Low-Latency Data Access in a Java-based Distributed In-Memory Key-Value Storage

Inaugural-Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von
Stefan Nothaas
geboren in
Cham

Düsseldorf, November 2019

aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Berichterstatter:

1. Prof. Dr. Michael Schöttner

2. Prof. Dr. Stefan Conrad

Tag der mündlichen Prüfung: 30. Januar 2020

Abstract

Large scale highly interactive online or batch processing offline graph applications require either low latency or high throughput for processing huge graphs with trillions of edges and billions of vertices. To keep data-access times low, systems designed for this type of big data application typically keep all data in-memory and aggregate hundreds or thousands of servers in cluster or cloud environments to create an extensive storage backend. However, highly parallel graph applications typically store and process large graphs consisting mostly of small objects less than 128 bytes. These requirements are challenging for the backend storage, the distributed processing platform, the local memory management and the network subsystem.

This thesis addresses three primary research questions in the context of a Java-based distributed in-memory key-value storage: (1) highly concurrent and distributed (graph) processing on a Java-based in-memory key-value storage; (2) a memory management in Java providing low-latency data-access and low-overhead synchronization for large graph datasets consisting of many small objects; (3) a network subsystem for highly concurrent sending and receiving of messages leveraging low latency and high-throughput network-interconnects in Java applications.

First, this thesis proposes a general compute platform and a graph processing framework for a Java-based distributed in-memory key-value storage. The compute platform builds on top of the key-value storage executing concurrent and distributed computations on storage nodes to benefit from data locality. The platform offers services to either dispatch light-weight SIMD-based computations or heavy-weight and coordination-based computations to multiple servers. The framework was evaluated with the breadth-first search algorithm (part of the Graph500 benchmark) to compare the proposed concepts to other state-of-the-art graph processing systems.

The second contribution addresses low-latency local data-access in an in-memory key-value storage in Java. It proposes a low memory- and access-overhead memory management designed for an in-memory key-value storage but also applicable in any highly parallel Java application. A custom key-value translation mechanism was extended to support low-overhead concurrent data access using a custom per-object read-write lock without considerably increasing the per-object memory overhead. The latter is kept low by a custom fixed-block allocator optimized for small objects in typical graph data-sets. The evaluation shows that our proposed solution provides an at least five-times lower memory-overhead compared to two other memory managers of other state-of-the-art Java-based key-value systems and outperforms them up to 28-fold with 128 threads on read-heavy workloads.

With InfiniBand interconnects available in HPC and cloud environments, distributed applications can highly benefit from single-digit microseconds latency and gigabytes per second throughput. The third and last contribution addresses the network with a focus on InfiniBand hardware and proposes a Java-based transport agnostic network subsystem for highly concurrent synchronous and asynchronous messaging in Java applications. This subsystem is complemented by an InfiniBand transport implementation to leverage the performance of such high-speed hardware. The evaluation shows that our solution provides high throughput and scalability on local and distributed concurrency even on worst-case all-to-all communication patterns compared to two state-of-the-art InfiniBand-based MPI implementations.

Contents

Chapter 1

Introduction

1.1 Context and Motivation

Today, **networked digital devices** are part of our everyday private and working life. Individuals use smartphones to connect to family, friends and the world, health accessories to track health data during workouts, or control their homes using networked light switches, light bulbs, and temperature controls. **Digitization** has been and is still present and ongoing in many areas of our lives. This digital transition also includes converting information of physical objects, e.g., documents, books or historical films/photography, for preservation purpose or adding sensors to devices in industrial facilities for monitoring and optimization production.

All these real-world hardware and software applications, which are continuing to increase every year, generate vast amounts of data. A study by the International Data Corporation (IDC) describes this as the **global datasphere** and shows that all devices on the world generated a total estimate of 33 Zettabytes of data in 2018 [107]. It is estimated, that this increases to 175 Zettabytes by 2025. Analyzing this enormous volume of data is highly relevant to companies and even individuals in many applications, for example: Predicting traffic, outbreaks of epidemics or interests of an audience; Analyzing customer statistics in the banking sector or student data in the education sector [1].

Applications with "large amounts of data" are typically categorized as **big data** applications. Today, the term big data is well known among the industry, the media, and even the general public to describe applications with "a lot of data". However, characterizing big data just by data volume is too vague. Additionally, one has to consider that the data volume can be too large and too complex to be processed by commodity software and hardware [117]. Hence, big data (processing) has to consider the properties of the data-set, and the actual task of processing which can be **characterized by the "3 Vs"**: *volume* refers to the amount of data stored or generated, *velocity* defines the speed of new data getting generated and *variety* describes the type of data (structured, semi-structured or unstructured). A fourth V named *veracity* extends the initial definition and describes the "uncertainty of data" regarding quality and unpredictability (e.g. weather data) [112].

Large companies or individuals have to consider these characteristics regarding their use-case when either choosing from various existing and often open-source software or developing custom solutions. One of the most well-known processing frameworks (and programming paradigms) is MapReduce [26] and the open-source implementation Hadoop [27] for scalable and straightforward big data processing (further examples follow in Chapter 2).

Powerful hardware with high storage capacity is a must for big data processing. Often, a single commodity server is not sufficient for this task, especially on large applications with up to terabytes of data to process (examples to follow). In the past, only big companies or High-Performance-Computing centers had the financial resources to afford large clusters or supercomputers [120]. Today, **cloud data centers** by Amazon [2], Microsoft [7] or Google [46], for example, are providing very large public hardware resource pools. Hence, buying and maintaining expensive hardware is not necessary anymore. Cloud providers established a rent-based business model to provide access to elastic resources for companies and individuals [47].

The field of big data includes many applications, such as the social networks Facebook [129] and Twitter [66], search engines like Google [22], simulations in bioinformatics [105] and state management in cell phone networks [110]. The social network Facebook served about 2.27 billion active users monthly in 2018 [36], the search engine Google had 50-60 billion web pages indexed in January of 2019 [125] and "The Human Connectome Project" [124] aims at studying the human brain by providing a compilation of neural data (approx. 100 trillion vertices). Further fields of applications include collecting data of customers for advertising [78], health data [58], spatial [138] and sensor data [99]. This collecting of data and its analysis has become an essential task for many companies and researchers introducing new challenges regarding hardware and software. Many of them already reached the limits of disk-based storages and started using in-memory caching-techniques [89, 137].

1.2 Requirements and Challenges

Often, a **graph-based data model** fits naturally and is applied to structure the data with vertices describing chunks of data (e.g. profiles, postings or images in a social network) and edges used to describe the relationship between one or multiple chunks (e.g. friend status or "likes" in a social network) [129, 116]. The graph-based data model can be further abstracted and implemented using a general key-value data model which is the foundation of **key-value storages**. Typically, these storages are based on hash tables for object lookup and providing basic **create, read, update and delete (CRUD) operations** for data access and modification. It is also possible to store data of a graph-based model in tables implemented by traditional and also modern databases with a relational database management system (RDBMS) data model and a structured query language (SQL) [115]. However, the benefit of natural object representation is lost by having to convert the graph-data to fit the table-based structure (see Section 2.2 for further discussion of storages).

Depending on the type of graph [116, 28], e.g. (non) directed graph or weighted/unweighted graph, various fundamental graph algorithms classes such as statistics (PageRank [101]), local clustering coefficient), traversal (breadth-first search [10]), components (e.g. weakly connected components), community detection using label propagation [133], path finding (single-source shortest paths [133]) or partitioning [128, 60] are typically used in graph applications [55, 84]. Refined versions are based on these algorithms targeting specific tasks such as (random) graph walks (e.g. forest fire [69]) or graph coloring [109]. Many of these algorithms process either larger portions of the graph or even the full graph. This leads to many challenges regarding computation and storage which are elaborated on further below.

With graph-structured data, different types of data processing can be applied. *Offline processing* requires the graph data to be loaded before a batch processing system applies one or multiple processing steps to analyze it and extract information (e.g., snapshots of social networks [129, 116] or bioinformatics [124]). A *temporal analysis* involves having multiple snapshots of the graph or parts of it. These are compared and analyzed with a focus on how the graph and its relationships evolved [70], e.g., to determine rising and falling trends in a social network. These types are typically batch-processed-based and require **high throughput to keep the overall processing times low**. In contrast, *online processing* refers to a system serving interactive user requests (e.g. social networks [89], search engines [22], state management in cell phone networks [110]) and often guarantees contracted service level agreements (SLAs) to customers. Typically, such SLAs specify that a certain percentage of requests must be processed and replied to in a defined timeframe (e.g., the user has to receive a response to 95% of all issued requests in 100 ms or less). Thus, **fast response times requiring low latency data access is mandatory** to fulfill these agreements.

Depending on the application and the number of entities (e.g., sensors, users) involved in generating data, the **data volumes can be huge and even grow exponentially over time** on live systems. On high entity/user interaction, such systems generate even billions of vertices and trillions of edges [23, 89, 6].

However, **the magnitude of objects stored are small and read access dominates the request distribution**. This essential requirement is verified by a series of workload analyses at Facebook giving valuable insight on operation distribution and object size on a large scale real-world system. A production workload shows that 70% of all objects are less than 64 bytes in size and 99% still less than 1 kB [89]. A request analysis of TAO, Facebook's geographical cache, shows that 45% of all edges had no attributes or labels attached. The remaining 65% of all edges have an average size of less than 97.8 bytes [18]. Read-operations dominate the request distribution with 99.8%. More than 50% of all vertices were less than 256 bytes in size but are still larger than edges. Another analysis of Facebook's memcachd deployment with 284 billion recorded requests further emphasizes the small request size, as well as reads, being the dominating request type [6].

