects (HOTI'05)*. Aug. 2005, pp. 51–57. DOI: 10.1109/CONECT.2005.24.

[211] Matt Ezell, Dave Dillow, Sarp Oral, Feiyi Wang, Devesh Tiwari, Don E. Maxwell, Dustin Leverman, and Jason Hill. 'I/O Router Placement and Fine-Grained Routing on Titan to Support Spider II'. In: *Cray User Group Conference (CUG 2014)*. 2014.

[212] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 'Fastpass: A Centralized "Zero-queue" Datacenter Network'. In: *2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. 2014, pp. 307–318. DOI: 10.1145/2619239.2626309.

[213] OpenSFS. *Lustre 2.7.0 Released*. Mar. 2015. URL: http://lustre.org/lustre-2-7-0-released/ (visited on 06/24/2018).

[214] Richard Mohr, Michael J. Brim, Sarp Oral, and Andreas Dilger. 'Evaluating Progressive File Layouts For Lustre'. In: *Cray User Group Conference (CUG 2016)*. 2016.

[215] Huong Luu, Babak Behzad, Ruth Aydt, and Marianne Winslett. 'A multi-level approach for understanding I/O activity in HPC applications'. In: *IEEE International Conference on Cluster Computing (CLUSTER)*. 2013, pp. 1–5.

[216] Lawrence Livermore National Laboratory (LLNL). *Interleaved Or Random (IOR) Benchmark*. URL: `https://github.com/LLNL/ior` (visited on 08/03/2018).

[217] Hongzhang Shan, Katie Antypas, and John Shalf. 'Characterizing and Predicting the I/O Performance of HPC Applications using a Parameterized Synthetic Benchmark'. In: *2008 ACM/IEEE Conference on Supercomputing (SC08)*. 2008, 42:1–42:12. DOI: `10.1109/SC.2008.5222721`.

[218] Jacqueline H. Chen, Alok Choudhary, B. DeSupinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, et al. 'Terascale Direct Numerical Simulations of Turbulent Combustion using S3D'. In: *Computational Science & Discovery* 2.1 (Jan. 2009), p. 015001.

[219] J. Logan et al. 'Skel: Generative Software for Producing Skeletal I/O Applications'. In: *2011 IEEE Seventh International Conference on e-Science Workshops*. Dec. 2011, pp. 191–198. DOI: `10.1109/eScienceW.2011.26`.

[220] Cristina Manzano. 'BeeGFS in the DEEP/-ER Project'. In: BeeGFS User Meeting 2016, Kaiserslautern (Germany), 18 May 2016 - 19 May 2016. May 18, 2016. URL: `http://juser.fz-juelich.de/record/811106`.