Elastic Fabric Adapter: A Viable Alternative to RDMA over InfiniBand for DBMS?

Master thesis by Dwarakanandan B.M
Date of submission: April 25, 2022

1. Review: Prof. Dr. Carsten Binnig
2. Review: Tobias Ziegler

Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department
Data Management Lab

Erklärung zur Abschlussarbeit
gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Dwarakanandan B.M, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 25. April 2022

Signature

Abstract

High-performance networking has witnessed substantial growth in recent years with technologies like Remote Direct Memory Access (RDMA) over InfiniBand offering exceptionally low latencies. This has had a profound impact on data-intensive system design with proven tangible benefits for distributed Database Management Systems (DBMS). At the same time, we observe wide scale adoption of cloud computing services with major data-centric systems being deployed on the cloud. However, RDMA over InfiniBand is still not widely available in the cloud. Instead, cloud providers often deploy their own high-speed networking technology and expose proprietary networking interfaces. In 2019 the largest cloud provider, Amazon Web Services (AWS), introduced a new network fabric called Elastic Fabric Adapter (EFA) aiming to provide more consistent lower latencies coupled with the cloud native Scalable Reliable Datagram (SRD) protocol. In this thesis, we aim to analyze EFA as an alternative to RDMA in the cloud by performing a systematic evaluation. We start with a brief overview of current network fabrics offered in the cloud, including a description of EFA's hardware and software stack. Followed by an in-depth performance evaluation of EFA by comparing it with RDMA over InfiniBand on various network parameters. Finally drawing from our evaluation, we highlight its implications for data-intensive system design.

Contents

List of Figures 1

Listings 2

List of Abbreviations 3

1. Introduction 4

1.1. Context and Motivation 4
1.2. Problem Statement 5
1.3. Goals and Contributions 6

2. Background 7

2.1. High-Performance Networks in the Cloud 7
2.2. User-space Networking 8
2.2.1. Direct Hardware access 9
2.2.2. Limit Memory Copy 9
2.2.3. Asynchronous model 9
2.3. Amazon Web Services 10
2.3.1. Elastic Compute Cloud - EC2 10
2.3.2. Virtual Private Cloud - VPC 10
2.3.3. Security Groups 10
2.3.4. Placement Strategy 10
2.4. Elastic Fabric Adapter 11
2.4.1. SRD Protocol and Driver Modules 11
2.4.2. ibverbs - EFA's Low Level Interface 12
2.4.3. OpenFabrics *libfabric* - EFA's High Level Interface 12
2.4.4. libfabric - RDM Communication Protocol v4 13
2.5. RDMA and Sockets 14

3. Implementation 15

3.1. Overview of Existing Projects 15
3.1.1. InfiniBand Verbs Performance Tests (*perftest*) 15
3.1.2. OSU Micro-Benchmarks (*OMB*) 15
3.2. Overview of *libefa* and *efa-bench* 16
3.2.1. *libefa* - A Wrapper Around *libfabric* for EFA 16
3.2.2. *efa-bench* - A Benchmarking Suite for EFA 18
3.2.3. Latency Micro-benchmark 18
3.2.4. Bandwidth Micro-benchmark 19
3.2.5. Message API Variants Micro-benchmark 20

3.2.6. Saturated Bandwidth Micro-benchmark 21
3.2.7. RMA Micro-benchmark 22
3.2.8. Multi-threaded Micro-benchmark 23

4. Evaluation 24
4.1. Setup and Methodology 24
4.1.1. Setup 24
4.1.2. Methodology 25
4.2. Latency Evaluation 26
4.2.1. RC-RDMA vs SRD vs TCP/IP 26
4.2.2. Inline Optimization 26
4.2.3. EFA's SRD vs UD Protocol 27
4.3. Synchronous Bandwidth 28
4.4. Asynchronous Bandwidth - Message Rate 29
4.5. NIC Parallelism 30
4.6. Multi-threaded Evaluation 31
4.7. Message Segmentation Overhead 33
4.8. Link Latency at Saturation 33
4.9. *libfabric* Message Transfer API Variants 34
4.10. One Sided Operations - RMA 35
4.11. Interface Evaluation - *ibverbs* vs *libfabric* vs *MPI* 35

5. Related Work 37

6. Conclusion and Future Work 39
6.1. Implications for System Design 39
6.2. Conclusion 40
6.3. Future Work 41

Bibliography 42

Appendices A. i
A. *Efa-bench* Micro-benchmark Suite . . . . . A. i
B. EFA Fabric Info . . . . . B. i
C. Evaluation Raw Data . . . . . C. i
D. Performance Measurements and Tuning . . . . . D. i

List of Figures

2.1. ENA vs EFA architecture (OS-Bypass) 8
2.2. Azure Accelerated Networking (SR-IOV) 8
2.3. EFA Hardware and Software Stack 11
2.4. Architecture of *libfabric* (OFI working group) [22] 12

3.1. EFA Communication Setup with *libfabric* 16
3.2. Architecture of the *efa-bench* Suite 18

4.1. Evaluation Setup 24
4.2. Impact of Message Size on Average Latency - RC-RDMA vs SRD vs TCP/IP 26
4.3. Relative Impact of Inline Optimization on Latency 27
4.4. EFA's SRD vs UD Protocol - Average Latency 27
4.5. EFA's SRD vs UD Protocol - Minimum Latency 28
4.6. Impact of Message Size on Synchronous Bandwidth 28
4.7. Impact of Transmission Depth (Outstanding Operations) on Async Bandwidth 29
4.8. Impact of Transmission Depth (Outstanding Operations) on Message Rate 30
4.9. Impact of Connection Pairs (Send queue) on Message Rate 31
4.10. Impact of Thread Count on Bandwidth (TX Depth - 128) 31
4.11. Impact of Thread Count on Message Rate (TX Depth - 128) 32
4.12. EFA Bandwidth Scaling w.r.t Thread Count at Various TX Depths (Msg Size - 64 Bytes) 32
4.13. Average Link latency at Segmentation Boundary 33
4.14. Average Link latency at Saturation (Message Size - 4096 Bytes) 34
4.15. *libfabric* Message Transfer API variants Impact on Average Latency 34
4.16. Performance of Emulated RMA Operations Compared to *send/recv* 35
4.17. Overhead of Emulated RMA *read* Compared to RMA *write* (Message Size - 8 kb) . . . 36
4.18. Performance Implication of EFA Interfaces (TX Depth - 256) 36

5.1. RDMA Performance evaluation vs Ethernet-IP (Binnig et.al) [7] 37
5.2. MPI Latency performance on EFA (Sourav et.al) [8] 38
5.3. SRD Flow Fairness compared to TCP/IP (Shalev et.al) [42] 38

D.1. Server Flame Graph iii
D.2. Client Flame Graph iii

Listings

3.1. *libefa* Node setup snippet 17
3.2. Latency micro-benchmark server snippet 18
3.3. Latency Micro-benchmark Client Snippet 19
3.4. Bandwidth Micro-benchmark Server Snippet 19
3.5. Bandwidth Micro-benchmark Client Snippet 20
3.6. Saturated Bandwidth Server Micro-benchmark 21
3.7. Saturated Bandwidth Client Micro-benchmark 21
3.8. RMA Micro-benchmark Server Snippet 22
3.9. RMA Micro-benchmark Client Snippet 22
3.10. Multi-threaded Server Micro-benchmark Snippet 23

List of Abbreviations

ACQ Admin completion queue

AENQ Asynchronous Event Notification Queue

API Application Programming Interface

AQ Admin Queue

AWS Amazon Web Services

DBMS Database Management System

DGRAM Unreliable Datagram

DMA Direct Memory Access

DPU Data Processing Unit

EC2 Elastic Compute Cloud

EFA Elastic Fabric Adapter

ENA Elastic Network Adapter

GCP Google Cloud Platform

HPC High Performance Computing

MPI Message Passing Interface

MTU Maximum Transmission Unit

NIC Network Interface Controller

RDMA Remote Direct Memory Access

RDM Reliable datagram message

RMA Remote Memory Access

SQ Send queues

SR-IOV Single-root input/output virtualization

SRD Scalable Reliable Datagram

TCP/IP Transmission Control Protocol/Internet Protocol

VPC Virtual private cloud

1. Introduction

1.1. Context and Motivation

High-performance networks have seen tremendous growth in recent years. There are various fabrics such as InfiniBand, Fast Ethernet, Fibre Channel and Omni-Path that are commonly used as interconnects for high performance applications. InfiniBand offers two network communication stacks, namely IP over InfiniBand (IPoIB) and Remote Direct Memory Access (RDMA). Notable among these is RDMA over InfiniBand which offers exceptionally low latencies, in the order of a few microseconds [46]. Such trends towards high-speed, low-latency networks have had a profound impact on data-intensive systems. Network fabric which was assumed to be the bottleneck in these systems [1], is no longer the case. This is evident by looking at the redesign of various distributed computing applications [24, 29] and distributed database systems [7, 26, 27, 25].

Cloud computing at it's core is the delivery of computing services over the Internet. Over the years, various different models of cloud computing have evolved, such as Private Cloud model, Public cloud model, Hybrid cloud model, etc. Out of which our focus is mainly on the public cloud. Cloud service providers own and run public clouds, which offer computing resources such as servers and storage over the Internet for an appropriate fee. The provider owns and manages all software, hardware, network infrastructure and other supporting infrastructure in a public cloud. Services offered by such providers can broadly be classified into infrastructure as a service (IaaS), platform as a service (PaaS), server-less computing, and software as a service (SaaS).

Public cloud services have seen wide-scale adoption by companies with major data-centric systems being deployed on the cloud. Factors such as flexibility, cost-savings, scalability, security, disaster recovery and many more have been the driving forces behind this trend. With advancements in technologies like Single-root input/output virtualization (SR-IOV), cloud providers have started pairing high speed network adapters with their virtual machines. Major cloud providers Amazon Web Services (AWS), Google Cloud Platform (GCP) and Microsoft Azure, have offerings such as Enhanced Networking [3], Andromeda [33] and Accelerated Networking [5] respectively. Some of these adapters are capable of [100 Gbit] networking, matching InfiniBand. However the low latencies achieved by RDMA over InfiniBand are unparalleled. Moreover, out of the three big providers only Azure offers InfiniBand in the cloud, and that too with a limited number of instance families [4]. All things considered, network fabric performance has been slow to catch up in the cloud.

Of all the public cloud service providers Amazon Web services (AWS) is by far the largest. AWS needs little introduction due to its immense popularity. Currently it offers more than 200 services spanning virtually across the entire spectrum of computing. The most critical services that form the basis for all other services can broadly be categorized into compute [e.g. AWS EC2, AWS Lambda], storage [e.g. Amazon S3, Amazon EBS, Amazon EFS] and network [e.g. AWS VPC]. With operations spanning the entire globe, AWS maintains multiple data-centers with different geographical locations termed as *AWS Regions*. Within these regions, multiple *Availability Zones* are provided to ensure segregation and fault tolerance. All AWS resources within a data center are inter-connected by Amazon proprietary network fabrics. With this context, we now shift our focus to the main topic at hand: Elastic Fabric Adapter (EFA).

Elastic Fabric Adapter (EFA)

Until recently the main network fabric to support High Performance Computing (HPC) applications at AWS was Elastic Network Adapter (ENA). Widely supported across Elastic Compute Cloud (EC2) instance families, this offered high throughput network capabilities in the cloud. However it's latency offerings with the traditional Transmission Control Protocol/Internet Protocol (TCP/IP) stack were not ideal for tightly coupled data intensive workloads. In 2019 AWS announced a new network fabric EFA supported by a cloud native proprietary network protocol called Scalable Reliable Datagram (SRD). EFA aims to provide more consistent lower latencies and higher through-put than it's TCP/IP counterpart. SRD is a cloud native network protocol developed by AWS for EFA. It is an Ethernet based software protocol optimized to run within the AWS cloud.

EFA is exposed to applications through a set of low-level kernel driver with minimal capabilities in combination with user-space libraries such as *ibverbs* and *libfabric*. *ibverbs* is an implementation of the RDMA verbs defined by the InfiniBand specifications. It is commonly exposed to the user-space as a system library named *libibverbs*. OpenFabrics Interfaces is a framework aimed at exposing network fabric communication for high- performance computing applications. It provides a standard set of APIs that are agnostic to the underlying network protocol implementations. *libfabric* is a core component of the OpenFabrics interface suite. Native capabilities supported by the EFA hardware are directly exposed by the low level interfaces, however features not native to the fabric are implemented in software by *libfabric*. These application programming interfaces (APIs) operate at different levels in the software stack, which incidentally comes with some trade-offs coupled with an impact on performance.

EFA has primarily been marketed towards HPC workloads by AWS. Research addressing EFA has also primarily been driven by the HPC community [8, 42]. Most of the focus has been around Message Passing Interface (MPI), which is a popular framework used for HPC workloads. MPI also internally uses *libfabric* as the interface for EFA. This does not directly correlate with the requirements of a data-intensive system such as a Database Management System (DBMS) [28].

1.2. Problem Statement

Current trends relevent to data-intensive systems can be broadly outlined as follows. A wide scale adoption of cloud computing services can be observed. Cloud vendors have started providing high-performance network fabrics aimed at tightly coupled workloads. This translates to more data-intensive systems operating over high-speed, low-latency cloud native network fabrics like EFA. Rapid growth of these network fabrics impacts the design and architecture of data-intensive systems. We identify a gap in the research aimed at systematically evaluating EFA by the data management community. This could be a consequence of EFA being marketed towards HPC and MPI workloads. Additionally there are no official hardware specifications provided by AWS from which performance metrics could be extrapolated. Moreover applications can utilize the EFA fabric using interfaces at various levels, some being very close to the hardware. An in-depth evaluation of EFA comparing it with existing research in low-latency fabrics is needed to better understand it's implications on system design.

Therefore, the following question remains:

Can EFA be a viable alternative to RDMA over InfiniBand for data-intensive systems?

1.3. Goals and Contributions

As stated above, we primarily aim to address the question, if EFA can be a viable alternative to RDMA over InfiniBand for data-intensive systems by performing an in-depth systematic evaluation. In this section, we outline our goals for such an evaluation.

Network fabrics offered in the cloud. We initially plan to provide a brief overview of the current state of network fabrics offered in the cloud. In addition, we also plan to outline technologies such as *SR-IOV* and *User-space networking* that enable cloud vendors to provide high performance network fabrics.

Describing the EFA fabric. We plan to explore the EFA fabric in detail. Describing it along with the SRD protocol and highlight it's functional differences compared to RDMA and TCP/IP. These network fabrics have some fundamental differences in terms of hardware capabilities, protocol implementations, programming interfaces, networking models etc. Highlighting these differences becomes relevent before an in-depth performance evaluation.

Performance of EFA in comparison with RDMA. We next focus on evaluating the performance of EFA by comparing it with RDMA on various parameters such as Latency, Bandwidth, Message rate using reproducible micro-benchmarks. Including an in-depth evaluation strategy for both EFA and RDMA. We plan to provide comprehensive implementation details for any micro-benchmark suits that we develop. This is the core contribution of the thesis, which will directly address the problem at hand.

Evaluating the programming interfaces for EFA. As pointed out in the introduction, EFA has various programming interfaces at different levels in the software hierarchy. This includes the comparision of EFA's low level interface *ibverbs* with EFA's higher level interfaces such as *libfabric* and *Open-MPI*. We plan to explore these and determine any overheads or trade-offs in using higher level interfaces.

Implications for system design. Finally based on their performance characteristics, we plan to highlight the implications for system design. These characteristics would naturally have major implications for data-intensive systems on the cloud.

2. Background

In this section, we provide a brief overview of the key technologies and concepts used in this thesis. Firstly we describe the current state of high performance networking in the cloud. Followed by a brief description of the AWS cloud. Then we explore EFA's capabilities and features in detail. Finally we compare it's functionality with RDMA and socket interfaces.

2.1. High-Performance Networks in the Cloud

Virtualization is almost synonymous with cloud computing. Network adapters in virtualized environments have been extensively studied and their performance characteristics do not resemble that of real hardware [11]. This directly extends to the cloud where physical hardware is exposed to the user through various hypervisors. Studies have shown that traditional TCP/IP performance between virtual machines on the cloud shows significant variance and degradation [43]. Advancements in technologies like SR-IOV has enabled cloud providers to greatly improve network performance. SR-IOV is a virtualization specification that allows isolation of PCI Express peripherals. This allows High performance network hardware to essentially circumvent the hypervisor in the network data-path.

AWS markets their high-performance networking capabilities as *Enhanced networking*. This feature can be enabled on virtually all current generation instance families using one of two mechanisms. First is ENA, a network fabric offering that supports [100 Gbps] networking which can be enabled on instances with the *ena* kernel module. AWS provides a set of proprietary drivers to interface with this fabric. Second is through the Intel 82599 VF interface, which provides [10 Gbps] networking. This is supported by the *ixgbevf* Network Interface Controller (NIC) Virtual Function drivers [3]. More recently AWS introduced EFA, a network adapter aimed at providing more consistent latencies and higher throughputs compared to it's previous offerings. (Figure 2.1) briefly highlights the differences between EFA and ENA, also showcasing OS-Bypass architecture. EFA will be explored in greater detail throughout this thesis.