The analysis of a web graph containing 3.5 billion web pages with 128.7 billion links shows that it fits into the machine with one terabyte of random access memory (RAM) further emphasizing the small average vertex and edge sizes [82].

Traditional hard drive disk-based (HDD) systems or today's flash storage technology, e.g., solid state disk (SSD), can store these vast volumes of data but are not optimized for reading and writing of small objects. However, RAM is faster and provides a 1000-times lower latency than

disk access [50]. Furthermore, traditional disks and SDDs are not optimized for **highly random access** compared to RAM which is crucial to many algorithms that are based on graph traversal, e.g., breadth-first search (BFS) [84] (also see paragraph further above). Thus, **keeping all data in-memory** is an essential requirement. However, this also requires **aggregation of resources** of multiple machines as the amount of RAM per machine is more limited and more expensive than HDD/SSD space.

Merely storing data without considering efficiency increases the per-object overhead especially with the majority of objects being tiny in size. This additional overhead increases the total amount of memory required to store all data which needs to be distributed across more machines. If a server stores fewer objects, data locality decreases per server. Storing all data requires more hardware and increases inter-server communication due to a higher degree of distribution. Thus, **efficient memory management for small objects is crucial to provide high per server data locality, cost efficiency and lower inter-server communication**.

By benefiting from data locality on single servers, algorithms can avoid having to request data stored on a remote server. With all data in-memory, *temporal* as well as *spatial locality* benefit from low access times, too. However, with increasing data volumes the degree of distribution increases as well. Naturally, the likelihood of requesting data from remote servers increases with fewer data stored locally. Instead of requesting many individual objects or a large volume of data from the remote machine, one might **consider moving the more lightweight computation to the remote** and execute on the majority of data locally. However, this solution cannot always be applied and does not guarantee to solve the problem of inter-server traffic entirely. Especially on large graphs with highly complex data dependencies, optimal data distribution and partitioning are considered NP-hard problems [38].

Relying on high locality is not feasible in general especially with algorithms randomly accessing great portions or even the whole graph (e.g., traversal) resulting in **complex all-to-all communication patterns**. This situation applies to batch processing tasks and especially to interactive applications with many users (see Facebook example above). When focusing on the inter-server communication aspects, the workload analysis above concludes that due to the majority of objects being rather small the overall **average network package size is also small on inter-server communication**. However, the application can also use batching for specific types of requests to lower the overall network message overhead [89]. Still, due to the high degree of input-freedom (e.g., by human users), a general random access pattern remains. Thus, **the network must provide high throughput on batch based processing tasks as well as/or low latency on highly interactive online applications**.

Todays still commonly used Gigabit Ethernet hardware cannot provide single-digit microsecond remote access latencies. Further evolutions of Ethernet (10, 40 and 100 Gbit/s) offer higher bandwidth with backward compatibility but are still CPU bound due to the majority of the networking stack implemented in software. However, network interconnects like InfiniBand [81] provide sub-microseconds best case latency with remote direct memory access (RDMA) capable hardware and require less CPU power by implementing the lower four layers (physical, data link, network, and transport) of the OSI (Open Systems Interconnection) stack in hardware. A particular interface called “verbs” is used to bypass the kernel and communicate directly with the hardware compared to traditional socket communication. Today, the hardware is not only available in high-performance computing (HPC) centers [120] but becomes available in cloud data centers [52].

With current multi-core hardware, a single server is already capable of running computation tasks in parallel or serving multiple interactive user requests concurrently. However, a high level of concurrency and low-latency requirement in applications requires sophisticated management of threads and hardware resources [106]. The application or system must be designed for **concurrency awareness** to utilize this hardware accordingly. Combined with distributed computing, **leveraging the power of multi-core and distributed computing together** is a very challenging task. Additionally, scalability of the system is further complicated with increasing server count.

1.3 Research Questions and Contributions

This section outlines the primary research questions for this thesis as well as summarizes its key contributions. Based on the previous Section 1.2, the following **requirements of graph-based big data Java applications**, which also resemble the primary objectives, are addressed in this thesis:

- Efficient big data processing in Java
- Storing and processing of large graphs consisting of many small objects
- Running concurrent and distributed computations with a focus on graph algorithms in a Java environment
- Low latency and scalable local data access for many small objects
- Low latency and scalable remote data access for small messages using low-latency networks

The author uses the DXRAM storage system, initially proposed by Dr. Florian Klein in his thesis [61] and presented in Section 2.5, as a foundation to address the following **primary research questions regarding the presented requirements** in this thesis. The requirements and research questions are further refined and elaborated on in the dedicated Chapters 3, 4 and 5.

1. Can an in-memory key-value storage be used as a scalable compute platform especially for graph data-sets with concurrent and distributed algorithms? (Chapter 3)
2. Can the local storage provide low-latency data access and scalability on highly concurrent local computations benefitting from data locality? (Chapter 4)
3. Can the network support graph-based applications efficiently regarding low latency and handling of many small messages even on highly concurrent random remote access? (Chapter 5)
4. Is Java a suitable environment for the questions 1.-3. stated above? (Chapters 3, 4, 5)

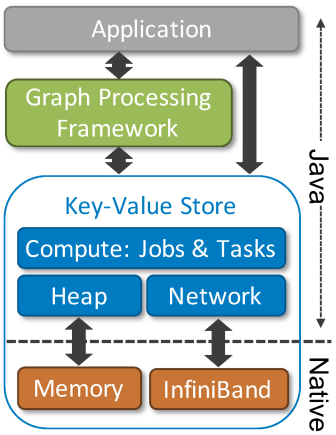

Figure 1.1: The "big picture" showing the major components relevant for the research questions of this thesis. Each of the following Chapters 3, 4 and 5 discusses one or multiple components in detail.

The fourth research question is a shared question concerning the first three. Hence, all research questions are addressed in the **context of Java big data applications**. The main system categories relevant in this context are key-value backend storages and graph processing systems. Figure 1.1 depicts the big picture outlining the architecture of the storage system with the relevant components for this thesis which are addressed by the research questions 1. to 3. in dedicated chapters.

The **major contributions** of this thesis are:

1. Analyzing and evaluating existing solutions to leverage InfiniBand in Java applications.

2. Developing and evaluating an intuitive to use network subsystem which abstracts communication primitives typically used in concurrent Java applications using currently available low-latency networking technology.

3. Development and evaluation of a highly efficient storage for many small objects providing low-latency data access for highly parallel Java applications.

4. When combining items 2. and 3., these two major building blocks greatly enhance the performance of the DXRAM system and also extend it beyond a simple storage system creating a flexible, scalable and low-latency compute-platform in Java. The results were used to build a general compute platform as well as the foundation of the graph processing platform DXGraph on top DXRAM.

1.4 Publications

International Conferences
The following publications are full papers of 10 pages each.

- **Stefan Nothaas**, Kevin Beineke and Michael Schöttner. *Leveraging InfiniBand for Highly Concurrent Messaging in Java Applications*. In Proceedings of the 18th International Symposium on Parallel and Distributed Computing (ISPDC). 2019. Copyright 2019 IEEE. `https://ieeexplore.ieee.org/document/8790899`

- Kevin Beineke, **Stefan Nothaas**, and Michael Schöttner. *Scalable Messaging for Java-based Cloud Applications*. In Proceedings of the Fourteenth International Conference on Networking and Services (ICNS). 2018. Acceptance Rate: 29%.

- Kevin Beineke, **Stefan Nothaas**, and Michael Schöttner. *Fast Parallel Recovery of Many Small In-memory Objects*. In Proceedings of the 23rd IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2017. Acceptance Rate: 32.8%.

- Kevin Beineke, **Stefan Nothaas**, and Michael Schöttner. *High Throughput Log-based Replication for Many Small In-Memory Objects*. In Proceedings of the 22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2016. Acceptance Rate: 29.9%.

Workshops at International Conferences

- **Stefan Nothaas**, Kevin Beineke and Michael Schöttner. *Optimized Memory Management for a Java-Based Distributed In-Memory System*. In Proceedings of the 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). 2019. 10 pages. Copyright 2019 IEEE. `https://ieeexplore.ieee.org/document/8752955`

- **Stefan Nothaas**, Kevin Beineke, and Michael Schöttner. *Distributed Multithreaded Breadth-First Search on Large Graphs using DXGraph*. In Proceedings of the 1st High Performance Graph Data Management and Processing workshop (HPGDMP). 2016. 8 pages. Copyright 2016 IEEE. `https://ieeexplore.ieee.org/document/7830441`

- Kevin Beineke, **Stefan Nothaas** and Michael Schöttner. *Efficient Messaging for Java Applications running in Data Centers*. In Proceedings of the 18th International Symposium on Cluster, Cloud and Grid Computing (CCGRID). 2018. 10 pages. Copyright 2018 IEEE. `https://ieeexplore.ieee.org/document/8411076`

Technical Reports

- **Stefan Nothaas**, Fabian Ruhland and Michael Schöttner. *A Benchmark Suite to Evaluate InfiniBand Solutions for Java Applications*. Published on arXiv e-prints. October 2019. arXiv:1910.02245. 10 pages.

- **Stefan Nothaas**, Kevin Beineke, Michael Schoettner. *Ibdxnet: Leveraging InfiniBand in Highly Concurrent Java Applications*. Published on arXiv e-prints. December 2018. arXiv:1812.01963. 31 pages.