Meanwhile, Azure markets their High-performance networking stack as *Accelerated Networking*. Accelerated Networking is supported by most of Azures general purpose and compute optimized instance families. Enabling accelerated networking, in essence creates a Virtual Function within the physical NIC which shows up as a dedicated network adapter within the guest instance. This direct access to the hardware enables consistent lower latencies. (Figure 2.2) briefly highlights Accelerated Networking architecture. At their core, azure hosts are built on various models of Mellanox physical NICs supported mainly by *mlx4* or *mlx5* drivers. In addition to Accelerated Networking, Azure is the only major cloud provider that offers InfiniBand adapters in the cloud. However it is supported by only a few HPC instance families [4] (H, HB, HC and some of the N series). This negates the flexibility and benefits that cloud computing promises.

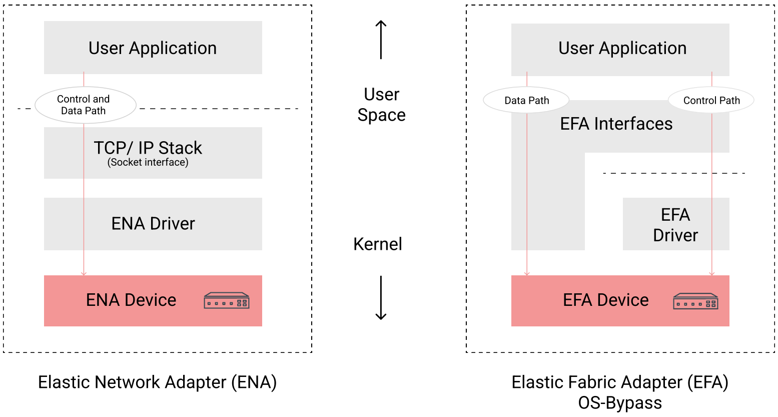


Figure 2.1.: ENA vs EFA architecture (OS-Bypass)

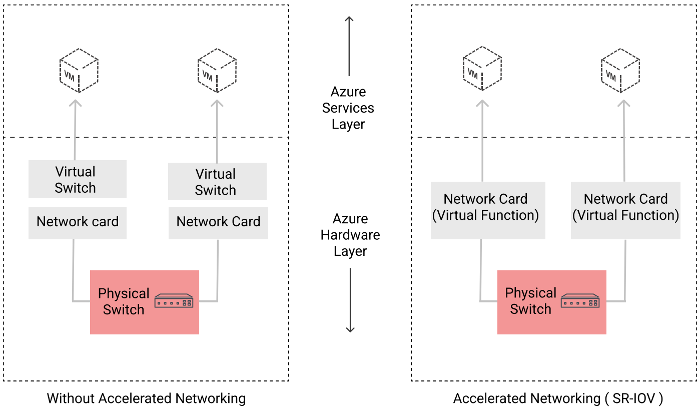


Figure 2.2.: Azure Accelerated Networking (SR-IOV)

Finally of the big three cloud providers GCP takes a slightly different approach to High-Performance networking in the cloud. GCP is primarily built on Jupiter network fabrics paired with a proprietary Software Defined Networking platform called Andromeda. Andromeda's architecture enables OS-Bypass using Intel's direct memory access engine and IOMMU [9] as opposed to SR-IOV.

2.2. User-space Networking

Traditional Kernel network stacks have evolved over the years to support a wide range of hardware and perform diverse functions. Traditional socket Application Programming Interface (API)s expose a stable interface to support the TCP/IP communication model. This has come at a high performance penalty. Moreover changes to the kernel's network sub-system are relatively cumbersome and slow. These factors have motivated specialized networking stacks such as Data Plane Development Kit (DPDK), OpenFabrics Interfaces (OFI), etc to operate in the user-space. In this section we describe some of the key features supported by these network stacks aimed at high-performance fabrics.

2.2.1. Direct Hardware access

To approach sub micro-second latencies, software stacks must be as lean as possible. This involves applications accessing the hardware command queues and structures directly, with minimal kernel involvement.

Kernel Bypass / OS-Bypass. Kernel bypass as the name indicates is the capability of an application to avoid calling the kernel sub-systems for data transfer operations. Complete independence from the kernel is impractical due to security considerations, hence data transfer operations also known as *fast-path* avoid the kernel. But certain control operations still utilize the kernel modules.

Direct Data Placement. When the application has direct access to NIC hardware, direct data placement avoids processing overhead and memory copies. Advance fabrics are even capable of remote data placement without the involvement of the peer processor, providing the basis for one-sided operations like RDMA.

2.2.2. Limit Memory Copy

Memory copies are expensive operations. Traditional networking stacks usually copy user data at both the sender and receiver side. This provides a simple programming interface, but at a performance penalty.

Network Buffers. Send/Receive buffers can either be owned by the application and registered with the NIC or the NICs buffers can directly be access by the application. Based on the complexity of the network protocol, an appropriate model is chosen.

Fabric Resource Management. Allocating resources on the NIC to avoid overrunning data buffers or queues is usually performed the kernel stack. But since our main focus is to avoid kernel calls, the specialized networking framework is responsible for all this resource management. This also involves implementing complex flow-control and congestion-control mechanisms based on the protocol.

2.2.3. Asynchronous model

Traditional sockets also offer an asynchronous mode of operation, but this involves extensive memory copy overheads, making the approach infeasible. For a true asynchronous model, the application must be able to submit an operation and later receive a confirmation regarding the status of this operation. All this without any additional memory copies or buffer operations.

Event Queues. To support asynchronous operations, events which correlate to a request are usually written to a queue by the NIC and the tail pointer is simply updated to indicate that a request is done. This simple inter-process communication model improves CPU cache, code locality without the overhead of complex interrupt/signal handling.

2.3. Amazon Web Services

EFA is a network fabric on the AWS cloud platform. Consequently, a few key concepts in AWS closely influence our evaluation setup and methodology. In this section we briefly describe some of them.

2.3.1. Elastic Compute Cloud - EC2

Although cloud providers offer a variety of services to end users, three key services namely compute, storage and networking form the basis for all other offerings. EC2 is the compute service in the AWS cloud. Based on their hardware capabilities, virtual machines termed as *instances* are grouped in different categories such as General purpose, Storage optimized, Compute optimized, etc. EFA can be paired with a majority of these instance types to provide high performance networking.

2.3.2. Virtual Private Cloud - VPC

Virtual private cloud (VPC) is an AWS service that enables logical grouping and isolation of other AWS services. These are basically virtually defined network topologies within amazon's data-centers. VPCs are used to define how resources inter-communicate within AWS regions and availability zones. [41]

2.3.3. Security Groups

A Security group is essentially a virtual firewall within the AWS cloud. Each VPC is pre-configured with a default security group. Rules within a security group are used to allow traffic to a instance [40]. EFA enabled instances have certain limitations regarding inbound and outbound traffic rules relating to security groups [38].

2.3.4. Placement Strategy

When a new compute instance is launched, it's placement within the AWS cloud has an influence on various parameters such as availability, failure tolerance, network latency, etc. This is controlled by an attribute called *Placement strategy*. AWS offers the following three strategy classes[39], each aimed at different use cases.

- **Cluster -** Instances are placed as close as possible. Recommended for applications that benefit from low network latency.
- **Partition -** Instances are divided and placed on logical segments called partitions. AWS ensures that each partition within a placement group has its own set of racks.
- **Spread -** Instances are each placed on distinct racks, with each rack having its own network and power source. Possibly across different Availability Zones.

2.4. Elastic Fabric Adapter

Until recently the main network offering by AWS for High-performance workloads was ENA. However for data-intensive and tightly coupled workloads ENA along with the traditional TCP/IP stack did not perform well [35]. In response to this AWS launched EFA a high-throughput low-latency network fabric on it's cloud platform. (Figure 2.3) describes the hardware and software stack of EFA. Starting with the hardware nitro card which is a PCIe attached network adapter powering the entire EFA stack. A cloud centric protocol called SRD forms the foundation of EFA's software stack. Traditional kernel network stacks prove to be inadequate when handling such high throughput devices. Kernel bypass architecture coupled with asynchronous network interfaces provides an excellent alternative to alleviate such bottlenecks by moving the packet processing to user-space. In the case of EFA, this is achieved using a low-level kernel driver with minimal capabilities in combination with user-space libraries such as *ibverbs* and *libfabric*.

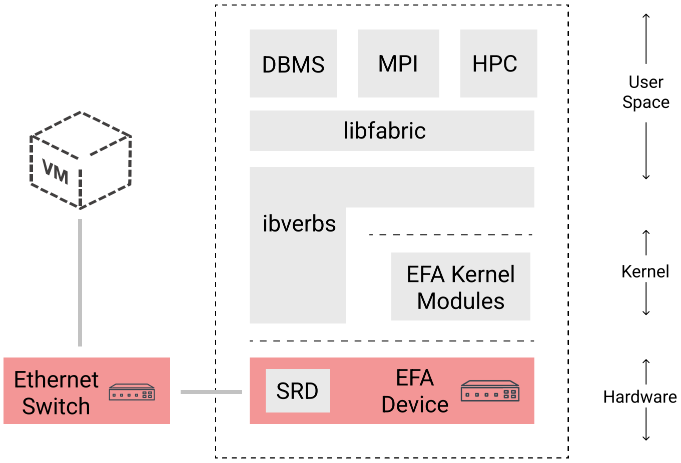


Figure 2.3.: EFA Hardware and Software Stack

2.4.1. SRD Protocol and Driver Modules

SRD: A Reliable out-of-order protocol. The foundation of EFA's software stack is SRD, a proprietary network protocol that provides reliable but unordered communication on top of commodity Ethernet switches [42]. However, unlike existing Ethernet protocols such as TCP/IP, SRD is purpose-built for Amazon's data centers and only offers out-of-order delivery. There are two main reasons for this decision: (1) Only few applications require in-order delivery and thus the application layer should handle in-order delivery if needed. (2) Out-of-order delivery reduces tail latencies. This is because in-order delivery may cause head-of-line blocking [42]. Additionally, to avoid hot paths in the network and thus reduce the chance of packet drops, SRD packets are sent across multiple network paths. Thus, out-of-order delivery is a direct consequence of sending packets across multiple paths, and enforcing the order would require large intermediate buffers or to drop out-of-order messages. Amazon controls the hardware SRD operates on, which in turn enables them to implement SRD's reliability in the AWS Nitro network cards (rather than in software).

EFA Kernel Driver. The EFA low-level driver interface can largely be classified into management and data-path interfaces. Management commands are submitted and monitored using a set of queues. Admin Queue (AQ), Admin completion queue (ACQ), and Asynchronous Event Notification Queue (AENQ). The data-path interface supports I/O operations using similar queue pairs namely, Send queues (SQ) and Receive queues each with an associated completion queue. [34]

2.4.2. ibverbs - EFA's Low Level Interface

SRD is exposed to the application via the *ibverbs* library. Ibverbs is the same library that allows user-space processes to use RDMA primitives to perform high-throughput, low-latency network operations on InfiniBand. The fact that EFA and RDMA over InfiniBand share the same low-level library is no coincidence, because EFA closely resembles the InfiniBand verbs specification [42]. However, there are important distinctions between EFA and reliable RDMA over InfiniBand. Reliable connected RDMA provides two types of operations: memory and messaging operations. Memory operations include one-sided *read/write/atomic* primitives and messaging operations the two-sided primitives *send/recv*. EFA does not support one-sided *read/write/atomic* primitives [42], but is limited to two-sided primitives. This means that RDMA designs that heavily rely on one-sided operations have to be reconsidered when porting them to EFA.

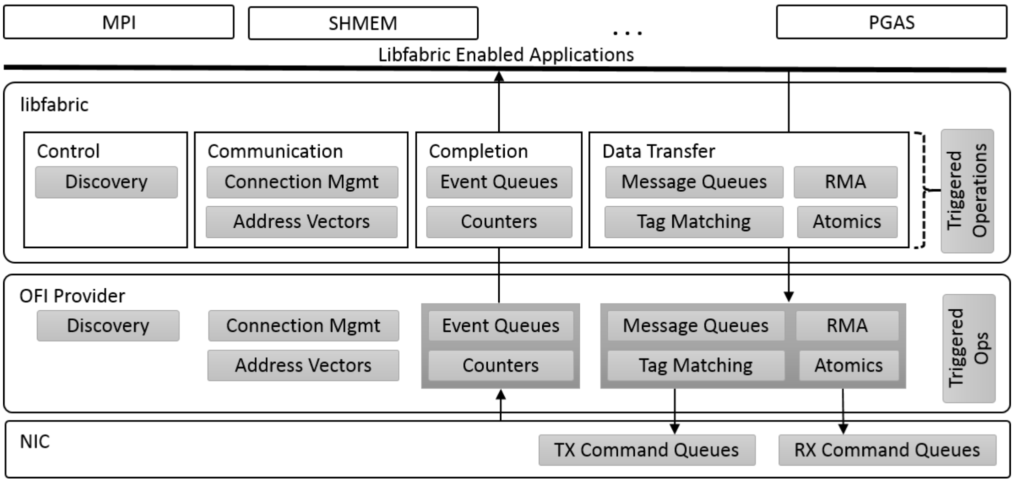


Figure 2.4.: Architecture of *libfabric* (OFI working group)[22]

2.4.3. OpenFabrics *libfabric* - EFA's High Level Interface

OpenFabrics Interfaces is a framework aimed at exposing network fabric communication for high-performance computing applications. It provides a standard set of APIs that are agnostic to the underlying network protocol implementations. (Figure 2.4) courtesy of the OpenFabrics Interfaces Working Group, provides an excellent overview of the architecture of *libfabric*. The framework can broadly be classified into control, communication, completion, and data-transfer services. Control services are used by the application to discover provider-specific features and capabilities. Communication services assist in address management, connection setup, and teardown between nodes. Since most of the interfaces are asynchronous, completion services provide queueing interfaces to monitor submitted operations. Data transfer services designed around core communication paradigms expose interfaces to trigger data transfer operations.[22, 23]

Provider. Each type of hardware device termed as a provider, hooks into the framework and implements optimized device-specific functionality. EFA is essentially one such provider under *libfabric*.

Endpoint type. Internally, communication models supported by the providers are exposed as endpoint types. In case of EFA primitives provided by *ibverbs* and some higher level functionalities are offered to the application developer. Two types of communication models are supported by EFA. An Unreliable Datagram (DGRAM) protocol (FI_EP_DGRAM), with a maximum datagram size equalling the Maximum Transmission Unit (MTU) of the underlying layer. A Reliable datagram message (RDM) protocol (FI_EP_RDM) that provides features such as error handling, segmentation, and reassembly with no limitation on message size.

Operating mode. Operating modes are used to specify optimal ways of accessing the provider. EFA requires the applications to use (FI_MSG_PREFIX) which essentially indicates that the application allocates space in the message buffer for the providers use. This is utilized by the provider to implement the EFA protocol header.

Fabric capabilities. Given the large amount of hardware adapters supported by *libfabric*, capability flags are a means to request and filter out specific hardware/software capabilities supported by a given provider. They are grouped into three broad categories.

- Primary capabilities: Capabilities that must be explicitly requested by the application
- Primary modifiers: Modifiers applied to primary capabilities. If not specified, all relevant capabilities for the provider are applied.
- Secondary capabilities: These are optional capabilities that may be requested by the application to limit endpoint functionality for security or performance reasons.

Appendix B lists all the capabilities supported by the RDM endpoint.

Addressing Formats. Various fabric interfaces take the destination or source address as a parameter. EFA uses a proprietary address format called (FI_ADDR_EFA) implemented within *libfabric*.

2.4.4. libfabric - RDM Communication Protocol v4

Since a substantial portion of this thesis focuses on reliable connected communication, it becomes essential to highlight the EFA RDM Communication Protocol. At it's core the EFA device is only capable of reliable out-of-order packet delivery up to the device's MTU size as specified by the underlying SRD protocol. To extend this functionality to include other communication patterns and hardware unsupported features, *libfabric* implements the RDM protocol for EFA [14]. The following features are implemented/emulated:

- Connected communications and transfer of messages exceeding MTU size.
- Atomic operations up-to MTU size supporting *atomic* primitives.
- Remote Memory Access (RMA) functionality supporting *read/write* primitives.

Table 2.1.: SRD compared to reliable connected RDMA and traditional TCP/IP sockets.

SRD (EFA)	RC RDMA (IB)	Sockets (TCP/IP)
Ethernet	InfiniBand	Ethernet
reliable	reliable	reliable
Messages	Messages	Stream
unordered	ordered	ordered
user-space	user-space	kernel-space
asynchronous	asynchronous	synchronous
no one-sided	one-sided	no one-sided

2.5. RDMA and Sockets

RDMA. Direct Memory Access (DMA) is a ubiquitous technology in current computing systems that basically allows hardware subsystems to access main-memory independent of the central processing unit. RDMA extends this functionality over multiple network interconnected nodes. Applications that require high-throughput low-latency communication with minimal CPU footprint have widely adopted RDMA.

Sockets. Traditional TCP/IP Sockets are the most widely used networking API, virtually supported by every operating system. It has been the de-facto network stack for many years. However it's synchronous nature and the need to support a wide variety of network hardware, negatively impacts performance.

Table 2.1 compares and summarizes the key differences between reliable EFA, reliable connected RDMA over InfiniBand, and traditional TCP/IP sockets. All three network approaches shown in the table are reliable. As a consequence of their similar design objectives (low latency and high bandwidth) we can observe that EFA and RDMA share many common characteristics. Both are designed to use message based semantics unlike TCP/IP Socket's streaming based protocol in which message boundaries are opaque to the transport layer. While SRD provides unordered packet delivery its message based semantics helps to handle out-of-order packets in the application level if needed. That would be infeasible with a byte streaming protocol as used by TCP/IP. RDMA's unique feature is the native one-sided support. It is important to note that there are other combinations which we have not discussed or shown in the table. For instance, RDMA over Converged Ethernet (RoCE) uses Ethernet as its fabric instead of InfiniBand.

3. Implementation

In this section we provide an overview of our benchmark implementation. Starting with pre-existing benchmark projects used in our evaluation methodology such as InfiniBand Verbs Performance Tests (*perftest*) and OSU Micro-Benchmarks (*OMB*). We then describe our novel micro-benchmark suite *efa-bench*.

3.1. Overview of Existing Projects

3.1.1. InfiniBand Verbs Performance Tests (*perftest*)

To isolate the fundamental properties of EFA and RDMA over InfiniBand we used the well-known performance micro-benchmark library *perftest* (available at [2]). Perftest uses *ibverbs* directly and supports EFA (SRD and UD) as well as reliable connected RDMA over InfiniBand. This makes the experiments directly comparable as both implementations use the same benchmarking code.

Perftest implementation details[2]:

- Latency benchmark measurements are performed on the same node, meaning only round-trip time is measured. This would be an issue on architectures with asymmetric network characteristics.
- Perftest recommends small sampling periods to derive accurate device performance. Large number of iterations leads to performance degradation. This is due to higher memory footprint of internal benchmark data-structures unrelated to the network adapter.

3.1.2. OSU Micro-Benchmarks (*OMB*)

As mentioned previously, EFA has been primarily marketed towards the HPC community which extensively relies on MPI. MPI is a open standard for message passing between large number of nodes within a parallel computing architecture. Most popular implementations of the MPI standard, Open-MPI and Intel-MPI are both built on top of the *libfabric* framework. The MVAPICH project provides a popular MPI micro-benchmark suite *OMB* (available at [45]).

We evaluate our micro-benchmark *efa-bench* built on top of *libfabric* against *OMB*. This provides us a good point of reference for *libfabric* based applications.

OSU Micro-Benchmark implementation details[45]:

- The latency tests *osu_latency* are carried out in a ping-pong fashion and average one-way latency is reported. Blocking variants of the MPI library functions (*MPI_Send* and *MPI_Recv*) are used.
- The multi-pair bandwidth message rate tests *osu_mbw_mr* send a fixed number of messages within a given window-size. This is repeated over multiple iterations to calculate the achieved bandwidth and message rate. Asynchronous variants of the MPI library functions (*MPI_Isend* and *MPI_Irecv*) are used.

3.2. Overview of *libefa* and *efa-bench*

3.2.1. *libefa* - A Wrapper Around *libfabric* for EFA

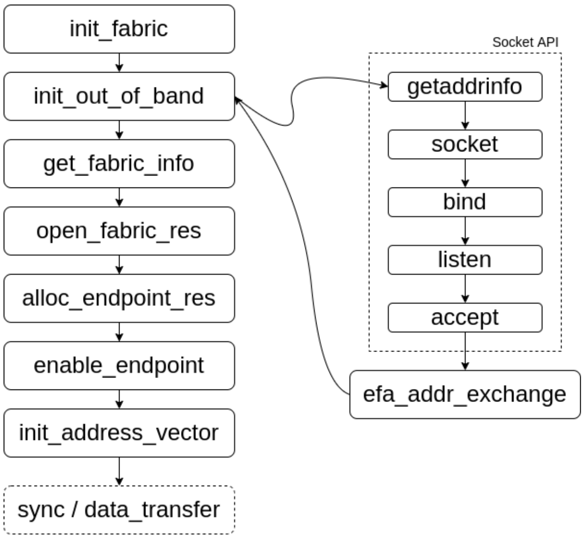


Figure 3.1.: EFA Communication Setup with *libfabric*

Building applications with *libfabric* involves a fair bit of complexity when compared to the traditional Sockets API. In order to abstract out the intricacies of libfabric services and to focus on the EFA provider, we implement a library *libefa*. We also drew inspiration from *fabtests*[15], a set of generic examples on how to use libfabric.

As described in section 2.4.3, the EFA provider supports two types of models *FI_EP_DGRAM* and *FI_EP_RDM*. Both of these are connection-less endpoints that allow data transfer without any connection establishment between the two nodes. However an important distinction compared to traditional sockets is that, even for connection-less endpoints *libfabric* needs the peer address to be inserted into the local addressing table before actual data transfer[22]. This is usually done during the application initialization so that later data-transfer operations can seamlessly use the pre-loaded address vectors.

From a high-level application's perspective, communication over EFA using *libfabric* can be categorized into two main phases. A communication-setup phase, where all *libfabric* resources are allocated and configured. A data-transfer phase where actual user data is exchanged. *libefa* provides an abstraction of these services by providing a *Node* object, that can be configured to function either as a *Server Node* or a *Client Node*. Majority of the steps involved in the connection setup phase are independent of Node type. Figure 3.1 provides the flow of a typical *libfabric* application during communication-setup.

init_fabric. Involves the initialization of various context resources maintained by the application like transmit and receive counters, sequence numbers, completion counters, file descriptors etc.

init_out_of_band. In order to obtain the peer node's EFA address required for the address vector, out-of-band communication using traditional sockets is utilized. This involves the typical bind, listen, accept phases on the server, while the client connects to this OOB port. Once a communication channel has been established, during the *efa_addr_echanged* phase EFA address is exchanged by both the nodes.

get_fabric_info. The main operation during this phase revolves around the *fi_getinfo* API. The application specifies a set of criteria based on which fabric information is retrieved. This is used to query and tune the fabric using various primary and secondary capabilities.

open_fabric_res. Various resources such as transmit/receive buffers, completion queues are allocated and these user-space memory regions are registered with the fabric through APIs like *fi_mr_reg*. For RMA operations, this process is also used to associate a secure key with the memory region used by the remote peer's *read* and *write* primitives.

enable_endpoint. The previously allocated resources like completion queues are associated with an endpoint using APIs like *fi_ep_bind*. The endpoints are also configured using flags to indicate the capabilities they must support such as *FI_SEND*, *FI_WRITE*, *FI_REMOTE_WRITE*.

init_address_vector. As mentioned, to facilitate connection-less endpoints the exchanged EFA address is inserted into the local address table using APIs like *fi_av_insert*.

sync / data_transfer. Once all the initialization phases are complete, the peers reach a synchronization point and exchange their status. This *sync* can either be performed out-of-band using the previously opened socket connection or through the EFA channel itself. At this point the communication-setup is complete and the nodes are ready for data-transfer operations.

```

/*
* User initializes a Server/Client node object by providing a set of parameters like,
* libfabric provider type, endpoint type, ports on which the server should listen, etc.
* Depending on the experiment, specific fabric hints may be specified
*/
libefa::Node::Node(std::string provider, std::string endpoint, std::string port, std::
    string oobPort, fi_info *userHints)
{
    // Initializes a context unique for this connection
    struct ConnectionContext ctx;
    init_connection_context(&ctx);
    ...
    // Initialize user specified options for this connection context
    init_options(&ctx.opts);
    ...
    // Performs all the Communication Setup phases as described above
    init_fabric(&ctx);
}
```

Listing 3.1: *libefa* Node setup snippet

3.2.2. *efa-bench* - A Benchmarking Suite for EFA

In order to perform a comprehensive evaluation of EFA we implemented various micro-benchmarks. Network parameters like latency, throughput, message rate, NIC parallelism were explored in-depth.

Figure 3.2 provides an architectural overview of the *efa-bench* suite. *efa-bench* is built on top of the previously described *libefa* wrapper, which in-turn uses *libfabric*. At a high level, there are modules that cover two-sided primitives like *send/receive*. RMA modules for one-sided primitives like *read/write*. To ensure accurate performance measurements, the network hardware counters are monitored for transmited/received packets, bytes etc.

(Appendix A) provides a detailed build and usage guide, with all supported *efa-bench* flags. Additionally, details about performance measurements and tuning for the micro-benchmarks can be found in (Appendix D).

In the next sections we briefly describe the implementation details of various micro-benchmarks in *efa-bench*. Based on the benchmark type, the fabric is appropriately configured using *fi_info* flags and capabilities.

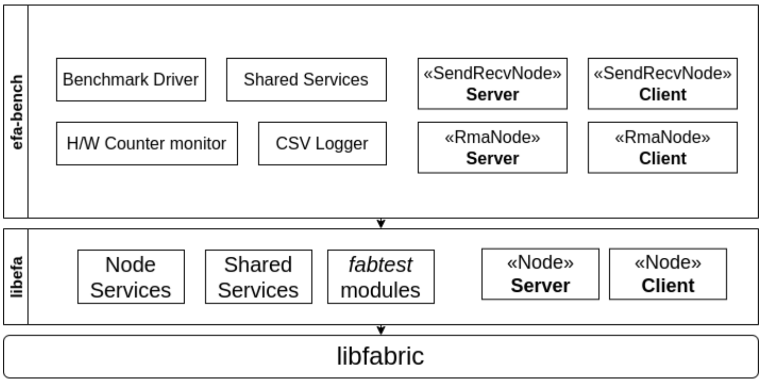


Figure 3.2.: Architecture of the *efa-bench* Suite

3.2.3. Latency Micro-benchmark

Latency benchmarks are carried out in a ping-pong fashion similar to *perftest* and *OMB*. Two-way latency is measured on one of the nodes and the calculated average one-way latency is reported. Snippets 3.2 and 3.3 briefly describes the ping-pong pattern used in the server and client respectively. Server performs a TX followed by an RX operation, while the client performs a reverse of this. The TX/RX operations represent blocking calls, essentially each asynchronous *fi_sendmsg/fi_recvmsg* API call is immediately followed by a completion request.

```

void pingPong()
{
    // Initialize and configure a server node
    Server server = Server();
    ...
    while (true) {
        // Post a transmit request and wait for it's completion event
        server.tx();
        ...
        // Post a receive request and wait for it's completion event
        server.rx();
    }
    ...
}
```

Listing 3.2: Latency micro-benchmark server snippet

```

void pingPong()
{
    // Initialize and configure a client node
    Client client = Client();
    ...
    while (true) {
        // Post a corresponding receive request and wait for it's completion event
        client.rx();
        ...
        // Post a transmit request and wait for it's completion event
        client.tx();
    }
    ...
}
```

Listing 3.3: Latency Micro-benchmark Client Snippet

3.2.4. Bandwidth Micro-benchmark

In order to measure maximum throughput, the primary goal is to keep the transmission queues filled with requests and ensure the adapter is active throughout the experiment. We use a simple data transfer pattern analogous to the sliding window mechanism shown in snippet 3.4. Transmission requests are submitted to the fabric until we hit a predefined window-size. At which point, we wait for completion events from the fabric.

In case the fabric lacks resources to handle additional transmission requests, *libfabric* defines a special error mechanism *FI_EAGAIN*[18]. These exceptions have been handled appropriately by draining the completion queue.

```

void batch()
{
    // Initialize and configure a server node
    Server server = Server();
    ...
    while (true) {
        /*
        * Keep posting transmit requests until we hit window size,
        * performed using asynchronous non-blocking APIs
        */
        if (outstandingOps < windowSize) {
            server.postTx();
            outstandingOps++;
        }
        ...
        /*
        * Once the number of outstanding operations reach window size,
        * Read the completion queue for N completion events
        */
        if (outstandingOps == windowSize) {
            server.fiCqRead(N);
            outstandingOps -= N;
        }
    }
    ...
}
```

Listing 3.4: Bandwidth Micro-benchmark Server Snippet

As for the client [3.5], it continuously posts RX requests and processes the completions as and when the receive buffers get filled.

```

void batch()
{
    // Initialize and configure a client node
    Client client = Client();
    ...
    while (true) {
        /*
        * Keep posting receive requests to accept all Server transmissions.
        */
        client.postRx();
        client.fiCqRead();
    }
    ...
}
```

Listing 3.5: Bandwidth Micro-benchmark Client Snippet

3.2.5. Message API Variants Micro-benchmark

Inject messages. *libfabric* provides an optimized version of the *send* primitive called *fi_inject*. For messages transmitted using inject, the transmission buffers are immediately available for reuse and completion events are not generated [18]. However the request message size is limited by *INJECT_SIZE* which in case of *efa* is 3928 bytes.

Tagged messages. Messages which carry a key or tag with the message buffer. The tag is used at the receiving endpoint to connect an incoming message with a corresponding buffer. Based on the tags associated with the send and receive buffers, the posted receive buffers are matched with inbound send messages. [19]

We implement tagged and inject benchmarks similar to the latency micro-benchmarks to measure their performance. The client side has no change here since this is purely a *send* primitive optimization.

3.2.6. Saturated Bandwidth Micro-benchmark

To evaluate the impact of link saturation on various network parameters, we implement a multi-threaded micro-benchmark. As described in snippet 3.6 a set of traffic generator threads are started prior to the benchmark to saturate the link bandwidth. The degree of saturation is controlled by a *SATURATION_WAIT* factor that determines how frequent each thread is allowed to submit transmission requests. The traffic generator threads use a busy wait mechanism as opposed to sleep which ensures minimal kernel scheduler impact. After the traffic generators have warmed-up, another worker is spawned to perform the actual micro-benchmark.

```

void trafficGenerator()
{
    // Initialize and configure a server node
    Server server = Server();
    ...
    while (true) {
        // Keep posting transmit requests to saturate the link using the batch paradigm
        server.tx();
        ...
        // Wait for SATURATION_WAIT microseconds before posting the next batch
        server.busyWait(SATURATION_WAIT);
    }
}

void saturationDriver()
{
    // Spawn N trafficGenerator threads to saturate the link
    for (i = 0; i < N; i++)
        std::thread(trafficGenerator)
    ...
    // Spawn a worker thread to run the latency benchmark
    std::thread(worker)
}
```

Listing 3.6: Saturated Bandwidth Server Micro-benchmark

The *saturationDriver* client code is very similar to the server. It spawns the same number of threads as the server to accept all the incoming messages [Snippet 3.7]. This ensures link saturation. After which another client worker thread runs the selected benchmark.

```

void trafficGeneratorClient()
{
    // Initialize and configure a client node
    Client client = Client();
    ...
    while (true) {
        /*
        * Keep posting receive requests to accept all Server transmissions.
        */
        client.rx();
        ...
    }
}
```

Listing 3.7: Saturated Bandwidth Client Micro-benchmark

3.2.7. RMA Micro-benchmark

For evaluating the performance of one-sided primitives emulated by *libfabric*, we implement RMA micro-benchmarks using *fi_read* and *fi_write* APIs. The fabric is appropriately configured for remote operations, virtual addresses and memory registration keys are exchanged. After which the data patterns are fairly similar to the batch micro-benchmarks.

```
void rma()
{
    // Initialize and configure a server node
    Server server = Server();
    // Exchange the keys used during memory registration
    server.exchangeKeys();
    ...
    while (true) {
        /*
        * Post an RMA request at the specified remote memory address,
        * The remote address along with the key is specified
        * through an fi_rma_iov structure
        */
        rma_iov.addr = REMOTE_ADDR;
        rma_iov.key = REMOTE_KEY;
        server.rma(rma_iov, WRITE);
    }
    ...
    // Sync with the client to indicate the end of one-sided operations
    server.sync();
}
```

Listing 3.8: RMA Micro-benchmark Server Snippet

As mentioned previously, since one-sided operations are emulated in *libfabric* and not natively supported, we need to ensure the client's progress engine is running throughout the RMA benchmarks.

```
void rma()
{
    /*
    * Initialize and configure a client node,
    * Receive buffers are registered with a key that the server needs later.
    */
    Client client = Client();

    // Exchange the keys used during memory registration
    client.exchangeKeys();
    ...
    /*
    * Wait for the server to complete it's one-sided operations,
    * This is needed to ensure the connection stays alive
    * and libfabric's internal progress engine handles the RMA requests.
    */
    client.sync();
}
```

Listing 3.9: RMA Micro-benchmark Client Snippet

3.2.8. Multi-threaded Micro-benchmark

To best evaluate NIC parallelism, we implement multi-threaded benchmarks that build on the single-threaded ones. As shown in snippet 3.10, the driver code spawns multiple worker threads, each allocated to a different port. Each of these workers create a fabric endpoint with an independent context. Although under the hood, certain *libfabric* resources may be shared, for instance the address vector table.