- Kevin Beineke, **Stefan Nothaas**, Michael Schoettner. *DXRAM's Fault-Tolerance Mechanisms Meet High Speed I/O Devices*. Published on arXiv e-prints. July 2018. arXiv:1807.03562. 21 pages.

Journal Articles

- Kevin Beineke, **Stefan Nothaas**, Michael Schoettner. *DXNet: Scalable Messaging for Multi-Threaded Java-Applications Processing Big Data in Clouds*. Published in the International Journal on Advances in Internet Technology, Vol. 11, No. 3&4, 2018. 19 pages.

1.5 Software

In the course of this thesis, the author worked on the following major software packages. With multiple contributors of software relevant to this thesis, a detailed breakdown is given in the dedicated Sections 3.4, 4.4 and 5.4.

DXRAM is a distributed in-memory key-value storage and compute platform written in Java. It is optimized for storing many small objects (< 128 bytes) efficiently with low-latency local and remote data access. For persistence, data is replicated asynchronously to logs on SSD on remote servers. A crash-recovery failure-model is implemented to ensure data availability on hard- or software failures. The system can be used as an in-memory backend storage or an interactive compute platform. The project is open source and available at GitHub [32].
Contributors (in chronological order): Dr. Florian Klein, Michael Schoettner, Dr. Kevin Beineke, Stefan Nothaas, Michael Birkhoff, Philipp Rehs, Filip Krakowski, Burak Akguel, Christian Gesse.
Size and language(s): $\sim$ 37k lines of code; Java and C.

DXNet is an event-driven high performance messaging library for highly concurrent and distributed Java applications. It implements asynchronous and synchronous messaging primitives with a custom transparent and highly efficient serialization. An interface abstracts the underlying network and is implemented by an Ethernet-based transport (Java NIO) and InfiniBand-based transport (Ibdxnet). DXRAM uses DXNet as its network subsystem. The project is open source and available at GitHub [31].
Contributors (in chronological order): Marc Ewert, Dr. Florian Klein, Dr. Kevin Beineke, Stefan Nothaas, Filip Krakowski, Christian Gesse.
Size and language(s): $\sim$ 11k lines of code; Java.

Ibdxnet is a transport implementation for DXNet using the native C-verbs library to allow communication using InfiniBand hardware. It abstracts the verbs application programming interface (API) by implementing a dedicated subsystem with connection management and a scalable and highly optimized pipeline for low-latency and zero-copy processing of buffers (from DXNet) and provides communication either over reliable connected (RC) or unreliable datagram (UD) queue pairs (QPs). The project is open source and available at GitHub [42].
Contributors (in chronological order): Stefan Nothaas, Fabian Ruhland.
Size and language(s): $\sim$ 7k lines of code; C++.

DXMem is a memory manager for Java applications optimized for storing billions of small objects (< 128 bytes) efficiently. A custom serialization of Java objects ensures fast and low-overhead de-/serialization. Data access is optimized for high concurrency with a low-overhead read-write locking mechanism. DXRAM uses DXMem for local memory management and the implementation of the key-value-based backend storage. The project is open source and available at GitHub [30].
Contributors (in chronological order): Dr. Florian Klein, Dr. Kevin Beineke, Stefan Nothaas, Florian Hucke.
Size and language(s): $\sim$ 12k lines of code; Java.

DXGraph is a framework built on top of DXRAM providing data structures, algorithms, and utilities for graph processing on DXRAM. It utilizes DXRAM's API to implement re-usable tasks for loading and processing of graphs including an implementation of the BFS algorithm according to the specifications of the Graph500 benchmark. The project is open source and available at GitHub [41]. Contributors (in chronological order): Stefan Nothaas, Philipp Rehs.
Size and language(s): $\sim$ 5k lines of code; Java.

cdepl is a Bash-script-based framework to simplify the deployment of distributed applications to different types of cluster environments. It abstracts the underlying cluster environment for the applications to allow transparent deployment to different cluster setups and applications abstract tasks like configuration or starting of instances. cdepl supports DXRAM, a collection of other systems and different kinds of benchmarks which were used for the evaluations in the course of this thesis. The project is open source and available at GitHub [20].
Contributors (in chronological order): Stefan Nothaas, Kevin Beineke, Fabian Ruhland, Filip Krakowski.
Size and language(s): $\sim$ 1k lines of code; Bash.

1.6 Organization of the Thesis

The introductory Chapter 1 presents the context and motiviation for this thesis (Section 1.1), the requirements and challenges (Section 1.2), the research questions and contributions of this thesis (Section 1.3) and the publications (Section 1.4) as well as published software (Section 1.5) the author of this thesis contributed to. Afterwards, background information is given in Chapter 2 with a focus on the Java environment (Section 2.1), relevant systems and technologies regarding in-memory key-value storages (Section 2.2), graph processing systems (Section 2.3) and high speed interconnects (Section 2.4). Section 2.5 introduces the DXRAM storage system which is used for development in this thesis.

Dedicated chapters address the three primary research question (stated in Section 1.3) with each chapter discussing one or multiple components depicted in the big picture presented in Figure 1.1. Each chapter further refines one primary research question and presents the respective contributions in detail. Chapter 3 addresses the computation requirements by presenting an additional computation component added to the core of DXRAM. DXGraph adds a layer for graph-based applications to DXRAM. Chapter 4 presents the redesign of DXMem, DXRAM's memory management, with a focus on highly concurrent Java applications. Chapter 5 is dedicated towards the network subsystem DXNet, used by DXRAM, for highly concurrent

messaging in Java applications. This chapter also includes the evaluation of available solutions to leverage InfiniBand in Java applications and the native library Ibdxnet to support InfiniBand hardware with DXNet.

Chapter 6 lists further contributions of this thesis which are not directly addressing the primary research questions. Conclusions are presented in Chapter 7 including the achievements of this thesis (Section 7.1), lessons learned (Section 7.2) and future directions (Section 7.3).

Chapter 2

Background and Overview

This chapter presents relevant background information regarding the Java environment (Section 2.1) and communication using high-speed interconnects (Section 2.4), as well as an overview of existing and to this thesis relevant storage systems (Section 2.2) and graph processing systems (Section 2.3). A detailed discussion and evaluation of selected systems is given in dedicated publications in this thesis (see Chapters 3, 4 and 5). Section 2.5 presents an overview of the DXRAM storage system with its key features. As part of the contributions of this thesis, separate chapters with dedicated publications describe selected features in detail. Systems or fields of research that are related but not relevant to this thesis are just mentioned briefly.

2.1 The Java Environment

This thesis focuses on dedicated graph processing systems as well as in-memory key-value based storages/caches for, but not limited to, graph-based applications. Java-based systems are of particular interest as **Java is widely used in the field of big data** for batch processing [27, 135], stream processing [131, 135], key-value caches [48, 53, 96] data grids [39] and graph processing [132].

Today, the Java language and environment can be considered very mature and offers beneficial features such as automatic garbage collection, static type-safety, robust exception handling and a rich standard library including concurrency support. Especially classes and interfaces like Unsafe [80], the Java Native Interface (JNI) [72] or the "New IO" library with ByteBuffers and NIO networking [98] provide means for low-latency I/O operations (e.g., memory, network). However, some of these advantages are often considered performance disadvantages which lead to the common misconception that Java applications are always slower than applications written in traditional native-compiled languages such as C and C++ [119]. These languages are also used when developing large big data systems [100, 115, 118, 19, 21, 86, 76].

However, with many optimizations over the years, e.g., run-time compilation, garbage collection, Java can no longer be considered a slow language/environment in general anymore [102]. This thesis shows in different publications that Java is indeed a suitable language for developing such systems, also when compared to systems implemented in C and C++ (see Chapters 4 and 5).

2.2 Key-Value Cache/Storage Systems

With RDBMSs being developed and used since the 70's [24], these systems or their original design are often used, even today, in big data applications [115, 130, 22] for storing or caching data but are not the focus of this thesis. Such traditional data stores often implement a column/table-based or document-based design which is not always optimal for this thesis's target applications and data-sets.

The **key-value datamodel** allows storing data as tuples consisting of a unique key identifying a value (e.g., binary data) [136]. More complex data models (e.g., graphs) are typically built on top of this basic model which is implemented by **key-value stores and caches**. The main difference of a storage and a cache lies in the method of how data is persisted and recovered in case of storage server failure. Often, *caches* require external sources to re-fill them after a crash while *storages* provide built-in solutions, e.g., crash-recovery. However, this particular feature is not the main focus of this thesis. Hence, the author does not explicitly differ caches from storages and vice versa after this section. Further features offered by these systems are available on a variety of implementations (both caches and storages).

Pure key-value stores often limit the set of available operations to **create, read, update and delete (CRUD)** [79] instead of implementing SQL based query language [68]. Some backend stores implement transactions instead of a BASE consistency model (basic availability, soft-state, and eventual consistency). The latter is more commonly implemented as it favors scalability over consistency [104]. This super-set of systems is typically described as NoSQL systems which also includes key-value stores and caches [75]. The design of the back-end storage of these systems is still relevant to this thesis, but particular features like the query language are not.

This thesis focuses on systems keeping **all data in-memory** to ensure low-latency data access for highly interactive applications. With commodity servers having limited RAM and CPU resources, **distributed systems** typically provide scalability by aggregating many servers. Additional backup mechanisms storing data and modifications on disk for consistency and the recovery to handle server failures are often implemented on such storage systems but not further discussed in this thesis.