The client communication pattern is again very similar to the server. To match each server worker, a client worker is spawned and establishes connection with the appropriate port. After which, the client worker posts receive operations.

```
void serverWorker(PORT, OOB_PORT)
{
    // Initialize and configure a server node at the specified port
    Server server = Server(PORT, OOB_PORT);
    ...
    while (true) {
        ...
    }
}

void clientWorker(PORT, OOB_PORT)
{
    // Initialize and configure a client node that connects to the specified port
    Client client = Client(PORT, OOB_PORT);
    ...
    while (true) {
        ...
    }
}

void multiThreadDriver()
{
    /*
    * Spawn N worker threads to run the benchmark on the specified port.
    * Each connection maintains an independent connection context.
    */
    for (i = 0; i < N; i++)
        std::thread(worker, PORT + i, OOB_PORT + i)
    ...
}
```

Listing 3.10: Multi-threaded Server Micro-benchmark Snippet

4. Evaluation

In this section, we first outline our setup and evaluation methodology for both EFA as well as RDMA. Then we discuss the results of our extensive evaluation, comparing EFA with RDMA over InfiniBand across various network parameters.

4.1. Setup and Methodology

4.1.1. Setup

EFA as mentioned in previous sections is a network fabric offered on AWS EC2 instances. And since this platform does not offer RDMA over InfiniBand capabilities, we use two different hardware platforms for our evaluation.

EFA Setup. Before we describe our test setup, a few key limitations of EFA must be highlighted [38]:

- The EFA enabled instance must be a member of a security group that allows all inbound and outbound traffic to and from the security group itself.
- EFA OS-bypass traffic is not routable. In other words, EFA traffic cannot be sent from one subnet to another. Normal IP traffic from the instance however remains routable.

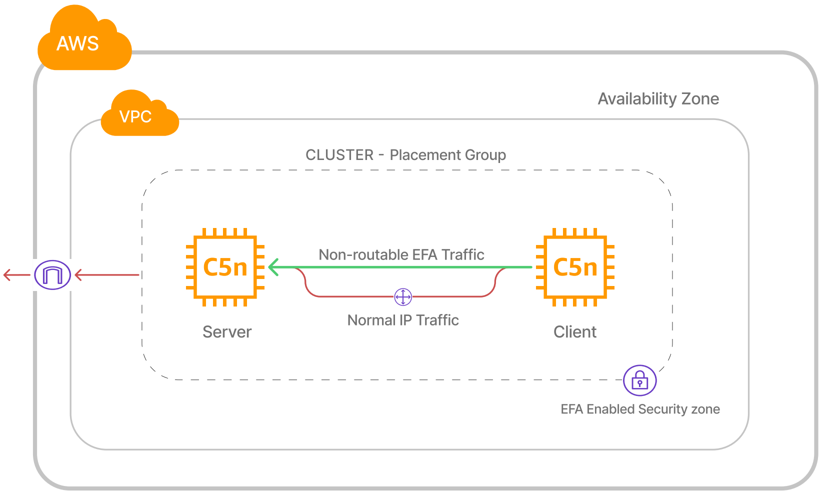


Figure 4.1.: Evaluation Setup

Figure 4.1 describes our EFA evaluation test bench on the AWS cloud. We first create a security group within an AWS Virtual Private Cloud (VPC) and configure it's rules to allow EFA traffic. Our test-bed consists of two *c5n.18xlarge* instances with 192 GB main memory and 72 vCPUs connected via 100 Gigabit EFA network adapters. These instances are also equipped with an ENA adapter that handles normal IP traffic. As for the software stack, we use an EFA supported Linux operating system *Amazon linux 2* running kernel version 5.10. We also replicated the setup using *c5n.metal* instances and obtained similar results.

Since our primary goal is to achieve the lowest possible latency with EFA, we explicitly specify a placement strategy before launching the test instances. We naturally launch our instances into a *Cluster* placement group (Explained in Section 2.3.4) to meet our latency goals.

RDMA Setup. The RDMA over InfiniBand experiments were conducted on two bare-metal machines with 1 TB main memory and 56 CPUs directly connected with an InfiniBand network using Mellanox ConnectX-5 MT27800 NICs (InfiniBand EDR 4x, 100 Gbps). A Linux based operating system running kernel version 5.4 was used.

4.1.2. Methodology

To isolate the fundamental properties of EFA and RDMA over InfiniBand we used the well-known performance micro-benchmark library perftest [2]. Perftest uses ibverbs directly and supports EFA (SRD and UD) as well as reliable connected RDMA over InfiniBand. This makes the experiments directly comparable as both implementations use the same benchmarking code. Additionally to extract other interesting EFA specific features, we use *efa-bench*. Also since perftest only supports single-threaded experiments, we use *efa-bench* for all multi-threaded evaluations. Lastly, to evaluate EFA's programming interfaces, we include *OSU-MPI*[45] as well.

To minimize data outliers, all benchmark data was collected three times and the average results were considered for evaluation. The benchmarks were run for sufficiently long to ensure stable reproducible results. Also to make certain there were no anomalies due to CPU scheduling and context switching, executions were pinned to specific cores using *taskset*. External software libraries like *libfabric* were built in *release* mode to ensure there were no performance penalties. For similar reasons, our micro-benchmark suite *libefa* and *efa-bench* were compiled using *gcc*'s *-O3* optimization flags. Raw data relevent to the upcoming evaluation sections can be found in (Appendix C).

4.2. Latency Evaluation

As latency is critical for many applications, we start our evaluation by comparing the latency of SRD (EFA) with RC-RDMA (IB). To classify the latency improvement of SRD (EFA) over EC2's traditional 100 Gigabit networking solution we included sockets (TCP/IP) as a reference. As mentioned, we use perftest [2] for EFA as well as RDMA and sockperf [30] for sockets.

4.2.1. RC-RDMA vs SRD vs TCP/IP

Figure 4.2 compares the effect of varying message sizes on the average latency, i.e., half-round-trip latency. Both EFA and RDMA provide lower latencies than sockets. However, when comparing RDMA with SRD, we can observe that RDMA's latency is around $20\times$ lower for messages below 512 bytes. For 8 kB messages RDMA's latency is still $10\times$ lower than SRD's latency. Thus, although the latency results for EFA are considerably better than for the TCP/IP sockets, there remains a substantial gap to RDMA over InfiniBand (similar latencies for RDMA are achievable on Azure VMs [31]).

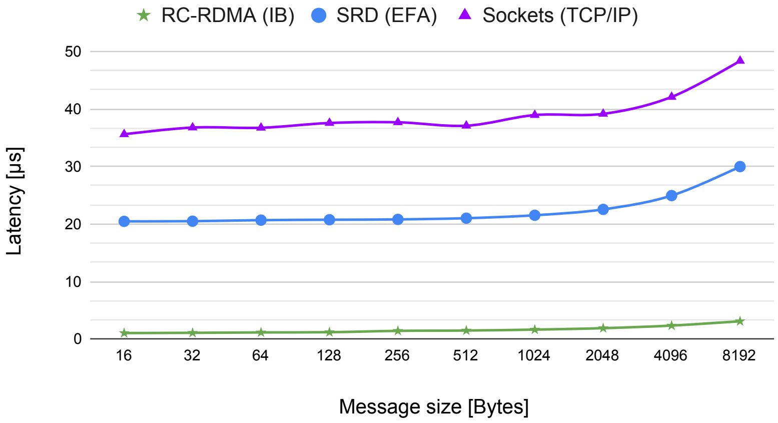


Figure 4.2.: Impact of Message Size on Average Latency - RC-RDMA vs SRD vs TCP/IP

4.2.2. Inline Optimization

To improve the latencies of small messages a well-known optimization in RDMA is to *inline* the payload in the message request. Without inlining, the NIC performs an additional step to fetch the payload of a message request using DMA over PCIe. When using inlining, the CPU directly copies the payload inside the message request and thus the NIC can avoid the additional step to fetch the payload. Consequently, the message can be transmitted faster, and thus latency decreases, as shown by Kalia et al. [25]. Essentially, inlining shifts work from the network card back to the CPU. Because inlining can be applied to SRD for messages up to 32 bytes, we investigate if this optimization is as beneficial as for RDMA (which supports inlining for messages up to 220 bytes). Figure 4.3 shows the relative latency improvement of inlining compared to a no inlining baseline. Although the absolute latency improvement of inlining is around 500 nanoseconds for both (EFA and RDMA), in relative terms the effect is only substantial for RDMA due to EFA's higher base latencies.

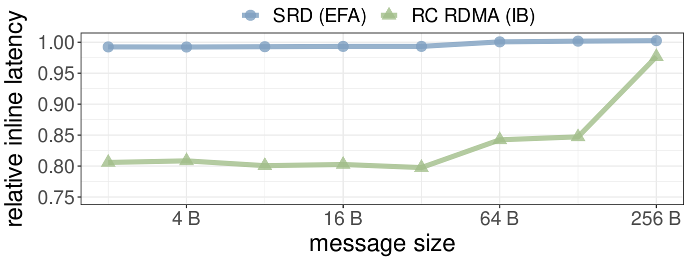


Figure 4.3.: Relative Impact of Inline Optimization on Latency

4.2.3. EFA's SRD vs UD Protocol

As the latency of SRD (EFA) is one order of magnitude higher compared to RDMA, we next evaluate if Unreliable Datagram (UD) (EFA) offers lower latencies. Figure 4.4 compares the effect of varying message sizes on latency on both connection types. Important to note is that SRD has an MTU of 8 kB while UD supports only 4 kB. From Figure 4.4 we can see that SRD (EFA) consistently has a slightly higher latency of around 1.5 μs. These are minor differences – especially when considering that SRD provides reliability in contrast to UD. One argument made in the past to favor UD RDMA over RC RDMA was its superior scalability [25, 26]. However, this does not translate to EFA's SRD protocol as described in Section 2.4.1.

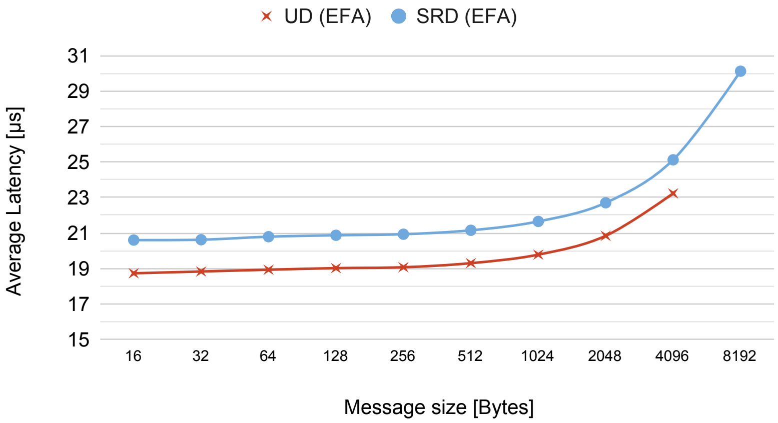


Figure 4.4.: EFA's SRD vs UD Protocol - Average Latency

Furthermore we evaluated the minimum latencies achieved by UD (EFA) and SRD (EFA) shown in Figure 4.5 . This describes the best case scenario for link latency with ideal instance placement and packet trajectory within the *AWS* cloud. Naturally SRD (EFA) would also benefit from this ideal scenario. From Figure 4.5 we see that even in this case, SRD (EFA) only has a slightly higher latency of around 2 μs.

Therefore, we think the trade-off to sacrifice reliability to get a slight latency improvement is often not worthwhile for EFA. Since many applications require reliable delivery, and if not provided from the hardware as in SRD, it needs to be handled by the software stack, which increases code complexity and may equalize the latency improvement.

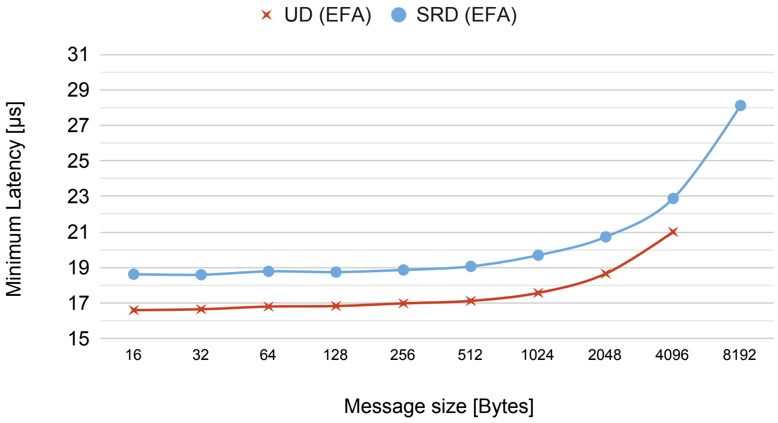


Figure 4.5.: EFA's SRD vs UD Protocol - Minimum Latency

4.3. Synchronous Bandwidth

In this section, we evaluate the effect of message size on synchronous bandwidth using a single thread. Synchronous in this context, refers to a communication model where after each transmission request, we wait for its completion event before posting the next one. SRD and RC RDMA both adhere to a delivery-complete semantics which means that the completion is generated when the remote network card receives the message. For this setup, we can see in Figure 4.6 that RDMA achieves a much higher bandwidth. The reason is that RDMA's latency is much smaller and thus when cross-referencing the bandwidth results with the latency results from Figure 4.2 the achieved bandwidth is not surprising. Therefore, the achievable bandwidth of both (EFA and RDMA) is limited by their respective latencies. For instance, with 8 kB RDMA achieves roughly 2.5 GB/s which is 10 times more than SRD achieves in this setup (similar to the latency gap).

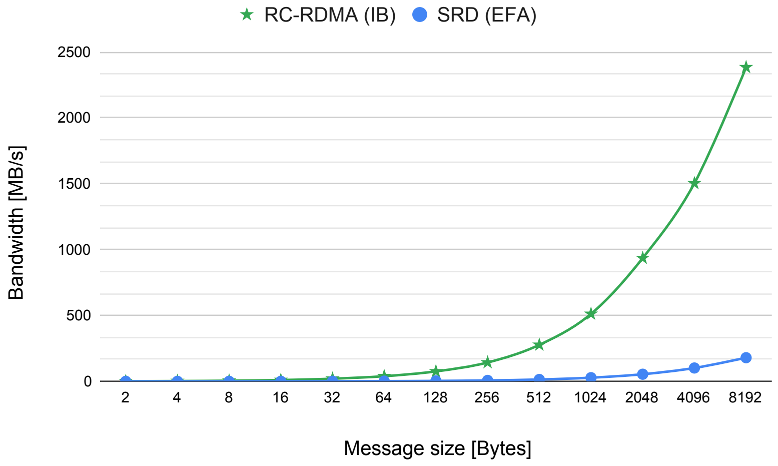


Figure 4.6.: Impact of Message Size on Synchronous Bandwidth

4.4. Asynchronous Bandwidth - Message Rate

As shown above, latency becomes the limiting factor in synchronous networking. Subsequently, we investigate how asynchronous networking avoids this limitation. Therefore, we vary the number of outstanding messages (i.e, the transmission depth) and show in Figure 4.7 how the bandwidth and Figure 4.8 how the message rates are affected.

First, we focus on larger messages. With 4 kB messages and a transmission depth of about 64, RDMA achieves the maximum bandwidth (i.e., around 12GB/s). In contrast, EFA does not fully saturate the bandwidth and instead seems to be message bound at around 2 M messages, i.e., the same message rate as with smaller message sizes. When messages are sufficiently large, i.e., 8 kB and larger, both EFA and RDMA achieve the maximum bandwidth. However, Figure 4.7 shows that only RDMA is able to reach the full bandwidth when the transmission depth is small. The reason for that lies in RDMA's lower latency since it enables RDMA to process outstanding messages more quickly. For instance, a transmission depth of 8 means that we always try to have 8 messages outstanding. As RDMA's latency is lower, the completion for these 8 outstanding messages is generated faster and new messages can be transmitted. Conversely, because EFA's latency is higher the completion takes longer and thus requires a higher transmission depth to achieve the maximum bandwidth.

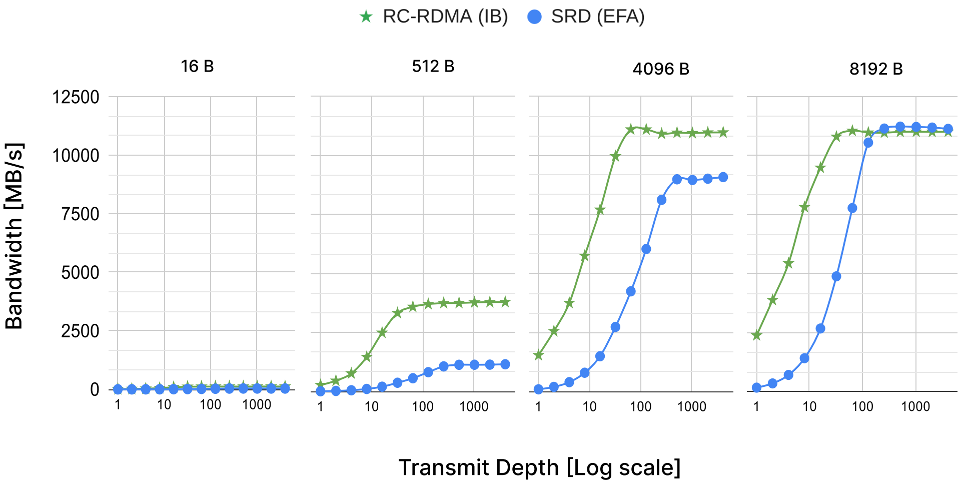


Figure 4.7.: Impact of Transmission Depth (Outstanding Operations) on Async Bandwidth

We now focus on smaller messages, i.e., 16 and 512 byte. where we mainly focus on the message rate. When comparing the respective message rates for both 16 and 512 byte, we can observe from Figure 4.8 that the message size does not affect the maximum message rate. RDMA achieves around 7 M messages per second and EFA around 2 M messages per second for both message sizes. One may now wonder what the limiting factor for EFA's message rate is. What we can already rule out is being latency-bound because we send multiple messages asynchronously. Additionally, we are not bandwidth bound either for 16 and 512 byte messages as we can clearly observe from Figure 4.7. Hence, we argue that the message rate is limited by the network card, as discussed in upcoming evaluations.