In-memory caches are typically used to cache hot data from slow disk or disk-based systems (e.g. databases), running in the same data-center, in RAM and provide lower-access latency, especially on read-intensive workloads [129]. Multiple cache instances are aggregated to form a cluster offering large amounts of fast memory for data intensive applications. Often, these systems also support running computations on their instances lowering access times to locally already available data and utilizing often unused CPU resources. Selected and relevant systems for this thesis are *memcachd* [37] and *TxCache* [103] which are implemented in C, and *Ehcache* [33], *Hazelcast* [48], *Ignite* [96] and *Infinispan* [53] which are implemented in Java. All these systems are limited to Ethernet-based networks and do not support newer fast networks like InfiniBand (see Section 2.4).

In-memory storages extend the concept of caches by adding mechanisms for data persistence to handle server failures and to avoid data loss (see the previous paragraph). Selected and relevant systems for this thesis are *Alluxio* [71] and *Redis* [19] which supported Ethernet

networks. However, when scaling the storage to hundreds or even thousands of servers, large data-sets are unavoidably scattered. This process increases the **inter-server communication overhead in volume (number of requests) and range (number of connections to remote servers)** resulting in increased overall latency. Systems have been proposed that are specifically designed to address this network bottleneck using, often exclusively, high-speed networks such as InfiniBand. This includes systems such as *Apache Crail (Incubating)* [3], *FaRM* [29], *FASTER* [21], *HERD* [59], *MICA* [74] and *RAMCloud* [100].

In-memory NoSQL databases extend the concept of in-memory storages by adding features such as, but not limited to, a query language (often similar to or partially based on SQL), an advanced consistency model or message queues for change notifications. This includes systems such as *Aerospike* (formely known as Citrusleaf) [118], *Apache Geode* [39] and *Megastore* [8] (according to the authors a mix of RDBMS and NoSQL).

2.3 Graph Processing Systems

Graph processing systems are either built on top of a backend key-value storage, implement their own or rely on remote servers providing storage. For processing graph-based data-sets, **the backend storage must be capable and optimized for handling such structured data-sets** which is one of the key-objectives regarding local and also remote performance. Thus, the design of various existing key-value storages/caches, especially their memory management, is of high interest to this thesis. Systems offering features beyond a basic backend storage, such as NoSQL stores (see Section 2.2), are also of interest to this thesis but mainly regarding their backend storage design.

Graph databases like *InfiniteGraph* [54], *Neo4j* [87] or *Titan* [127] are optimized for storing graph-structured data on dedicated database instances and allow queries to the stored data by external applications. These are of partial interest to this thesis regarding their data model and storage implementation, but not their typical database features such as the query language.

Graph processing platforms are either implemented on top of a backend-storage or as external applications connecting to dedicated graph databases [28]. Typically, these distributed platforms either implement an offline analytics platform (e.g. based on batch processing) or an endpoint to serve interactive requests for processing queries of web applications. Systems implementing a key-value data model often provide a vertex-centric programming model for implementing graph algorithms. Selected examples include *Giraph* [4] which is based on Pregel and implemented on top of *Hadoop*'s MapReduce framework [27], Google's *Pregel* [77], *GraphX* [132] built on top of *Spark*'s data parallel framework [135], Microsoft's *Graph Engine* [114] (formerly known as Trinity) and *Turi* (formerly known as GraphLab) [76]. A more generic computational approach is taken by the shared memory system *Grappa* [86] which does not enforce a specific programming model for graph processing by default.

Graph processing libraries providing efficient and optimized implementations of commonly used graph algorithms are not of interest to this thesis because they do not address concurrency, synchronization, consistency and distribution aspects. This also applies to temporal graph analysis focusing on the evolution of a graph over time by analyzing time-based snapshots.

2.4 Communication using High-Speed Interconnects

On a large scale with data distributed to many servers, **data locality degrades and network latency becomes the dominating factor** for overall application access latency. With high-speed interconnects such as InfiniBand, Intel Omnipath or High-Speed Ethernet, this next generation of network technology is already well known for years in the field of HPC [120] but also becoming available in public clouds today [52]. This technology offers significant advantages for large batch processing tasks as well as highly interactive always online applications. Applications can utilize RDMA operations to read/write data from/to remote machines without involving the remote's CPU. For messaging/signal/event-driven applications, a more traditional form of data exchange by using messaging verbs is also available and often preferred. The protocol includes a full kernel bypass on the local system directly communicating with the host channel adapter (HCA) from userspace.

This thesis addresses high-speed communication concerns using InfiniBand with a particular focus on Java applications.

InfiniBand in Java. The available solutions to leverage InfiniBand in Java applications are limited at the time of writing this thesis. Existing systems can utilize *IP over InfiniBand (IPoIB)* [56], *JSOR* [126], *libvma* [73] or the *Sockets Direct Protocol (SDP)* [45] to transparently redirect socket-based traffic over InfiniBand (which is not just limited to Java applications). To program the RDMA-capable hardware directly, an implementation of the "verbs" API must be used which is implemented by the native *C-verbs* [97] and Java *jVerbs* [121] libraries. With these solutions, existing applications with socket-based networks stacks can be accelerated or new software stacks can be designed to further benefit from InfiniBand hardware, e.g. accelerating Redis [123], memcached [57], Spark [5] or Apache Kafka [49] with InfiniBand.

MPI (Java) and HPC. Special InfiniBand implementations or wrappers of the traditional, and in HPC well known, Message Passing Standard (MPI) are used for big data processing providing a network stack with abstracted communication primitives, e.g. *FastMPJ* [35], *mpiJava* [83], *MVAPICH2* [85], *Open MPI* [95]. But, this thesis is not focusing on message passing or HPC and does not further discuss general HPC frameworks/libraries like UCX [113] which also abstract high-speed networks stacks for distributed computing.

Systems designed for high-speed networks. To leverage the true potential of InfiniBand hardware, new systems have been proposed by the industry and the science community. Focusing on key-value storage and graph processing system in this thesis, systems like *RAMCloud*, [100], *FaRM* [29], *MICA* [74], *HERD* [59] or *Apache Crail* [3] were designed and developed with a focus on high-speed networking hardware. All systems are implemented in C or C++ except Apache Crail which is implemented in Java.

2.5 The DXRAM Storage System

This section presents an overview of the current state of the DXRAM storage system with its major key features and achievements.

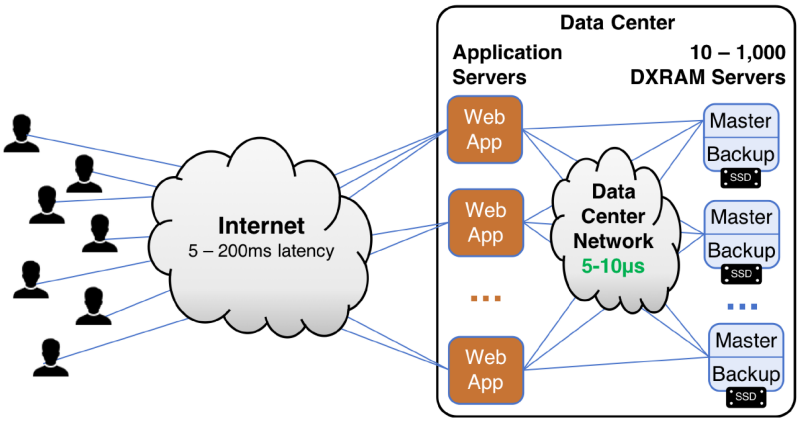

Figure 2.1: Application use-case when using DXRAM as a pure backend storage.

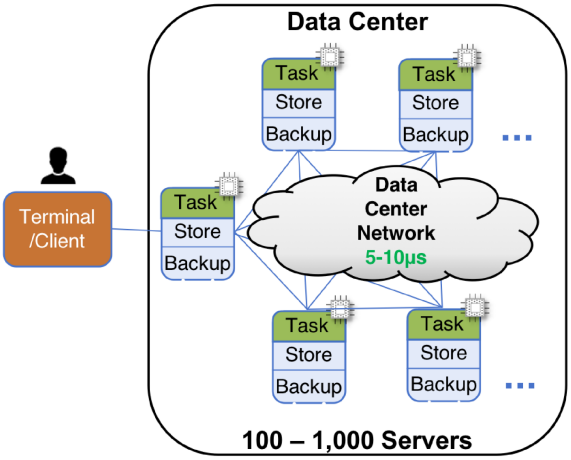

Figure 2.2: Application use-case when using DXRAM as a compute platform.

DXRAM is a Java-based distributed in-memory key-value store and compute platform for low-latency data-center/cloud (graph-based) applications such as, but not limited to, social networks, search engines or general I/O-bound long-running scientific computations. Figure 2.1 depicts a typical application use-case where DXRAM is used as pure backend storage in a data center. A low-latency network connects the storage servers of DXRAM within the data-center with web applications on dedicated servers issuing requests to the backend storage. Figure 2.2 depicts another use-case scenario where DXRAM is used as a compute platform. Compute

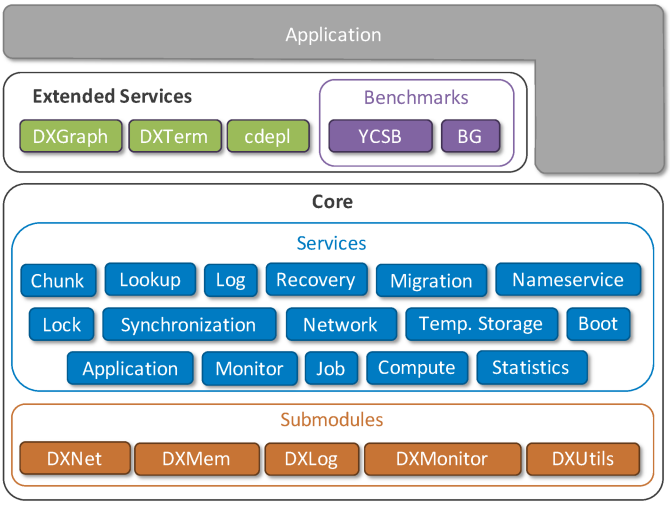

Figure 2.3: Overview of DXRAM's layered architecture [11].

tasks, submitted by the user(s), are running on DXRAM servers with access to the backend storage in the data center. The user can submit these tasks to the system, using a terminal or client application. Afterward, the system deploys the submitted tasks to one or multiple servers. DXRAM is open source and available at Github [32].