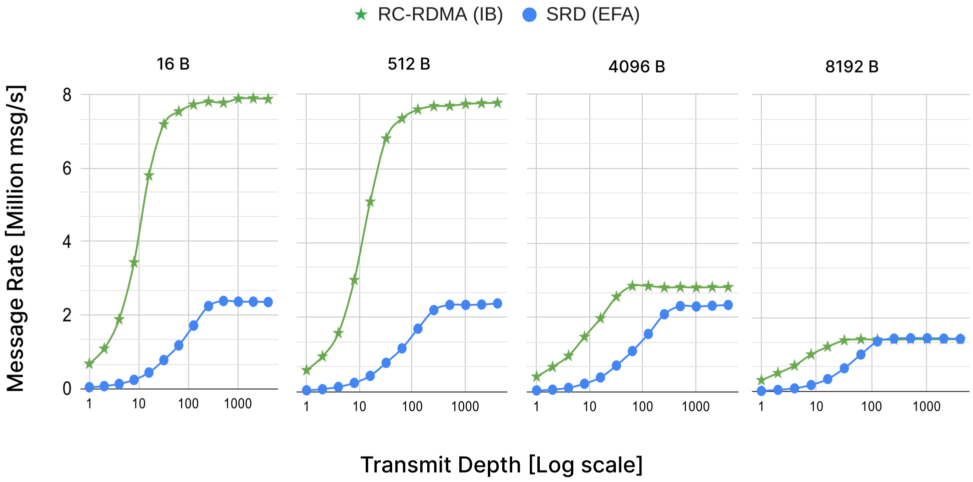


Figure 4.8.: Impact of Transmission Depth (Outstanding Operations) on Message Rate

4.5. NIC Parallelism

In this section, we evaluate if Asynchronous bandwidth is limited by the Data Processing Unit (DPU) of the network card. This can happen if a powerful CPU core overwhelms a less powerful DPU on the NIC. Often a single connection is handled by a single NIC DPU [25]. Consequently, increasing the number of connections may lead to the utilization of multiple NIC DPUs, which ultimately improves the message rates.

To evaluate if NIC parallelism can be exploited we use a single thread and increase the number of connections. Figure 4.9 shows that a single CPU core utilizes multiple connections with SRD and thus achieves a peak message rate of around 4 M with 4 connections (16 and 512 byte messages). With 4 kB message size, EFA eventually reaches the full bandwidth with 2 connections (same message rate as RDMA which achieves full bandwidth). In contrast, RDMA does not profit from having multiple connections, instead, performance degrades slightly for smaller messages. We can conclude that RDMA is CPU-bound with small messages. We confirmed this statement by using a single connection and posting a linked list of multiple messages to the NIC, i.e., in perftest this is done by setting the *post list* parameter to 16. This allows the NIC to process the requests at full speed without being dependent on the CPU. For RDMA the message rate with 64 byte messages peaked at around 17 M messages per second whereas EFA's messages per second remained at 2 M due to the processing limit of a single NIC DPU.

Based on these results, we can reasonably speculate that a single Nitro Card DPU can achieve around 2 M messages per second.

An important point of note here is the usage of a single CPU thread to manage all these connection pairs. We achieve a maximum of 4 M messages per second by exploiting multiple Nitro Card DPUs. However, this does not imply that the entire NICs maximum message rate is 4 M messages per second. We later evaluate benchmarks with multiple threads to determine the peak message rate of the EFA NIC.

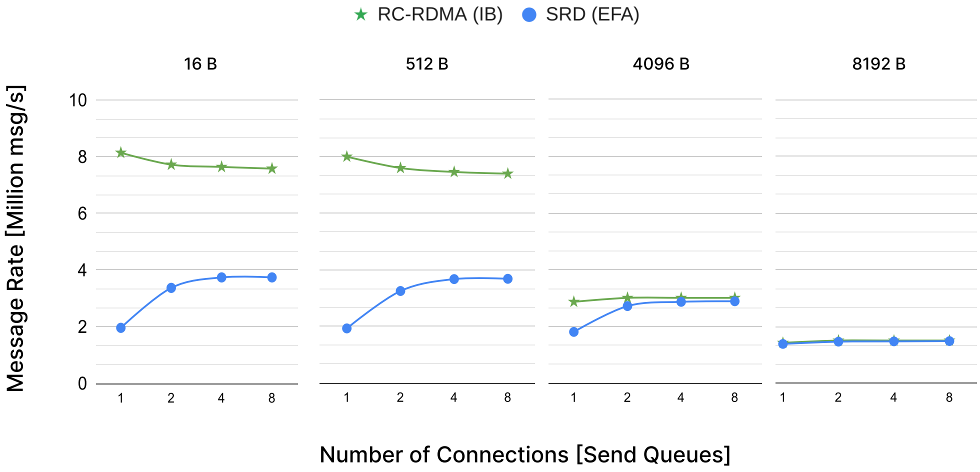


Figure 4.9.: Impact of Connection Pairs (Send queue) on Message Rate

4.6. Multi-threaded Evaluation

To determine peak fabric performance and to eliminate most CPU bottlenecks impacting our single-threaded evaluations, we turn to evaluating benchmarks with multiple threads.

We do not use the *perftest* benchmark suite since it does not support multi-threading. Instead, we use *efa-bench* with both the *EFA provider* and *VERBS provider* to evaluate EFA and RC-RDMA respectively. *libfbaric* has substantial CPU overhead as we will show in the next experiment. However, the effect of this overhead is compensated for in this experiment by using multiple CPU resources.

Figure 4.10 shows how increasing the number of CPU threads affects bandwidth. For 4 kB message size, EFA and RDMA reach the full bandwidth. Whereas RDMA reaches the bandwidth limit already with 2 threads, EFA requires at-least 4 threads.

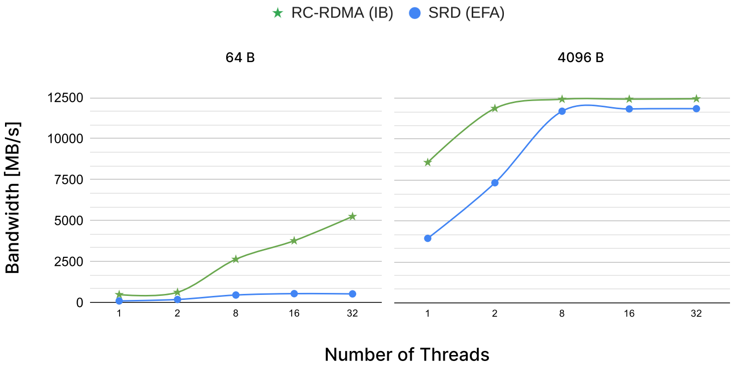


Figure 4.10.: Impact of Thread Count on Bandwidth (TX Depth - 128)

As shown in Figure 4.11, the message rate with 64 byte messages for EFA is capped at around 8 M. Collectively evaluating this with the results from Figure 4.9 where we deduced that a single NIC DPU reaches 2 M messages per second we can speculate that the NIC may have 4 DPUs in total. On the other hand, the RDMA network card achieves more than 30 M messages per second.

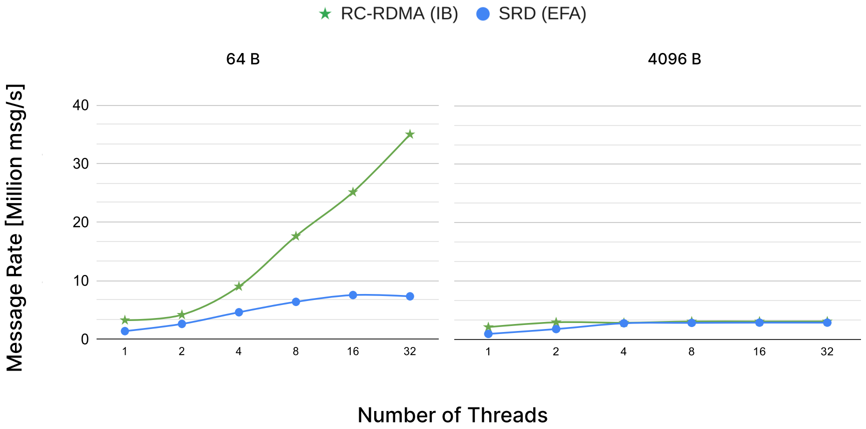


Figure 4.11.: Impact of Thread Count on Message Rate (TX Depth - 128)

Finally we evaluate how well EFA multi-threaded experiments scale at various Transmission depths (Outstanding operations). As shown in Figure 4.12 at smaller transmission depths, with an increase in thread count, we see a corresponding linear increase in bandwidth. However at higher TX Depths, we notice thread counts do not scale so well, this might be a direct consequence of total outstanding operations at the EFA NIC. For instance using 16 threads each with a TX Depth of 100, translates to an effective 1600 outstanding operations at the NIC. This is already well beyond the point of diminishing returns. Further increasing this to 32 threads each with a TX Depth of 100 leads to an effective outstanding 3200 operations, without a substantial increase to overall bandwidth/message rate.

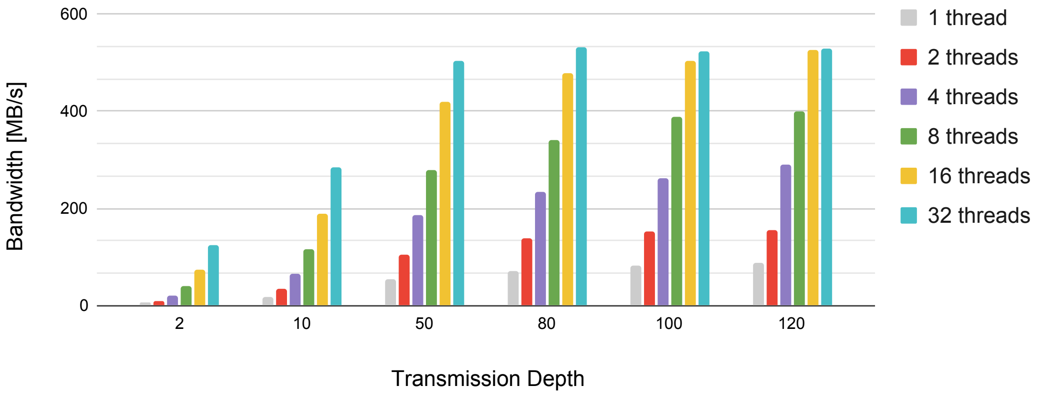


Figure 4.12.: EFA Bandwidth Scaling w.r.t Thread Count at Various TX Depths (Msg Size - 64 Bytes)

4.7. Message Segmentation Overhead

We next evaluate specific characteristics mostly relevant to EFA. As mentioned in section 2.4.4 EFA's SRD protocol natively only supports message sizes upto link MTU, 8760 bytes. However the capability to send messages larger than this is entirely implemented in software by *libfabric*. In this section, we evaluate the overhead introduced by this implementation.

We start with a message size lower than the MTU limit and gradually increase message size in small increments. As shown in Figure 4.13 we notice an abrupt jump in latency at the segmentation boundary of about 8760 bytes. This overhead is mainly due to packet reassembly that must take place at the destination node, before the message is considered received. The vertical axis in Figure 4.13 has intentionally been zoomed in to portray this effect. However in absolute the terms, the overhead is only around 1 μs. Given the relatively high baseline of EFA's latency, this overhead is insignificant.

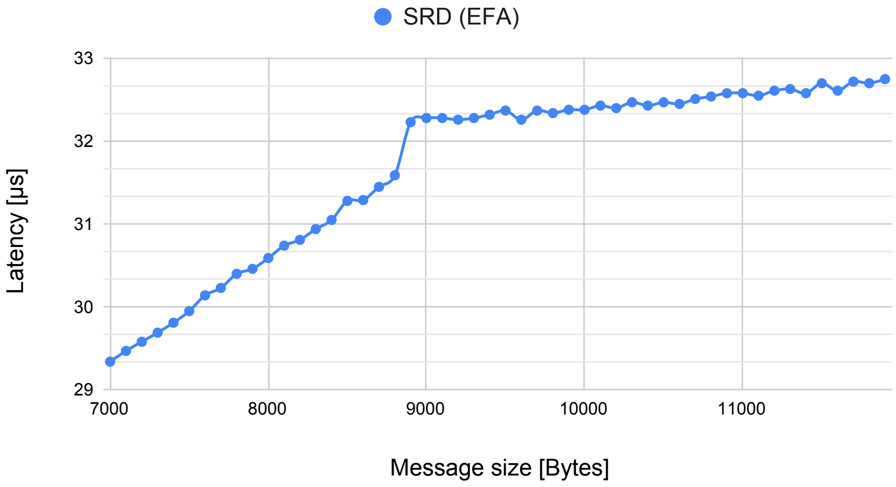


Figure 4.13.: Average Link latency at Segmentation Boundary

4.8. Link Latency at Saturation

In this section we evaluate the impact of NIC saturation on Average latency. For this evaluation, we fix our message size at 4096 Bytes. Using a set of 8 traffic generator threads, we saturate the NIC and the link to a desired *Saturation bandwidth*. As described in section 3.2.6, this is controlled using a *SATURATION_WAIT* factor.

Figure 4.14 shows how the EFA fabric handles NIC saturation and it's impact on average latency. Under normal operating conditions, for a message size of 4096 Bytes we expect an average latency of about 25 μs with SRD. With a decrease in *SATURATION_WAIT* factor, or delay between subsequent requests from the traffic generator, all NIC resources such as command queues, DPUs, would be under an increasing amount of load. This is shown by a gradual increase in latency upto a load of about 10 GB/sec. After which point we see a steep increase in latency. At peak saturation, we see latencies of about 50 μs for a 4096 Byte packet, which is almost twice the normal unsaturated latency.

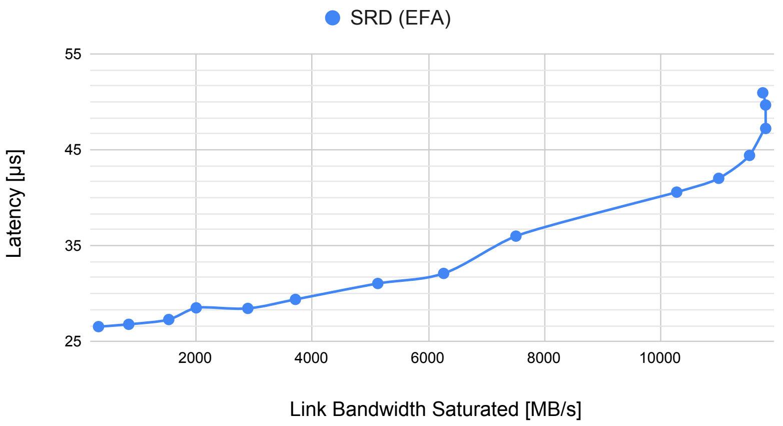


Figure 4.14.: Average Link latency at Saturation (Message Size - 4096 Bytes)

4.9. *libfabric* Message Transfer API Variants

Next we turn our attention to *libfabric* specific features. *libfabric* provides various APIs for the *send/recv* primitives. In this section we evaluate Tagged message transfers (*fi_tsend*, *fi_trecv*), Inject message transfers (*fi_inject*, *fi_tinject*) and how they compare to the traditional message APIs (*fi_send*,*fi_sendmsg*, *fi_recv*).

Naturally, tagged message transfers requires additionally computation on the receiver side for tag matching. Our aim is to evaluate its overhead if any on latency. As shown in Figure 4.15, we see no noticeable overhead with tagged transfers, neither do we see any meaningful improvement to latency with Inject transfers. This might be due to the EFA provider not supporting selective completions in libfabric [21]. In summary, if the application benefits from segregating transfer streams based on tags and uses the Tagged transfer APIs, we expect to see no overhead.