The following paragraphs present the primary features and contributions of the DXRAM storage system which are part of its layered architecture depicted in Figure 2.3.

Flexible data model and data structures [64, 61]. DXRAM stores data as key-value tuples called **chunks**. The key is a 64-bit unique ID, named **chunkID (CID)**, and the value is the binary serialized data of a Java object. DXRAM's custom de-/serialization allows fast de-/serialization of complex and nested objects to the backend storage or network buffers for remote transfers. More complex data structures, e.g., dynamic lists, tables or graph data (vertices, edges), can be implemented as DXRAM chunks. In his master thesis, Kai Neyenhuys [88] implemented a distributed index based on a B-Tree structure. In the course of a project, Ruslan Curbanov developed DXDdl, a data-description language for blueprinting DXRAM data structures [40]. At the time of writing this thesis, further implementations of data structures like a dynamic list or hash map are currently being worked on by students.

Distributed and transparent object lookup, and scalable meta-data manamagement [64, 62, 61]. DXRAM assigns different tasks to two different server roles: Peer and Superpeer. **Peer**s, in general, are storage and compute instances with different capabilities like local key-value chunk storage, running computations on peers (e.g., jobs, tasks or applications) or chunk backup to SSD. A peer without any of these capabilities is a client with remote access to the DXRAM infrastructure. **Superpeer**s store global metadata like chunk locations, provide a monitoring facility, detect server failures and coordinate the recovery of failed storage peers, and provide a naming service.

Every server in DXRAM is identified by a unique 16-bit **node ID (NID)** assigned on startup. Superpeers are arranged in a Chord-like structure adapted to the data-center environment. General inter-server network traffic between nodes is not routed using the overlay. Instead, all nodes (server role independent) keep a list of available servers to allow direct connections to remotes on demand. The overlay is used for assigning metadata management responsibilities, to detect failures and recovery coordination. For reliability, superpeers replicate their data to a configurable number of succeeding superpeers in the overlay (default three).

The system identifies chunks and their location by a 64-bit globally unique ID split into a 16-bit NID of the creator node and 48-bit locally unique sequential number (LID). Thus, the initial location of every chunk is known apriori. The sequential LID allows storing the location of multiple chunks as space efficient ranges instead of single IDs. As the location might change, e.g., recovery of a failed storage server or migration of hot data, superpeers store the locations in a modified B-Tree for fast lookup which is optimized for storing CIDs ranges efficiently.

Memory management for billions of small objects with low latency on highly concurrent access [63, 61, 93]. DXMem provides local memory management enabling highly efficient storage for many small objects and low-latency data access with low-overhead synchronization mechanisms for concurrent Java applications. DXMem's tailored allocator stores the binary serialized data of chunks outside of the Java heap with marginal per-object memory footprint. A custom paging-like address translation ensures fast and memory efficient translation of CIDs to the corresponding native memory address for local chunk lookup.

Low latency and high throughput network subsystem for highly concurrent applications [12, 17, 90, 92, 108, 94]. DXRAM uses DXNet as its network subsystem. DXNet provides low-latency and high-throughput messaging with a modular transport layer. It implements higher-level messaging primitives abstracting typical asynchronous and synchronous messaging patterns. With lock-free data structures, fast concurrent serialization, zero-copy, and zero-allocation, it is optimized for highly-concurrent Java applications. DXNet currently supports InfiniBand networks using Ibdxnet and Ethernet networks using Java NIO.

Fast Asynchronous logging to SSD and crash-recovery failure-model [14, 15, 13, 16]. To ensure reliability and avoid data loss, DXRAM scatters chunks of one storage server (backup source) to one or multiple remote servers (backup destination). The type of distribution, e.g., random, disjunctive or location-aware, is configurable. A backup source can also be a backup destination for remote storage servers at the same time. The system stores all incoming backup-data in a log on disk (SSD or HDD) on the destination. This log is optimized for high throughput for many small objects. A two-level logging mechanism ensures fast persistency and speeds up recovery in case of server failure.

Storage monitoring and data migrations to handle hot spots [65]. Superpeers are monitoring their corresponding peers by gathering common metrics such as CPU, memory or network load periodically. This monitoring facility enables detection of different types of hotspots regarding storage (low memory), computing resources (high CPU load) or network requests (high traffic). The superpeer can detect such hotspots by analyzing the data and executing measures, e.g., data migration of (subsets of) hot data. Thus, a hotspot can either be moved entirely to another peer or split to multiple peers to distribute the overall load. Computations or Applications running on DXRAM can use its data migration mechanism to execute a more contextual based and precise load balancing.

Computations on storage servers benefiting from data locality and multi-core hardware [91]. DXRAM provides multiple services to execute parallel and distributed computations on storage servers. The *Job Service* enables running lightweight and short computation methods on peers using a work-stealing approach with a fixed size worker pool. Jobs can be delegated to remote servers to enhance data locality improving the overall performance. The *Master-Slave Service* provides an infrastructure to run tasks distributed to multiple servers. A compute group, which consists of multiple servers (configurable) and is managed by the service, executes each task. A task script chains multiple tasks with implicit synchronization (super-step) between tasks. To avoid data races and provide synchronization mechanisms for computations, DXMem implements a per chunk read-write lock for data synchronization and locking. Furthermore, DXRAM implements a distributed barrier to enable super-step synchronization for distributed computations on peers.

An interface allows developers to create custom applications packed as jar-packages called **DXApp**s and execute them on DXRAM peers. Compared to the built-in computational infrastructure, this gives developers a higher degree of freedom to develop and run custom distributed and concurrent applications on DXRAM. Applications have full access to the DXRAM API just like deployed tasks and jobs.

DXGraph - Graph Processing on DXRAM [91]. DXGraph is a framework building on top of DXRAM providing several utilities such as data structures or algorithms implemented as tasks, jobs or applications to enable graph processing with DXRAM. It also includes a distributed and concurrent implementation of the Graph500 benchmark [84] implemented using DXRAM tasks.

Benchmarks. Several built-in and external benchmarks allow testing and analyzing the performance of DXRAM or its subsystems. DXNet [31] with its built-in benchmark can be executed independently of DXRAM to evaluate the pure network performance of a selected transport using a variety of configurable parameters. DXMem [30] also provides a built-in and DXRAM independent benchmark, similar to the Yahoo! Cloud Serving Benchmark, which can be used to determine the performance of the memory management using different workloads.

The *Yahoo! Cloud Serving Benchmark (YCSB)* [25] is a benchmark to evaluate different (in-memory) storage systems using workloads of common cloud/online services with a DXRAM client available. The *BG Benchmark* [9] evaluates data storages with a focus on social networking actions and sessions and a DXRAM client available for evaluation. The *LDBC graphalytics benchmark* [55] is designed specifically for benchmarking systems with graph analytics and processing workloads. At the time of writing this thesis, Ruslan Curbanov is working on supporting DXRAM with this benchmark.

Chapter 3

Using an In-Memory Key-Value Store as a Compute Platform for Java Graph Applications

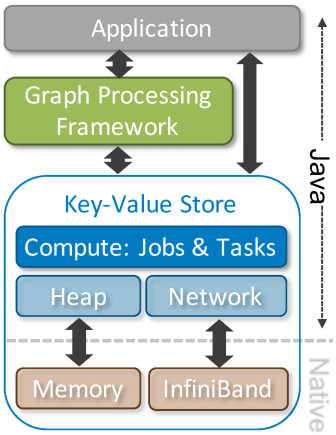

Figure 3.1: The "big picture" of this thesis with the relevant components for this chapter highlighted ("Application" layer and interface, "Graph Processing", "Compute", and interfaces).

This chapter discusses the first research question "Can an in-memory key-value storage be used as a scalable compute platform especially for graph data-sets with concurrent and distributed algorithms?" in a Java environment (see Section 1.3).

First, in Section 3.1, the author presents the addressed requirements concerning an in-memory key-value storage that result from this major question based on the previously introduced context (see Section 1.1) and its challenges (see Section 1.2). Based on the stage of work of the DXRAM storage system presented in Section 3.2, Section 3.3 elaborates on the major research question in detail. Section 3.4 presents the resulting contributions of this work followed by a copy of the publication:

Stefan Nothaas, Kevin Beineke, and Michael Schöttner. "Distributed Multithreaded Breadth-First Search on Large Graphs using DXGraph". In Proceedings of the 1st High Performance Graph Data Management and Processing workshop (HPGDMP). 2016. 8 pages.

3.1 Requirements

Two categories of requirements regarding graph processing on an in-memory key-value storage must be considered: First and foremost, running distributed and parallel computations on the storage in general. Second, providing an extra layer that extends the system and its key-value data-model beyond the basic key-value foundation for graph-based applications. These requirements have to be considered in a Java environment.