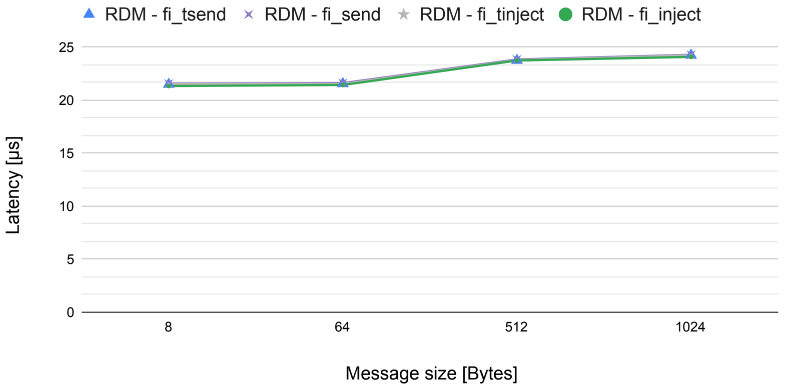


Figure 4.15.: *libfabric* Message Transfer API variants Impact on Average Latency

4.10. One Sided Operations - RMA

In this section we evaluate libfabric's emulated one sided RMA operations for EFA. An important point to note here is just like in two-sided communication, there are still two endpoints involved in one-sided communication. EFA provider in *libfabric* requires the responder application to keep the progress engine running to facilitate the communication.

First, we consider synchronous performance of the one sided operations and compare it with the *send* API. As expected, Figure 4.16 shows *send* and *write* behave very similar to each other. However the emulated *read* API has a noticeable overhead. This might be due to how *read* is implemented within the EFA RDM Protocol [14], requiring two packets namely *REQUEST_TO_READ* and *READ_RESPONSE* to complete one *read* operation. We next evaluated the asynchronous bandwidth performance of the two emulated operations as shown in Figure 4.17. Here again the results reiterate the same fact, showing a wider gap in performance between *read* and *write*.

Therefore, although the emulated one-sided operations provide a seamless interface for porting applications from one network device that supports one-sided operations to another, they do not provide any performance benefit like true one-sided operations, such as in RDMA.

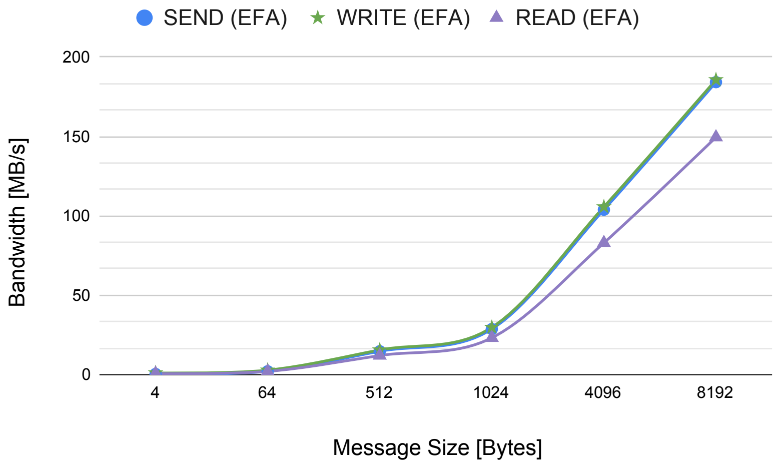


Figure 4.16.: Performance of Emulated RMA Operations Compared to *send/recv*

4.11. Interface Evaluation - *ibverbs* vs *libfabric* vs *MPI*

We now compare the performance of *libfabric* with the lower-level *ibverbs* interface. We examine the bandwidth as well as the message rate in Figure 4.18. To verify that our *libfabric* implementation performs as expected, we compared it to the OSU MPI performance test, which uses *libfabric* as well. Figure 4.18 shows that using the lower-level library *ibverbs* yields the best performance, i.e., around 50% more messages per second.

When using multiple threads, we may hit the hardware limits of the EFA NIC. Here there is an argument to be made for *libfabric*'s high level interface. However, we found that the library had many

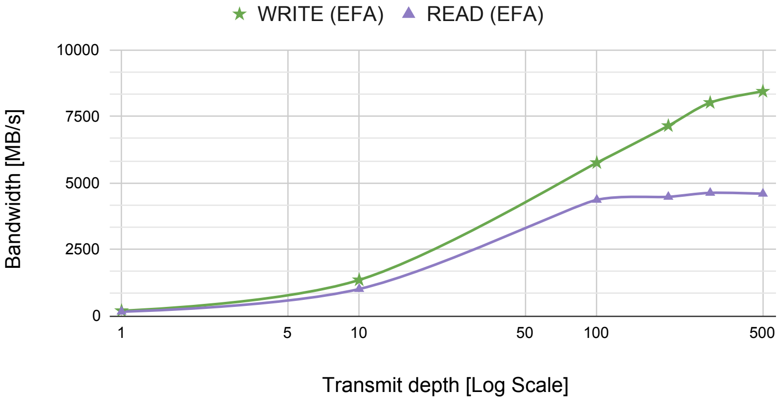


Figure 4.17.: Overhead of Emulated RMA *read* Compared to RMA *write* (Message Size - 8 kb)

performance knobs and thus obtaining optimal performance was a challenge. If the application does not intend to use any of *libfabric*'s feature set or emulated capabilities, *ibverbs* might be the optimal choice.

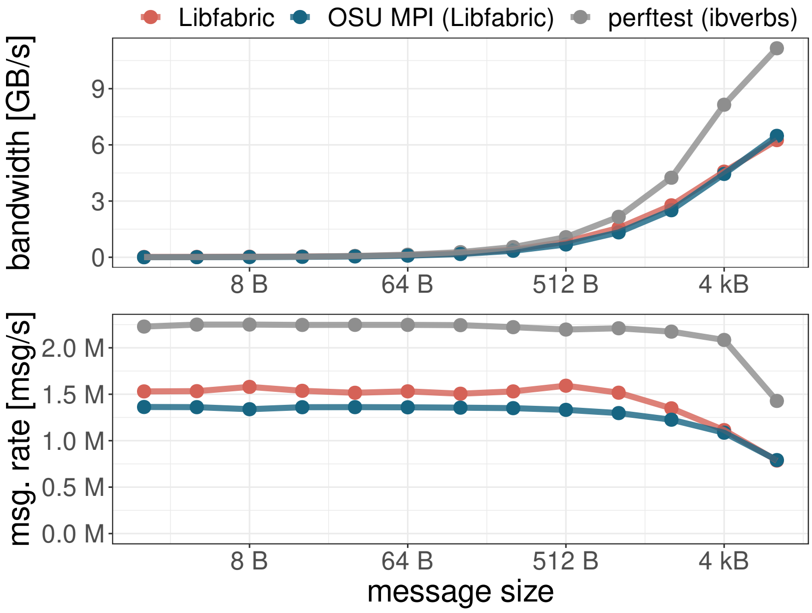
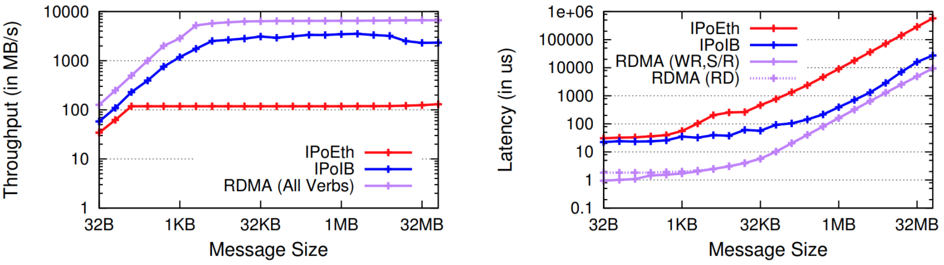


Figure 4.18.: Performance Implication of EFA Interfaces (TX Depth - 256)

5. Related Work

As highlighted in previous sections, EFA has primarily been marketed towards HPC workloads by AWS. As a consequence most research addressing EFA has also primarily been driven by the HPC community. We think that the broad availability of EFA has the potential to trigger a redesign of distributed databases. Unfortunately, there has only been very limited work on EFA in general.

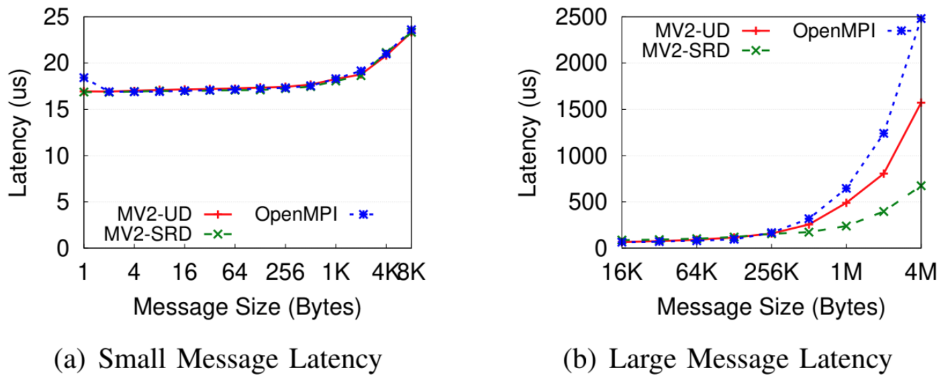
RDMA on the other hand has been extensively researched by the database community, with papers systematically evaluating RDMA's performance characteristics[7, 25, 50]. The paper by Binnig et.al [7] evaluates various high performance network fabrics and argues that distributed DBMS architecture needs a fundamental re-work to fully exploit these networks. Figure 5.1 briefly describes the RDMA performance characteristics highlighted by the paper. There are other papers that systematically investigate RDMA communication models and primitives for large scale high performance systems [50]. We draw inspiration from some of these research works to build our evaluation strategy.



(a) Throughput (b) Latency

Figure 5.1.: RDMA Performance evaluation vs Ethernet-IP (Binnig et.al)[7]

Most EFA research is geared towards MPI as it is the de-facto standard in distributed HPC applications [42, 47, 8]. Notable is this paper from Shalev et al. (Amazon) [42] which discusses some of the design decisions behind the proprietary SRD protocol. Researchers have also explored design changes and performance improvement in cloud based systems against in-house clusters notable Zhai et al [48]. The other two papers [47, 8] specifically focus on MPI primitives and on how different MPI implementations perform with EFA. They explore the features and performance characteristics of EFA in the context of high performance MPI libraries. They propose a zero-copy design to improve network performance involving large messages. Various communication patterns such as Point-to-point, collective performance involving one-to-all, OpenMPI application performance with 3DStencil, Cloverleaf, MiniGhost were explored. Figure 5.2 briefly describes their analysis, showcasing point to point latency performance of MPI over OSU Micro-Benchmarks(MV-2) versus traditional OpenMPI[8].



(a) Small Message Latency (b) Large Message Latency

Figure 5.2.: MPI Latency performance on EFA (Sourav et.al)[8]

There are some papers in the machine learning community which use EFA for distributed training [44, 49]. However, they focus on the machine learning part and use EFA transparently. The only database paper which mentions EFA is from Barthels et al. [6] in which they show in one experiment that their findings are transferable to the off-the-shelf network infrastructure of AWS.

There are also white-papers directly published by AWS in collaboration with Intel on the application of EFA in HPC workloads [35, 42]. They examine EFA with other AWS services to efficiently scale HPC applications. In addition to this, the SRD protocol has also been explained in great detail along with its application in data-center networking. The paper from Shalev et.al [42] first explain the design behind SRD, mainly focusing on three key areas - multi-path load-balancing, out of order delivery and congestion control. This is followed by an in-depth performance evaluation of SRD comparing it to TCP. They show SDR's multipath load balancing and congestion control on Nitro cards both decreases packet drop in the network as well as faster recovery from drops. As highlighted in Figure 5.3, all of these characteristics contribute to exceptionally good flow fairness in SRD.

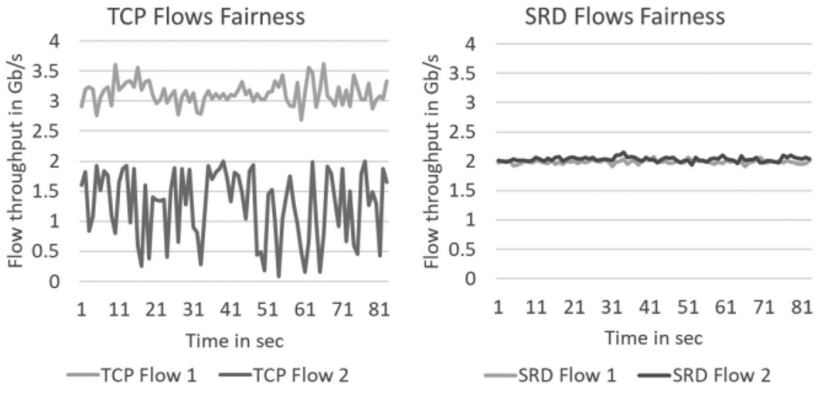


Figure 5.3.: SRD Flow Fairness compared to TCP/IP (Shalev et.al) [42]

6. Conclusion and Future Work

Although both EFA and RDMA over InfiniBand are advertised for a similar audience, their performance characteristics are quite different. This has major implications on how distributed data processing systems for the cloud should be designed and optimized. In the following section, we summarize our main findings and their implications on system design.

6.1. Implications for System Design

Latency. As we have seen in Figure 4.2, EFA's latency decreased twofold compared to traditional TCP/IP sockets. Nonetheless, the latencies of EFA are still an order of magnitude higher than those of RDMA. Although the latency results are better than TCP, there is a substantial gap in performance w.r.t RDMA. Also since EFA is natively a cloud network fabric, cloud deployment factors such as placement strategies are important to ensure optimal latency. Latency is a very important network parameter that has direct implications for system design. Due to EFA's higher latency it is not as beneficial to send small messages as in RDMA. Therefore, RDMA techniques which aim at decreasing the message size, such as message shrinking [25] and inline optimization, are not as effective in EFA. Rather, to amortize the high latency, techniques such as batching or piggybacking should be considered for system design.

Bandwidth. As shown in Figure 4.7, the bandwidth of EFA is strongly dependent on the transmission depth and the message size. For smaller messages sizes, extensive optimizations are required to exploit bandwidth performance. To saturate the bandwidth, a large transmission depth (e.g., 256) and message sizes are important. When the message size is below 8 kB, NIC-parallelism should be exploited by either using multiple connections for a single-threaded application or multiple threads. With even larger messages (e.g., 8 kB) those sophisticated optimizations are not necessary to achieve the full bandwidth.

Message Rate. Figure 4.8 shows that the achievable message rate for small messages in EFA is considerably smaller than in RDMA. This has a direct implication on other network parameters. Since AWS usually pairs these EFA adapters with high-end instances, CPU performance is not of particular concern. Multiple flows are needed to fully exploit the NIC. NIC parallelism is crucial to utilize the maximum message rate, preferable with multiple threads to achieve 8 M messages/second. This directly synergizes with SRD characteristics of efficiently handling multiple simultaneous network flows.

No one-sided operations. Besides performance characteristics, other factors such as the functionality of the interface play an important role. For example, one-sided operations, the key primitives for many state-of-the-art distributed RDMA systems, are not natively available in EFA. However a higher level framework such as *libfabric* could be used to provide one-sided primitive capabilities. Such an implementation completely done in software naturally comes with a performance penalty as shown in 4.16. System designs based on one-sided primitives might need to be revisited.

Out-of-order-delivery. In contrast to reliable connected RDMA, EFA does not natively guarantee in-order delivery. This can however be accomplished using *libfabric* by specifying a Message ordering flag such as *FI_ORDER_SAS* [16]. Systems which rely on ordering need to handle re-ordering in the software stack which may induce some overhead. However considering the limitations imposed by AWS on EFA enabled instances, specifically the fact that EFA traffic is non-routable and has to work within the same subnet, such out-of-order deliveries pose less of a challenge.

Ibverbs vs. libfabric. The available EFA interfaces offer different characteristics. Each of these come with certain advantages and disadvantages depending on the applications requirement. We argue that for database systems the low-level *ibverbs* is likely better suited. *Ibverbs* yields better performance and some higher-level functions of *libfabric* may impact the database system. For instance, the emulation of one-sided verbs might interfere with the thread scheduler of the database system. Libfabric is advantageous if the system/ application is expected to run on different fabrics for the provider can be simply exchanged. Therefore, when performance is paramount, evidently it is best to use the low level library directly.

EFA is proprietary. Unlike InfiniBand, it is not yet possible to equip on-premise machines with EFA. This has major implications for certain software systems. Migration to the AWS cloud is essentially a non-negotiable prerequisite for interconnecting systems with EFA. Therefore, EFA can only be exclusively used in AWS EC2 instances leading to vendor lock-in.

EFA is not standalone. As a direct consequence of being a proprietary cloud fabric, EFA's pairings and offerings are directly controlled by AWS. Applications are forced to use specific instance types that support EFA. In majority of the cases this is not an issue as high-performance network requirement usually mandates a high-capacity compute/memory setup as well. AWS also provides a wide variety of instances that support *Enhanced Networking*[37]. However the fact that EFA is not a standalone product is to be noted.

6.2. Conclusion

Overall, even though EFA does not yet come close to the latency of RDMA over InfiniBand, we believe that it comes with potential for distributed database systems. This is because EFA is already widely available in AWS today [36] which comes with a wide offering of available compute instances. Compared to TCP/IP, which was the only available networking option in AWS before, EFA considerably reduces the latency.