Presented in Chapter 1, graph-applications process and store huge graphs consisting of many small objects. Many algorithms create worst-case access patterns resulting in highly random access to the data. Thus, the computations should run as close as possible to the stored data to benefit from locality and to lower expensive communication with remote servers. The system must support this by running parallel computations locally on storage servers. Furthermore, with multi-core resources common today, concurrency must be considered to be able to exploit these resources. The Java environment already provides utilities for synchronization of data races on a single server. However, with the vast amounts of data, a distributed approach becomes inevitable making concurrency control across multiple servers necessary. To avoid data races and synchronize between computation steps, the system has to provide utilities for managing distributed concurrency as well.

Graph-based applications have to store their data using the back-end storage of the system. Storing the data requires appropriate data structures that go along well with the natural graph representation. For offline processing systems, the data is typically loaded from files stored on disk, first. Loading real-world modeled workloads [116] and pre-generated synthetic data is essential as well for testing and benchmarking. The latter can be generated by graph-generators, e.g., the Kronecker generator of the Graph500 [84] reference implementation. These graphs allow testing of arbitrary small and large scales which are not covered by the real-world data-sets, e.g., to evaluate the system's limits regarding storage capabilities.

Graph algorithms, especially traversal-based, generate highly random access patterns resulting in complex inter-server all-to-all communication patterns. Thus, keeping as much locality as possible and lowering remote communication overhead for such algorithms is essential for performance. The Graph500 [84] is an established graph benchmark to evaluate this worst-case and fundamental system requirement by implementing a breadth-first search. This algorithm is one of the commonly used foundations for traversal-based algorithms [55] and can be implemented with local and distributed concurrency.

3.2 Stage of Work

To address the requirements proposed in Section 3.1, the author conducts his research with the Java-based DXRAM in-memory key-value storage system. Initially, Dr. Florian Klein designed DXRAM as a pure backend key-value storage [61]. This section addresses only the relevant aspects of the DXRAM storage for this chapter. Features such as monitoring, backup and recovery are omitted here.

The DXRAM system already implemented an overlay structure forming a cluster of servers for a distributed storage. A per storage server local memory manager implemented the in-memory key-value-based storage optimized for storing many small objects efficiently. However, the memory management was lacking de-/serialization of Java objects and could handle raw binary data (Java byte-arrays), only. Distributed exchange of data between servers was possible using the Ethernet-based network subsystem supporting asynchronous and synchronous messaging primitives. The DXRAM client and API supported basic CRUD operations to access and modify binary data stored on remote storage servers. Multi-CRUD operations allowed batching of multiple chunks per operation, e.g., multi-get to get multiple chunks by invoking a single get-operation.

3.3 Research Questions

With the DXRAM storage providing general very low per-object metadata overhead, it was already optimized for storing many small objects found in typical graph data-sets. However, with data access exclusively using a remote client API, various benefits of the system and the hardware have not been exploited thus far. Batching creates some locality but does not fully exploit locality on random access patterns of graph algorithms. This state of work leads to some crucial questions that have to be considered when developing a suitable solution for graph-based applications: How can the DXRAM storage system be extended to support local computations on storage servers? However, considering the various fundamental graph-based algorithms, what's the impact on the system? Is this impact limited to the storage, only or does it affect other subsystems as well? How to leverage the power of the available multi-core hardware regarding local and especially distributed concurrency? What requirements are imposed by these algorithms and how can the system address these in general and not just for one specific algorithm?

Further questions arise when switching to the point-of-view of the graph application: How to represent graph-data adequately for the application and allow efficient processing by the system? With tasks required by many graph-applications such as loading/generating data or executing commonly used graph algorithms, is it possible to create a high-level abstraction that can be re-used by many applications without impacting the system's performance significantly?

3.4 Contributions

Our publication proposes a design of two core modules extending DXRAM to run computations on storage servers as well as the graph processing framework DXGraph including graph data loading, and a distributed and multi-threaded breadth-first-search reference implementation according to the Graph500 specification [84]. The contributions stated below which are not explicitly assigned to any author/contributor are by the author of this thesis.

The *JobService* compute module allows applications to submit and run lightweight jobs with a fixed size thread pool on DXRAM storage servers. This approach is suitable for rather short computations that run single threads on multiple data subsets (SIMD principle). For more complex computations that have to run the algorithm concurrently on a single server and distributed across multiple servers, the application can submit tasks to the *TaskService*. It provides mechanisms for coordination and synchronization of concurrent and distributed compute tasks running on one or multiple DXRAM storage servers and are suitable for massively parallel computations spanning large datasets.

DXGraph provides data structures for storing graphs using a natural representation with vertices and edges on DXRAM storage servers. It uses the new compute modules to implement typical tools required for graph processing, e.g. (distributed) loading of datasets or generating synthetic data for benchmarks, and algorithms which are commonly used in graph applications.

The evaluation shows that our implementation of the BFS algorithm with DXRAM and DXGraph is up to five times faster compared to GraphLab's and Grappa's, two state-of-the-art C++-based systems.

With DXRAM storing tiny objects efficiently, further research in the graph application domain was initiated by Dr. Florian Klein who created the initial implementation of the DXRAM storage system. To bring DXRAM to this next stage, the author of this thesis started analyzing the graph-processing application domain to determine the requirements by the applications of this field.

The author of this thesis implemented the two core services *JobService* and *TaskService* to enable running computations on DXRAM storage servers. Furthermore, this was preceded by a large and complex refactoring phase of the whole DXRAM system, as DXRAM was designed as a pure backend storage, initially. The refactoring was a close collaboration between Dr. Kevin Beineke and Stefan Nothaas. Dr. Kevin Beineke refactored DXRAM's bootstrapping with ZooKeeper, the DXRAM overlay, logging, and network subsystem to integrate into the new foundation. Stefan Nothaas created a new core for the DXRAM system to meet the requirements of the target application domain and provide future extensibility of the system. Furthermore, he was involved with adapting the local memory management as well as the network subsystem to use DXRAM's custom de-/serialization interface for chunk objects (see Chapters 4 and 5).

Initially, the author started developing DXGraph including the highly optimized BFS implementation, loading of graph data from an ordered edge list file format and basic graph data structures. In October 2016, the project lead on DXGraph was passed to Philipp Helo Rehs starting work on his Ph.D. thesis with a focus on graph applications. Philipp Helo Rehs added further tasks for loading different graph file formats, new data structures and implement the Bron-Kerbosch algorithm.

Dr. Kevin Beineke, Prof. Dr. Michael Schöttner, and Philipp Helo Rehs took part in many discussions about the design and performance analysis of DXGraph and DXRAM.

Stefan Nothaas wrote the paper and evaluated all the systems presented in it. Dr. Kevin Beineke and Prof. Dr. Michael Schöttner reviewed the paper several times and helped improve it.

With the ongoing development of DXGraph, DXRAM showed significant deficits in various graph-based workloads regarding concurrency in local memory management access and remote data access latency. Thus, the research focus was shifted to local memory management and InfiniBand development to address these challenges (see Chapters 4 and 5).

Distributed Multithreaded Breadth-First Search on Large Graphs using DXGraph

Stefan Nothaas, Kevin Beineke and Michael Schöttner

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
E-Mail: stefan.nothaas@uni-duesseldorf.de

***Abstract*—Interactive graph applications are often generating irregular access patterns on very large graphs with trillions of edges and billions of vertices. In order to provide short response times for interactive queries, all these small data objects need to be stored in memory. DXRAM is a distributed in-memory system optimized to efficiently manage large amounts of small data objects. In this paper, we present DXGraph, an extension to allow graph processing on DXRAM storage nodes. For a natural graph representation, each vertex is stored as an object. We describe DXGraph's implementation of a breadth-first search (BFS) algorithm, as specified by the Graph500 benchmark. The preliminary evaluation of the BFS algorithm shows that DXGraph's implementation is up to five times faster than Grappa's and GraphLab's with a peak throughput of over 323 million traversed edges per second.**

***Index Terms*—Graph processing; Breadth-first search; Big data; Cluster computing; In-memory storage**

I. Introduction

Offline and online graph analytics need to process very large graphs with up to billions of vertices connected by trillions of edges. Interactive applications like social networks demand high performance and low-latency storage solutions to ensure fast response times for queries of potentially many interactive users. Facebook is already storing billions of small, less than 64 byte, objects resulting in a graph with trillions of edges [1]. Other graph examples are brain simulations with billions of neurons and thousands of connections each [2] or search engines for billions of indexed web pages [3].

Typically, databases and in-memory storages cannot handle small data objects efficiently and introduce a considerable large meta-data overhead on a per object basis. Therefore, it is often recommended to aggregate vertices and edges for queries which is impacting latency and burdening the developer. Holding all objects always in RAM reduces access latency dramatically but the huge amounts of small objects require an efficient memory management and fault tolerance to mask node failures.

DXRAM is a distributed in-memory storage system designed to efficiently store and handle many small data objects. This is achieved by a minimal meta-data overhead, scalability regarding number of storage nodes and high throughput for remote and local client requests. DXRAM is designed to run within a single data center, currently supporting Gigabit Ethernet (Infiniband planned).

The main contributions of this paper are:

- DXGraph and DXCompute: Data structures and tasks for loading, generation and processing of graphs with either lightweight jobs or master-slave coordinated tasks as computations on DXRAM.
- Direction optimized BFS implementation defined by the second Graph500 [2] kernel with highly efficient data structures.

The structure of the paper is as follows. Related work is discussed in section II, followed by an architectural overview of the DXRAM core in section III. Section IV describes the typical steps involved with graph processing using the breadth-first search as an example. Section V describes the implementation in DXGraph. Section VI presents the experimental results followed by conclusions and an outlook on future work in the last section VII.

II. Related Work

Many systems have been proposed to provide low-latency data access for online graph queries and offline graph analytics. Google's **Pregel** [4] introduced a new vertex centric computation model based on message passing for distributed offline graph processing. Each vertex receives messages and executes modifications on its own data with fault tolerance achieved through a checkpointing mechanism. DXGraph and other graph systems share some characteristics with Pregel, especially the vertex centric approach. However, DXGraph is not sending computations to vertices compared to Pregel. Furthermore, Pregel is not a key-value store and is targeting offline processing. DXRAM provides fault tolerance through a logging based approach instead of checkpointing. As an open source counterpart to Pregel, **Giraph** [1] uses Hadoop as a foundation for graph processing building on its existing MapReduce framework.

GraphLab [5] is an offline distributed in-memory processing framework for graphs. Also based on a vertex centric execution model and fault tolerance through a checkpointing mechanism, data is represented in a vertex centric manner. It is designed for machine learning and graph based applications with an API based on a three phase gather-apply-scatter approach. Input graph data is represented as user modifiable

HPGDMP16; Salt Lake City, Utah, USA; November 2016
978-1-5090-3880-0/16/$31.00

program state for each vertex. An update function executes the user's stateless computations on the data by transforming it within the scope of a a single vertex. The sync operation aggregates the results per vertex. Again, DXGraph shares the natural data representation and an execution phase provides a similar approach to the update function of GraphLab. However, a separate sync operation to aggregate results is not forced on the programmer. Depending on his application, he is free to choose the paradigm fitting his use case. For fault tolerance and persistence, logging is used for DXRAM and checkpointing for GraphLab.

With **GraphX** [6] utilizing Spark's data parallel framework for distributed graph computations, graphs are stored as tabular data instead of objects in a key-value store. Operations on the data are defined as transformations on the immutable graph yielding a new graph. Online graph analytics are enabled with interactive queries like load, transform and compute. Moreover, instead of creating backups of the altered data, fault tolerance is achieved by maintaining the operations to transform the base data. This approach is very different from DXGraph's and also many other systems of this category.

Microsoft's graph engine **Trinity** [7] introduces its Trininty Specification Language to define data schemata and to use the message passing protocol of its distributed in-memory key-value store and object management. Fault tolerance is provided by backing up the data to a shared distributed file system. Trinity provides a platform for online queries as well as offline graph analytics with a vertex centric approach. In contrast to Trinity, DXGraph does not provide a special language to define data structures or using any of its services included. DXRAM's logging approach for fault tolerance is also very different to the backup solution of Trinity. However, Trinity and DXGraph share similar goals as well as the basic architecture for the application programmer with a vertex centric approach and a natural graph representation.

Also using a vertex centric approach for its graph applications, **Grappa** [8] is a shared memory runtime system for clusters and multicore computers not limited to offline and online graph processing, only. It abstracts hardware by creating a single address space for the application as well as executing code in the form of tasks. Tasks are scheduled by Grappa's tasking system using a work stealing approach when mapping to threads. Moreover, Grappa's scheduling ensures low context switch times for worker threads when executing tasks. Though sharing similar goals by not limiting the system exclusively to graph processing, the shared memory architecture is the key difference to DXRAM's distributed key-value store. Furthermore, DXCompute provides different methods for executing code. Either the programmer creates his own solution to execute custom application code or, he uses the job system or tasking system (refer to III-B) provided by DXCompute to delegate scheduling and execution. Currently, Grappa does not provide any mechanisms for fault tolerance, though the authors are considering this for their future work.

III. Architecture Overview

A. DXRAM Core

DXRAM is a distributed in-memory system for data centers and is optimized for large amounts of small data objects. Such objects are common in interactive applications like search engines or social media networks which are based on enormous data graphs. DXRAM keeps all data always in RAM providing low-latency access even for irregular access patterns. Node failures are masked by transparent logging and recovery [9]. Figure 1 shows the layered architecture of DXRAM including the new extensions DXCompute (see section III-B) and DXGraph (see section III-C). Several components implement the backend whereas services provide the API for the programmer. Every DXRAM node is either a peer or a superpeer. Peers store data objects, may run computations and exchange data directly with other peers, and also serve client requests when DXRAM is used as a back-end storage. Superpeers store global metadata like the locations of data objects, implement a monitoring facility, detect failures and coordinate the recovery of failed nodes, and also provide a naming service. Objects stored in DXRAM's key-value store are called *chunks*. Every chunk has a 64-bit globally unique ID called a *chunk ID (CID)*. This ID consists of two separate parts: A 16-bit node ID of the object creator and a 48-bit locally unique sequential number. Thereby, 65,536 nodes with around 280 trillion chunks per node are addressable. The sequential generated CIDs allow the use of compact global metadata management by using range-based B-trees on superpeers and compact paging-like address translation tables on peers. The address translation yields $\mathcal{O}(1)$ performance as well as overall low memory consumption. A custom memory allocator for small objects ensures low memory overhead per object. A chunk can have an arbitrary size of up 2 GB (Java byte array maximum size) and is stored in dynamic sized and chained blocks of up to 8 MB.

B. DXCompute

DXCompute is a new layer built on top of the DXRAM architecture adding services to execute computations locally and also remotely on storage nodes. Interactive queries on graph data are supported by providing lightweight *Job*s managed by the *JobService* which uses a per node configurable fixed size thread pool. A work stealing approach implements implicit load balancing between threads of one JobService [10]. If a job needs to access data located on a remote node, the job can be delegated to the data-owning node. This will improve data locality when executing the job and increase performance.

If a computation involves more than one node, multiple nodes have to be coordinated. The *MasterSlaveService* (see figure 2) implements *compute groups* within the DXRAM network topology consisting of one coordinator (master) and an arbitrary number of compute nodes (slaves). The master node controls the slave nodes of its group by managing joining/leaving of slaves to the compute group, accepting *compute task*s, scheduling compute tasks to all slaves and synchronizing slaves between compute tasks. When writing

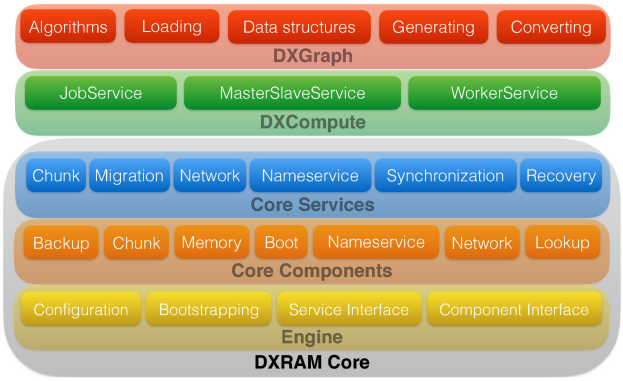

Figure 1: **The DXRAM layered architecture.** The engine and a collection of core services and components form the core of DXRAM. DXCompute is built on top of the core adding services for computations. DXGraph requires DXCompute and adds features for graph processing.

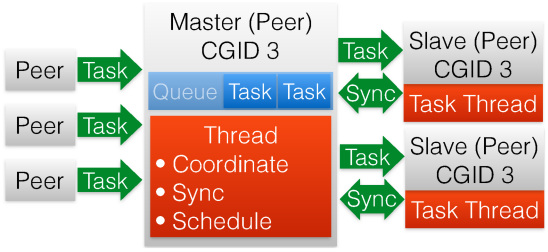

Figure 2: **MasterSlaveService architecture implemented in DXCompute.** The example shows three peers sending tasks to the master of the compute group (CGID) 3 with two slaves connected. Green indicates data flow over the network. Thread activity is colored red.

a compute task, the programmer has access to the current compute group's unique ID, the slave ID assigned to the node as well as node IDs of all other slaves of the current compute group and all of the core DXRAM services. The programmer can use the IDs as indices for partitioning his data or controlling the computation flow (see section V-B).

C. DXGraph

DXGraph extends DXCompute (see section III-B) by adding data structures and algorithms for graph generation, loading and processing. Currently, it contains compute tasks for the MasterSlaveService to load graph data from disk to DXRAM's key-value store and execute a multithreaded distributed BFS on a loaded graph. Vertices of the graph are represented naturally as *Vertex* objects and stored in DXRAM's key-value store (see IV-A). Details about the loading task are provided in section V-B and the BFS implementation is discussed in section V-C.

IV. Typical Graph Processing Steps

Data operations like online queries as well as offline graph processing are often based on a traversal of subgraphs. Accessing the neighborhoods of many vertices often results in irregular access patterns. Very large graphs do not fit into the main memory of a single machine and need to be partitioned, stored and processed on many machines (refer to IV-B). The combination of irregular access patterns on graphs stored on many machines typically results in a high network traffic.

The Graph500 benchmark [2] executes a breadth-first search on a huge graph to measure random access performance of clusters and shared memory machines. With BFS being one of the important building blocks for many graph algorithms and queries, it is a very good candidate to measure the overall performance of a graph processing system. This stress test is challenging for the network subsystem, the data lookup and the memory management. Many small data objects demand high efficiency and low overhead towards memory management.