Making an argument for applications already deployed on the cloud is fairly straightforward. Applications previously running on the ENA Network Adapter will see immediate benefits with little to no change. The following concerns should however be noted. EFA's traffic limitations imposed by AWS and its availability on the instance type desired by the application. For cloud non-native applications, the argument to use EFA becomes more challenging. Changing an application's architecture to suit the cloud is a significant investment. Disadvantages arising from vendor lock-in may be a cause for concern. Currently AWS offers the EFA adapter free of charge, but this might change in the future leading to cost overruns. The choice of EFA's programming interface can mitigate some of these concerns. For instance using a high-level framework such as *libfabric* offers inter-operablity with various underlying network fabrics. This however comes with its own set of performance implications as shown in our research.

Finally, compared to the alternatives offered by other major cloud providers, EFA proves to be a solid choice for data intensive applications. We believe that some of the current limitations of EFA, such as its low message rate, may be resolved by future Nitro Cards generations. AWS is also rapidly improving its instance support for EFA, leading to more economical high-performance systems.

6.3. Future Work

Our work primarily focused on evaluating the EFA network fabric against RDMA. There are similar high performance network offerings by other cloud service providers outlined in section 2.1. New network fabrics in search of lower latencies are constantly being introduced into this eco-system, for instance Google's *Aquila* inspired by InfiniBand promises extremely low latencies [32]. We believe there is scope for EFA to be evaluated against such cloud-native fabrics.

Additionally we also compared various programming interfaces available for EFA and highlighted their performance characteristics. Compared to *ibverbs*, there is a noticeable overhead in our micro-benchmark suite *efa-bench* as well as *OSU-MPI*, both based on *libfabric* [Section: 4.11]. Further analysis is needed to empirically determine the cause of this overhead.

All things considered, EFA is still a relatively new network fabric. As a consequence its software and hardware stack is rapidly evolving. With the introduction of newer generation Nitro cards, certain traits such as message-rate, peak throughput might need to be re-evaluated.

Bibliography

[1] S. Babu et al. “Massively parallel databases and mapreduce systems”. In: *Foundations and Trends in Databases*. 2013.

[2] OpenFabrics Alliance. *Infiniband Verbs Performance Tests*. 2022. URL: https://github.com/linux-rdma/perftest.

[3] Amazon AWS. *Enhanced networking on Linux*. 2022. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html.

[4] Microsoft Azure. *RDMA-capable instances*. 2022. URL: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-hpc#rdma-capable-instances.

[5] Microsoft Azure. *What is Accelerated Networking?* 2022. URL: https://docs.microsoft.com/en-us/azure/virtual-network/accelerated-networking-overview.

[6] Claude Barthels et al. “Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores”. In: *PVLDB* 12.13 (2019).

[7] Carsten Binnig et al. “The End of Slow Networks: It’s Time for a Redesign”. In: *PVLDB* 9.7 (2016).

[8] Sourav Chakraborty et al. “Designing Scalable and High-Performance MPI Libraries on Amazon Elastic Fabric Adapter”. In: *2019 IEEE Symposium on High-Performance Interconnects, HOTI 2019,* IEEE, 2019.

[9] Michael Dalton et al. “Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization”. In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 373–387. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/nsdi18/presentation/dalton.

[10] Dwarakanandan. *efa-bench Micro-Benchmark suite*. 2022. URL: https://github.com/dwarakanandan/efa-libfabric-bench.

[11] Ruediger Gad et al. “Network performance in virtualized environments”. In: *2011 17th IEEE International Conference on Networks*. 2011, pp. 275–280. DOI: 10.1109/ICON.2011.6168488.

[12] Google. *gflags library*. 2022. URL: https://github.com/gflags/gflags.

[13] Brendan Gregg. *CPU Flame Graphs*. 2022. URL: https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html.

[14] OpenFabrics Interfaces Working Group. *EFA RDM Communication Protocol version 4*. 2021. URL: https://github.com/ofiwg/libfabric/blob/main/prov/efa/docs/efa_rdm_protocol_v4.md.

[15] OpenFabrics Interfaces Working Group. *Fabtests*. 2022. URL: https://github.com/ofiwg/libfabric/tree/main/fabtests.

[16] OpenFabrics Interfaces Working Group. *Libfabric fi_endpoint*. 2022. URL: https://ofiwg.github.io/libfabric/v1.13.2/man/fi_endpoint.3.html.

[17] OpenFabrics Interfaces Working Group. *Libfabric fi_getinfo*. 2022. URL: https://ofiwg.github.io/libfabric/v1.13.2/man/fi_getinfo.3.html.

[18] OpenFabrics Interfaces Working Group. *Libfabric fi_msg API Manual Page*. 2022. URL: https://ofiwg.github.io/libfabric/v1.13.2/man/fi_msg.3.html.

[19] OpenFabrics Interfaces Working Group. *Libfabric fi_tagged*. 2022. URL: https://ofiwg.github.io/libfabric/v1.13.2/man/fi_tagged.3.html.

[20] OpenFabrics Interfaces Working Group. *Libfabric library*. 2022. URL: https://github.com/ofiwg/libfabric/releases/.

[21] OpenFabrics Interfaces Working Group. *Libfabric Programmer Manual: fi_efa*. 2022. URL: https://ofiwg.github.io/libfabric/v1.13.2/man/fi_efa.7.html.

[22] OpenFabrics Interfaces Working Group. *OFI Programmer's Guide*. 2022. URL: https://github.com/ofiwg/ofi-guide/blob/master/OFIGuide.md.

[23] Paul Grun et al. "A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency". In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 2015, pp. 34–39. DOI: 10.1109/HOTI.2015.19.

[24] Chengfan Jia et al. "Improving the Performance of Distributed TensorFlow with RDMA". In: *Int. J. Parallel Program.* 46.4 (2018).

[25] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Design Guidelines for High Performance RDMA Systems". In: *login Usenix Mag.* 41.3 (2016).

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs". In: *OSDI*. 2016.

[27] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Using RDMA efficiently for key-value services". In: *SIGCOMM*. 2014.

[28] Feilong Liu et al. "Beyond MPI: New Communication Interfaces for Database Systems and Data-Intensive Applications". In: *SIGMOD Rec.* (2020).

[29] Xiaoyi Lu et al. "High-performance design of apache spark with RDMA and its benefits on various workloads". In: *Big Data*. 2016.

[30] Mellanox. *Sockperf*. 2022. URL: https://github.com/Mellanox/sockperf.

[31] Microsoft. *HC-series virtual machine sizes*. 2021. URL: https://docs.microsoft.com/en-us/azure/virtual-machines/workloads/hpc/hc-series-performance.

[32] Timothy Prickett Morgan. *WITH AQUILA, GOOGLE ABANDONS ETHERNET TO OUTDO INFINIBAND*. 2022. URL: https://www.nextplatform.com/2022/04/12/with-aquila-google-abandons-ethernet-to-outdo-infiniband/.

[33] Google Cloud Platform. *Google Cloud networking in depth*. 2022. URL: https://cloud.google.com/blog/products/networking/google-cloud-networking-in-depth-how-andromeda-2-2-enables-high-throughput-vms.

[34] Amazon web services. *Linux kernel driver for Elastic Fabric Adapter (EFA)*. 2022. URL: https://github.com/amzn/amzn-drivers/tree/master/kernel/linux/efa.

[35] Amazon web services. *Lower the Time-to-Results for Tightly Coupled HPC Applications on the AWS Cloud with the Elastic Fabric Adapter*. 2020. URL: https://d1.awsstatic.com/HPC2019/Lower-Time-To-Results-wtih-EFA-Jan2020.pdf.

[36] Amazon Web Services. *EFA is now mainstream, and that's a Good Thing*. 2021. URL: https://aws.amazon.com/blogs/hpc/efa-is-now-mainstream/.

[37] Amazon Web Services. *EFA Supported Instance Types*. 2020. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/efa.html#:~:text=2.4.2%20and%20later-,Supported%20instance%20types,-The%20following%20instance.

[38] Amazon Web Services. *EFA User-guide*. 2022. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/efa.html.

[39] Amazon Web Services. *Placement Groups*. 2022. URL: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html.

[40] Amazon Web Services. *Security Groups*. 2022. URL: https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html.

[41] Amazon Web Services. *Virtual Private Cloud*. 2022. URL: https://docs.aws.amazon.com/vpc/latest/userguide/what-is-amazon-vpc.html.

[42] Leah Shalev et al. "A Cloud-Optimized Transport Protocol for Elastic and Scalable HPC". In: *IEEE Micro* 40.6 (2020).

[43] Ryan Shea et al. "A deep investigation into network performance in virtual machine based cloud environments". In: *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*. 2014, pp. 1285–1293. DOI: 10.1109/INFOCOM.2014.6848061.

[44] Indu Thangakrishnan et al. "Herring: rethinking the parameter server at scale for the cloud". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*. 2020.

[45] Network-Based Computing Laboratory - Ohio State University. *OSU Micro-Benchmarks*. 2022. URL: https://mvapich.cse.ohio-state.edu/benchmarks/.

[46] Jerome Vienne et al. "Performance Analysis and Evaluation of InfiniBand FDR and 40GigE RoCE on HPC and Cloud Computing Systems". In: *2012 IEEE 20th Annual Symposium on High-Performance Interconnects*. 2012, pp. 48–55. DOI: 10.1109/HOTI.2012.19.

[47] Shulei Xu et al. "MPI Meets Cloud: Case Study with Amazon EC2 and Microsoft Azure". In: *Fourth IEEE/ACM Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware, IPDRM@SC 2020*. IEEE, 2020.

[48] Yan Zhai et al. "Cloud versus In-House Cluster: Evaluating Amazon Cluster Compute Instances for Running MPI Applications". In: *State of the Practice Reports*. SC '11. Seattle, Washington: Association for Computing Machinery, 2011. ISBN: 9781450311397. DOI: 10.1145/2063348.2063363. URL: https://doi.org/10.1145/2063348.2063363.

[49] Shuai Zheng et al. "Accelerated Large Batch Optimization of BERBT Pretraining in 54 minutes". In: *CoRR* abs/2006.13484 (2020). URL: https://arxiv.org/abs/2006.13484.

[50] Binnig C Ziegler T Leis V. *RDMA Communciation Patterns*. 2020. URL: https://doi.org/10.1007/s13222-020-00355-7.

A. *Efa-bench* Micro-benchmark Suite

A.0.1. Prerequisites

- g++ >= 7.0
- cmake >= 3.10
- gflags >= 2.2.2 (Repository:[12])
- libfabric >= 1.13.2 (Repository: [20])

A.0.2. Build

Clone the *efa-bench* repository [10], and make sure all the prerequisites are installed. After which the following steps can be followed to build the project.

```
mkdir build && cd build
cmake ..
make
```

A.0.3. Help and Flags

```
-batch (Batch size) type: uint32 default: 1000

-benchmark_type (Type of benchmark Eg: batch, latency, inject, ping_pong,
  rma, rma_batch, rma_inject, rma_sel_comp, rma_large_buffer, sat_latency)
  type: string default: "batch"

-cq_try (Factor used in combination with batch size to determine CQ
  retrievals) type: double default: 0.8

-debug (Print debug logs) type: bool default: false

-dst_addr (Destination address) type: string default: "127.0.0.1"

-endpoint (Endpoint type Eg: dgram, rdm) type: string default: "dgram"

-fabinfo (Show provider info) type: bool default: false

-hw_counters (Path to network interface HW Counters) type: string
  default: "/sys/class/infiniband/rdmap0s6/ports/1/hw_counters/"

-iterations (Number of transfers to perform) type: uint32 default: 10
```

```
-mode (Mode of operation Eg: server, client) type: string default: "server"

-oob_port (Port for out of band sync/exchange) type: uint32 default: 46500

-payload (Size of transfer payload in Kilobytes) type: uint32 default: 64

-port (Port for data transfer) type: uint32 default: 47500

-provider (Fabric provider Eg: sockets, efa) type: string
  default: "sockets"

-rma_op (RMA Operation Eg: read/write) type: string default: "write"

-run_all (Run benchmark for all payloads) type: bool default: false

-runtime (Benchmark run-time in seconds) type: uint32 default: 30

-saturation_wait (Wait interval between iterations in nano seconds)
  type: uint32 default: 10000

-stat_file (Output stats file name) type: string default: "stat-file"

-tagged (Use Tagged message transfer) type: bool default: false

-threads (Thread count / Num of connection streams) type: uint32 default: 1
```

A.0.4. Sample Usage

Fabric Info:

```
./benchmark --fabinfo
```

Server/Client Mode:

```
./benchmark --flagfile=server.conf
./benchmark --flagfile=client.conf
```

Sample configuration flags:

```
--mode=server
--benchmark_type=batch
--port=47500
--oob_port=46500
--provider=efa
--endpoint=rdm
--iterations=100000
--payload=1024
--batch=100
--cq_try=0.8
--rma_op=write
--runtime=5
--threads=1
--hw_counters=/sys/class/infiniband/rdmap0s6/ports/1/hw_counters/
```

B. EFA Fabric Info

EFA *libfabric* RDM endpoint capabilities extracted using the *fi_getinfo* API[17]. These are selectively used while configuring the fabric.

```
fi_info:
    caps: [ FI_MSG, FI_RMA, FI_TAGGED, FI_ATOMIC, FI_READ, FI_WRITE, FI_RECV, FI_SEND,
    FI_REMOTE_READ, FI_REMOTE_WRITE, FI_MULTI_RECV, FI_LOCAL_COMM, FI_REMOTE_COMM,
    FI_SOURCE, FI_DIRECTED_RECV ]
    mode: [ FI_MSG_PREFIX ]
    addr_format: FI_ADDR_EFA
    src_addrlen: 32
    dest_addrlen: 0
    src_addr: fi_addr_efa://[fe80::39:f9ff:fe17:bdbc]:0:0
    dest_addr: (null)
    handle: (nil)
    fi_tx_attr:
        caps: [ FI_MSG, FI_RMA, FI_TAGGED, FI_ATOMIC, FI_READ, FI_WRITE, FI_SEND ]
        mode: [ FI_MSG_PREFIX ]
        op_flags: [  ]
        msg_order: [ FI_ORDER_SAS, FI_ORDER_ATOMIC_RAR, FI_ORDER_ATOMIC_RAW,
    FI_ORDER_ATOMIC_WAR, FI_ORDER_ATOMIC_WAW ]
        comp_order: [ FI_ORDER_NONE ]
        inject_size: 3928
        size: 4096
        iov_limit: 4
        rma_iov_limit: 0
    fi_rx_attr:
        caps: [ FI_MSG, FI_RMA, FI_TAGGED, FI_ATOMIC, FI_RECV, FI_REMOTE_READ,
    FI_REMOTE_WRITE, FI_MULTI_RECV, FI_SOURCE, FI_DIRECTED_RECV ]
        mode: [ FI_MSG_PREFIX ]
        op_flags: [  ]
        msg_order: [ FI_ORDER_SAS, FI_ORDER_ATOMIC_RAR, FI_ORDER_ATOMIC_RAW,
    FI_ORDER_ATOMIC_WAR, FI_ORDER_ATOMIC_WAW ]
        comp_order: [ FI_ORDER_NONE ]
        total_buffered_recv: 0
        size: 8192
        iov_limit: 4
    fi_ep_attr:
        type: FI_EP_RDM
        protocol: FI_PROTO_EFA
        protocol_version: 4
        max_msg_size: 18446744073709551615
        msg_prefix_size: 112
        max_order_raw_size: 0
        max_order_war_size: 0
        max_order_waw_size: 0
        mem_tag_format: 0xaaaaaaaaaaaaaaaa
        tx_ctx_cnt: 1
        rx_ctx_cnt: 1
        auth_key_size: 0
    fi_domain_attr:
        domain: 0x0
        name: efa_0-rdm
        threading: FI_THREAD_SAFE
        control_progress: FI_PROGRESS_AUTO
        data_progress: FI_PROGRESS_AUTO
        resource_mgmt: FI_RM_ENABLED
        av_type: FI_AV_TABLE
        mr_mode: [ FI_MR_LOCAL, FI_MR_VIRT_ADDR, FI_MR_ALLOCATED, FI_MR_PROV_KEY ]
        mr_key_size: 4
        cq_data_size: 8
        cq_cnt: 512
        ep_cnt: 256
        tx_ctx_cnt: 1
        rx_ctx_cnt: 1
        max_ep_tx_ctx: 1
        max_ep_rx_ctx: 1
        max_ep_stx_ctx: 0
        max_ep_srx_ctx: 0
        cntr_cnt: 0
        mr_iov_limit: 1
    caps: [ FI_LOCAL_COMM, FI_REMOTE_COMM ]
    mode: [  ]
        auth_key_size: 0
        max_err_data: 0
        mr_cnt: 65535
    fi_fabric_attr:
        name: EFA-fe80::39:f9ff:fe17:bdbc
        prov_name: efa
        prov_version: 113.10
        api_version: 1.13
    fid_nic:
        fi_device_attr:
            name: efa_0
            device_id: 0xefa0
            device_version: 6
            vendor_id: 0x1d0f
            driver: efa
            firmware: 0.0.0.0
        fi_bus_attr:
            fi_bus_type: FI_BUS_PCI
            fi_pci_attr:
                domain_id: 0
                bus_id: 0
                device_id: 6
                function_id: 0
        fi_link_attr:
            address: EFA-fe80::39:f9ff:fe17:bdbc
            mtu: 8760
            speed: 100000000000
            state: FI_LINK_UP
            network_type: Ethernet
```

C. Evaluation Raw Data

C.1. Latency Evaluation

Message Size	RC-RDMA (IB) (usec)	SRD (EFA) (usec)	Sockets (TCP/IP) (usec)
16	1.05	20.47	35.61
32	1.09	20.5	36.78
64	1.16	20.68	36.74
128	1.2	20.75	37.57
256	1.43	20.81	37.69
512	1.49	21.02	37.1
1024	1.65	21.53	38.95
2048	1.9	22.55	39.16
4096	2.35	24.95	42.1
8192	3.1	29.99	48.37

C.2. Synchronous Bandwidth Evaluation

Message Size	RC-RDMA (IB) (MB/s)	SRD (EFA) (MB/s)
2	1.28	0.06
4	2.56	0.11
8	5.1	0.22
16	10.17	0.45
32	20.22	0.9
64	39.19	1.79
128	75.72	3.57
256	143.31	7.13
512	277.02	14.19
1024	512.06	27.89
2048	935.73	54.18
4096	1503.17	101.47
8192	2384.01	179.54

C.3. Asynchronous Bandwidth Evaluation

C.3.1. RDMA Bandwidth

Transmission Depth	16 B (MB/s)	512 B (MB/s)	4096 B (MB/s)	8192 B (MB/s)
1	10.18	277.66	1545.4	2389.29
2	16.55	460.25	2560	3884.35
4	28.68	768.36	3755.64	5440.7
8	52.37	1465.02	5750.65	7817.76
16	88.43	2492.85	7704.67	9485.96
32	109.74	3325.54	9965.03	10798.08
64	115.09	3591.18	11100.46	11048.92
128	117.95	3704.37	11098.11	10972.79
256	119.16	3748.56	10920.12	10969.6
512	118.67	3754.43	10962.89	11003.66
1024	120.36	3776.95	10945.5	11009.75
2048	120.61	3789.59	10971.65	11004
4096	120.23	3794.42	10974.89	11009.91

C.3.2. RDMA Message Rate

Million messages per second.

Transmission Depth	16 B	512 B	4096 B	8192 B
1	0.67	0.57	0.4	0.31
2	1.08	0.94	0.66	0.5
4	1.88	1.57	0.96	0.7
8	3.43	3	1.47	1
16	5.8	5.11	1.97	1.21
32	7.19	6.81	2.55	1.38
64	7.54	7.35	2.84	1.41
128	7.73	7.59	2.84	1.4
256	7.81	7.68	2.8	1.4
512	7.78	7.69	2.81	1.41
1024	7.89	7.74	2.8	1.41
2048	7.9	7.76	2.81	1.41
4096	7.88	7.77	2.81	1.41

C.3.3. EFA Bandwidth

Transmission Depth	16 B (MB/s)	512 B (MB/s)	4096 B (MB/s)	8192 B (MB/s)
1	0.45	14.19	101.5	179.27
2	0.9	28.38	203.01	358.47
4	1.79	56.57	405.12	717.49
8	3.53	111.84	803.86	1424.36
16	6.52	206.67	1503.36	2682.34
32	11.8	374.6	2743.03	4889.63
64	17.81	566.44	4250.37	7780.34
128	26.09	823.15	6041.12	10549.42
256	34.16	1068.22	8121.6	11144.85
512	36.28	1135.31	8991.27	11226.28
1024	36.05	1137.8	8962.17	11213.49
2048	35.96	1141.88	9016.46	11186.09
4096	35.8	1157.97	9084.85	11132.14

C.3.4. EFA Message Rate

Million messages per second.

Transmission Depth	16 B	512 B	4096 B	8192 B
1	0.03	0.03	0.03	0.02
2	0.06	0.06	0.05	0.05
4	0.12	0.12	0.1	0.09
8	0.23	0.23	0.21	0.18
16	0.43	0.42	0.38	0.34
32	0.77	0.77	0.7	0.63
64	1.17	1.16	1.09	1
128	1.71	1.69	1.55	1.35
256	2.24	2.19	2.08	1.43
512	2.38	2.33	2.3	1.44
1024	2.36	2.33	2.29	1.44
2048	2.36	2.34	2.31	1.43
4096	2.35	2.37	2.33	1.42

C.4. NIC Parallelism Evaluation

Million messages per second.

C.4.1. RDMA Message Rate

TX Queue Pairs	16 B	512 B	4096 B	8192 B
1	8.14	7.99	2.87	1.43
2	7.72	7.59	3.01	1.51
4	7.64	7.45	3.01	1.51
8	7.58	7.39	3.01	1.51

C.4.2. EFA Message Rate

Million messages per second.

TX Queue Pairs	16 B	512 B	4096 B	8192 B
1	1.96	1.93	1.81	1.39
2	3.37	3.25	2.72	1.47
4	3.74	3.67	2.87	1.48
8	3.74	3.68	2.89	1.49

C.5. Multi-Threaded Evaluation

C.5.1. Bandwidth

	RDMA		EFA	
Thread Count	64 B (MB/s)	4096 B (MB/s)	64 B (MB/s)	4096 B (MB/s)
1	486.57	8578.48	99	3942.32
2	624.63	11880.6	187.54	7337.51
8	2639.36	12441.22	459.14	11709.26
16	3768.9	12445.66	543.4	11847.88
32	5250.07	12468.1	528.3	11864.08

C.5.2. Message Rate

Million messages per second.

	RDMA		EFA	
Thread Count	64 B	4096 B	64 B	4096 B
1	3.24	2.13	1.37	0.96
2	4.16	2.95	2.6	1.79
4	9	2.87	4.59	2.78
8	17.59	3.09	6.37	2.85
16	25.11	3.09	7.53	2.89
32	34.98	3.1	7.31	2.89

C.5.3. EFA bandwidth scaling with thread counts

All tabular values are in (MB/sec).

Transmission Depth	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
2	5.13	10.11	19.92	39.38	72.78	124.01
10	16.53	33.07	64	115.51	188.66	283.21
50	55.01	103.89	185.37	277.84	419.14	503.39
80	69.65	138.23	232.86	339.46	477.12	531.04
100	82.47	152.62	262.03	387.5	502.84	522.05
120	88.36	155.17	288.98	400.67	525.99	529.68

C.6. Latency SRD vs UD

C.6.1. Average Latency

Message Size (Bytes)	UD (EFA) (usec)	SRD (EFA) (usec)
16	18.73	20.6
32	18.83	20.62
64	18.93	20.79
128	19.02	20.87
256	19.07	20.93
512	19.3	21.15
1024	19.78	21.65
2048	20.84	22.7
4096	23.23	25.12
8192		30.11

C.6.2. Minimum Latency

Message Size (Bytes)	UD (EFA) (usec)	SRD (EFA) (usec)
16	16.6	18.62
32	16.65	18.59
64	16.8	18.78
128	16.83	18.74
256	16.98	18.86
512	17.12	19.06
1024	17.57	19.69
2048	18.65	20.72
4096	21	22.88
8192		28.11

C.7. Latency send vs inject vs tagged

All tabular values are in micro-seconds.

Message Size (Bytes)	RDM - fi_send	RDM - fi_tsend	RDM - fi_tinject	RDM - fi_inject
8	21.5	21.57	21.53	21.34
64	21.56	21.62	21.54	21.44
512	23.76	23.85	23.79	23.75
1024	24.25	24.29	24.24	24.09

C.8. Saturated Latency - 4096 Bytes

C.8.1. Saturation latency - 50th percentile

Wait Interval (ns)	Bandwidth (MB/sec)	Latency (usec)
400000	322	26.56
150000	842	26.8
80000	1531	27.3
60000	2003	28.52
40000	2891	28.46
30000	3710	29.4
20000	5127	31.06
15000	6260	32.11
10000	7504	36
5000	10270	40.59
4000	10993	42.03
3000	11523	44.42
1000	11801	47.23
800	11799	49.68
500	11751	50.95

C.8.2. Saturation latency - 99th percentile

Wait Interval (ns)	Bandwidth (MB/sec)	Latency (usec)
400000	322	29.82
150000	842	30.99
80000	1531	31.62
60000	2003	34.02
40000	2891	33.87
30000	3710	35.08
20000	5127	39.94
15000	6260	45.26
10000	7504	58.06
5000	10270	60.41
4000	10993	59.77
3000	11523	62.62
1000	11801	67.68
800	11799	72.66
500	11751	74.4

C.9. Segmentation Latency

Message Size (Bytes)	latency (usec)		Message Size (Bytes)	latency (usec)
7000	29.34		9500	32.37
7100	29.47		9600	32.26
7200	29.58		9700	32.37
7300	29.69		9800	32.34
7400	29.81		9900	32.38
7500	29.95		10000	32.38
7600	30.14		10100	32.43
7700	30.23		10200	32.4
7800	30.4		10300	32.47
7900	30.46		10400	32.43
8000	30.59		10500	32.47
8100	30.74		10600	32.45
8200	30.81		10700	32.51
8300	30.94		10800	32.54
8400	31.05		10900	32.58
8500	31.28		11000	32.58
8600	31.29		11100	32.55
8700	31.45		11200	32.61
8800	31.59		11300	32.63
8900	32.23		11400	32.58
9000	32.28		11500	32.7
9100	32.28		11600	32.61
9200	32.26		11700	32.72
9300	32.28		11800	32.7
9400	32.32		11900	32.75

C.10. RMA Operations

C.10.1. Send vs RMA

Message Size	SEND (EFA) (MB/sec)	WRITE (EFA) (MB/sec)	READ (EFA) (MB/sec)
4	0.35	1.06	0.64
64	2.05	2.77	2
512	14.68	15.51	11.99
1024	28.67	29.72	23.15
4096	103.86	105.66	82.94
8192	184.23	185.73	149.53

C.10.2. Read vs Write

Transmission Depth	WRITE (EFA) (MB/sec)	READ (EFA) (MB/sec)
1	185.53	152.51
10	1345.84	1004.63
100	5756.73	4361.2
200	7148.3	4473.72
300	8018.64	4623.42
500	8441.79	4590.73

D. Performance Measurements and Tuning

perf and *gdb* were used extensively throughout the implementation phase to optimize the micro-benchmarks. Sample EFA *perf stat* measured for a multi-threaded test at peak throughput of 100 Gbps.

D.0.1. Send/Receive *perf* stats

Server

```
BATCH SIZE 8 THREADS X 10 => 80 : 12157.76 MB/sec
 Performance counter stats for './benchmark --flagfile=server.conf --threads=8 batch=10
     --cq_try=0.9':

         97,084.86 msec task-clock:u              #    7.354 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
         22,14,396      page-faults:u             #    0.023 M/sec
 2,94,05,93,36,794      cycles:u                  #    3.029 GHz
 4,20,29,08,60,409      instructions:u            #    1.43  insn per cycle
   99,09,95,93,223      branches:u                # 1020.752 M/sec
       3,57,62,235      branch-misses:u           #    0.04% of all branches

      13.202391328 seconds time elapsed

      89.664576000 seconds user
       7.511977000 seconds sys
```

Client

```
BATCH SIZE 8 THREADS X 10 => 80 : 12157.76 MB/sec
 Performance counter stats for './benchmark --flagfile=client.conf --threads=8 batch=10
    ':

         95,588.09 msec task-clock:u              #    6.969 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
         22,12,773      page-faults:u             #    0.023 M/sec
 3,03,77,05,35,924      cycles:u                  #    3.178 GHz
 3,51,43,97,74,286      instructions:u            #    1.16  insn per cycle
   83,49,83,31,885      branches:u                #  873.522 M/sec
       2,32,19,252      branch-misses:u           #    0.03% of all branches

      13.715541605 seconds time elapsed
```

```
     90.108032000 seconds user
      5.604705000 seconds sys
```

D.0.2. Remote memory access (RMA) *perf* stats

Emulated RMA operations showing CPU utilization on both Server and Client side.

RMA *write* Server stats

```
Performance counter stats for './benchmark --flagfile=server.conf --iterations=100000'
   :

         5,221.06 msec task-clock:u               #    0.701 CPUs utilized
                0      context-switches:u         #    0.000 K/sec
                0      cpu-migrations:u           #    0.000 K/sec
         2,77,001      page-faults:u              #    0.053 M/sec
  14,17,38,07,528      cycles:u                   #    2.715 GHz
  20,93,42,75,715      instructions:u             #    1.48  insn per cycle
   5,08,93,44,072      branches:u                 #  974.773 M/sec
         4,11,499      branch-misses:u            #    0.01% of all branches

      7.450763777 seconds time elapsed

      4.525421000 seconds user
      0.853090000 seconds sys
```

RMA *write* Client stats

```
Performance counter stats for './benchmark --flagfile=client.conf --iterations=100000'
   :

         4,305.26 msec task-clock:u               #    0.799 CPUs utilized
                0      context-switches:u         #    0.000 K/sec
                0      cpu-migrations:u           #    0.000 K/sec
         2,76,777      page-faults:u              #    0.064 M/sec
   4,75,39,62,392      cycles:u                   #    1.104 GHz
   5,84,47,20,398      instructions:u             #    1.23  insn per cycle
   1,41,58,45,006      branches:u                 #  328.864 M/sec
         1,40,651      branch-misses:u            #    0.01% of all branches

      5.386132080 seconds time elapsed

      2.649497000 seconds user
      1.720351000 seconds sys
```

D.0.3. Flame-graphs

CPU profiles and Flame graphs were generated using a combination of *gdb -ex* and *perf record*[13]. These were used to identify and optimize *libfabric* capability flags.

```
perf record -F 99 -a -g -- taskset -c 5 ./benchmark --flagfile=server.conf --
    benchmark_type=batch --payload=8192 --batch=80 --cq_try=0.8 --runtime=5

perf script | ~/FlameGraph/stackcollapse-perf.pl  > out.perf

./benchmark --flagfile=server.conf --benchmark_type=batch --payload=8192 --batch=80 --
    cq_try=0.8 --runtime=5 &
./pmp 1000 0.001 $! > benchmark.gdb


cat benchmark.gdb | ~/FlameGraph/stackcollapse-gdb.pl | ~/FlameGraph/flamegraph.pl >
    gdb.svg
```

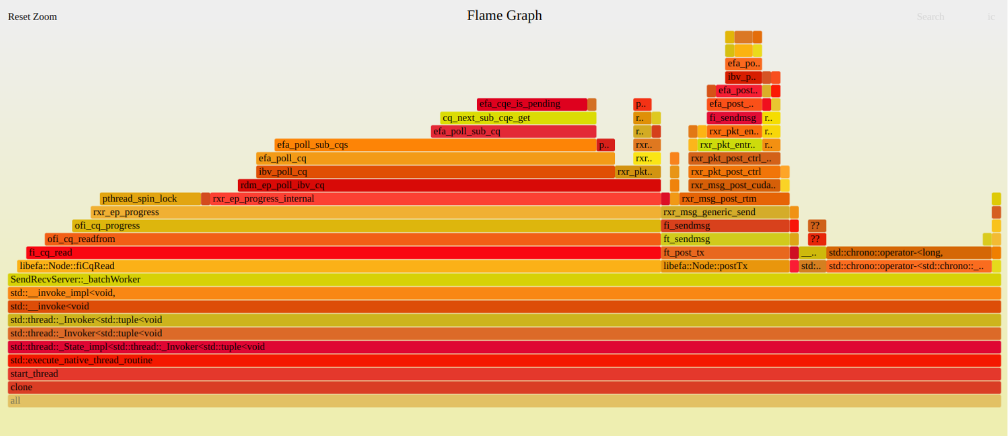


Figure D.1.: Server Flame Graph

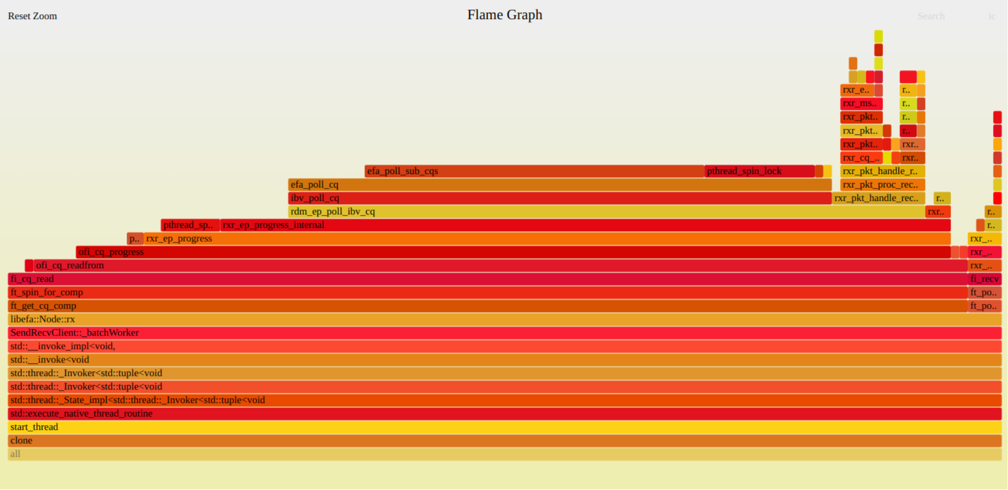


Figure D.2.: Client Flame Graph