A. Common Graph Representations

There are two ways to represent the graph and its components. The first method is a 2D representation as an edge matrix. A $N \times N$ sparse matrix, with N being the number of vertices of the graph, is created with entries (m, n) specifying that vertex m has an outgoing edge to vertex n. This is a typical format for shared memory [11] systems. The second method is a 1D representation as a collection of vertex objects. Each vertex object can contain further attributes but only a list of adjacent vertices is required. This representation fits GPUs [12], NUMA machines [13] or distributed memory systems [14]. This natural representation blends well with our key-value store by creating one object for each vertex and storing it with a unique CID. Furthermore, the graph is split into separate vertex objects allowing us to distribute them to different nodes easily as needed.

B. Graph Partitioning

As mentioned in section IV, if a graph does not fit into the memory of a single machine it needs to be split into multiple partitions. Partitioning a graph using FENNEL [15] or METIS [16] creates graph partitions with minimized edge cut reducing network communication on graph traversals compared to random partitioning. This can be part of the loading and graph construction step (see section IV-C) or a separate offline preprocessing step. However, computing an optimal partition is not a trivial task because the algorithms are in the category of NP-hard problems.

C. Data Loading and Graph Construction

Formats of existing graph data [17] might not fit the in-memory representation and require a conversion step for loading the data. This includes identifying vertices with IDs or hashes that are usable to the system or the actual in memory representation as objects or a sparse matrix. Basically, there are two approaches for converting graphs. *Offline conversion* is very flexible as we introduce a separate preprocessing step which does not have to involve the target graph processing system. This step converts the input data to an appropriate representation the target system can handle easily.

Online conversion constructs the graph from any (supported) input data format by executing the necessary steps during loading on the target system. However, the conversion steps might require additional memory and can lower the amount available for storing the final graph data. Furthermore, if the graph data is loaded over and over again, the online conversion will generate the same data but will always extend execution time of the loading phase.

Running an offline conversion step on the desired data set once and storing data in a fitting intermediate representation is the preferred approach to speed up the loading process which can take minutes or up to hours. When applying partitioning algorithms (see section IV-B), a preprocessing step is necessary anyway. Dynamic graphs like managed by Facebook grow and evolve over time and do not need the preprocessing steps.

D. The Breadth-First Search Algorithm

BFS is a building block for many graph processing algorithms. It traverses all reachable vertices from one source vertex determining their distance/depth. Algorithm 1 shows a common abstract implementation of the level synchronous top-down BFS algorithm.

The input graph is defined by $G(V, E)$ with the number of vertices $n = |V|$ and the number of edges $m = |E|$. BFS uses lists, also called *frontiers*, to keep track of vertices that have to be processed on the current iteration level (current frontier) and vertices that will be processed on the next iteration level (next frontier). Level synchronous BFS processes the graph in steps which we call *iteration level*s.

Performance for best-case and worst-case are equal because the search has to traverse all connected edges from the root. There are different ways to generate output data in this algorithm. Algorithm 1 stores the determined depth with each vertex, thus altering the input graph which might not be desired for some applications. Alternatively, one can store a list of parents for each vertex creating a spanning tree rooted at the input root r.

Algorithm 1: Sequential top-down BFS algorithm

Input: G(V, E), with dist for each neighbor nb of $v \in V$, nb.dist = -1; root vertex r
Output: G(V, E) with depth for each $v \in V$

```
1  r.dist = 0;
2  curfrontier ← r, nextfrontier ← ∅;
3  while curfrontier ≠ ∅ do
4      foreach v in curfrontier do
5          foreach neighbor nb of v do
6              if nb.dist = -1 then
7                  nextfrontier ← nextfrontier ∪ n;
8                  n.dist = v.dist + 1;
9      curfrontier ← nextfrontier;
10     nextfrontier ← ∅;
```

Beamer et al. [14] are proposing the "direction-optimizing BFS" algorithm, a hybrid approach for level synchronous BFS combining the classic top-down with a novel bottom-up approach to speed up BFS execution. When traversing the graph in top-down manner, the algorithm tries to visit every neighbor of every vertex of the current frontier on each iteration level. As the algorithm progresses and the depth level is increasing, many vertices are already visited resulting in many failed "not visited" checks. When the current frontier is large, most neighbors of the vertices in the frontier have already been visited but the top-down approach is still processing them. The bottom-up approach is more suitable in this situation. For every unvisited vertex of the graph, it checks if its list of neighbors contains one of the vertices in the current frontier i.e. is there a connection from any unvisited child to a parent of the current frontier. This requires keeping a list of already visited and unvisited vertices (see section V-A). By checking all unvisited vertices of the graph, the bottom up approach is only suitable if the current frontier contains a significant fraction of the graph. For a hybrid and high performant BFS implementation, one combines both approaches with top-down at the first and last iteration levels and bottom-up in the middle when the frontier is at its largest. Further details are explained in our implementation in section V-C2.

V. Graph Processing with DXGraph

The DXGraph layer contains an implementation of a distributed multithreaded direction-optimizing BFS algorithm with the DXRAM core and DXCompute layer. Furthermore, the layer contains data structures for the algorithm as well as tasks for generating and loading data. The DXRAM core provides the distributed key-value storage as well as message passing. Moreover, we used the MasterSlaveService from DXCompute to easily distribute and execute computation tasks on an arbitrary number of slave nodes.

A. Data Representation

Before execution, the graph data needs to be loaded. For storing the vertex data, we are using the natural 1D data representation (see section IV-A). The vertex IDs are refered to as CIDs and vice versa (depending on the context). For the implementation, both terminologies refer to the same number. Every vertex has a neighbor list of CIDs referencing other vertices stored as chunks and a field to assign the depth of the vertex.

B. Data Generation, Conversion and Loading

Input data is generated by the edge list generator of the Graph500 reference implementation [2]. The kronecker generator creates a random graph with low locality based on the scale and edge factor input parameters. A simple converter loads different graph input formats, such as the edge list format from the kronecker generator and creates an intermediate output graph suitable for DXRAM's key-value store allowing concurrent loading of the data on multiple nodes with low memory overhead. Furthermore, additional metadata

is generated to allow distribution of the data to an arbitrary number of nodes (random distribution). The metadata provides information on slicing the graph into almost equally sized partitions according to the number of nodes.

One thread on each DXRAM node is reading vertices of its assigned partition from the input graph file into an intermediate buffer. A second thread is removing vertices from the buffer, allocating memory in the key-value storage and storing the Vertex. Sequential loading of the vertices ensures that the sequentially generated CIDs are correctly assigned to match the order of the vertex IDs. However, the vertex IDs of the neighbor lists need to be re-based to match the node's local IDs starting with ID 1 on each node. This is a simple and inexpensive task because we can easily calculate a fixed offset for the IDs using the partition index for every vertex.

C. Optimized BFS Implementation

1) Data Structure BitVector: Efficiency of the algorithm is not only determined by the implementation of the algorithm itself but also by high performant and low overhead concurrent data structures. BFS requires an implementation of a frontier data structure storing the vertices to be processed in the current iteration and for all vertices to be processed in the next iteration. Common dynamic data structures like lists or queues are not very suitable for large graphs. Storing a 4 byte vertex ID, using an array to implement the data structure, the frontier requires 4GB for a graph with one billion vertices. Moreover, to avoid duplicates that increase memory consumption, the list needs to be iterated and checked if a vertex is already in the list. This operation is very expensive ($O(n)$ runtime) and slows down overall execution time of the algorithm.

Hence, a data structure providing $O(1)$ lookup time for entries, a low per vertex memory overhead, no duplicate entries and efficient concurrent access is required.

We address these challenges with a static bit vector based data structure called *BitVector* similar to Berrendorf's bitmap implementation [13]. Each bit represents the vertex ID (vid) by its index in the continuous array of bits where 1 indicates the list "contains" the vertex and 0 "not in the list". This allows highly efficient lookup, insert and remove operations using a primitive long *array* (64 bit per array entry) in $O(1)$ providing the following operations:

```
// Entry check
(array[vid / 64] & (1 << (vid % 64))) > 0
// Entry set
array[vid / 64] |= (1 << (vid % 64))
// Entry clear
array[vid / 64] &= ~(1 << (vid % 64))
```

The BitVector avoids inserting duplicates implicitly and yields constant time performance when adding entries. Implementing low overhead synchronization using compare and swap (CAS) operations is simple and allows highly concurrent access.

2) Algorithm: The algorithm is implemented as a task to be executed with our MasterSlaveService. Algorithms 2 and 3 show simplified versions of the control flow of the implementation. Furthermore, figure 3 shows the vertex data

Algorithm 2: Simplified version of our main task thread of the BFS implementation

```
   Input: G(V,E), RootVertexList rootlist
   Output: G(V, E) with depth for each v ∈ V
1  bfslevel = 0;
2  BitVector curfront, nextfront, visited;
3  startWorkerThreads();
4  foreach root in rootlist do
5      curfront.clear(); visited.clear();
6      if IsStoredCurNode(root) then
7          root.markVisited(bfslevel);
8          visited.insert(root);
9          curfront.insert(root);
10     loop = true;
11     while loop do
12         while curfrontier.notEmpty do
13             yield ;  // Wait for workers, hot standby
           /* BFS level sync with remote workers,
              exchange nextfront state                */
14         allRemoteNextFrontsEmpty =
            bfsLevelBarrierSync();
15         if nextfront.empty and allRemoteNextFrontsEmpty
            then
16             loop = false ;              // BFS finished
17         else
18             swapFrontiers(curfront, nextfront);
19             nextfront.clear();
20             bfslevel++ ;        // next iteration level
21 stopWorkerThreads() ;            // Workers: run=false
```

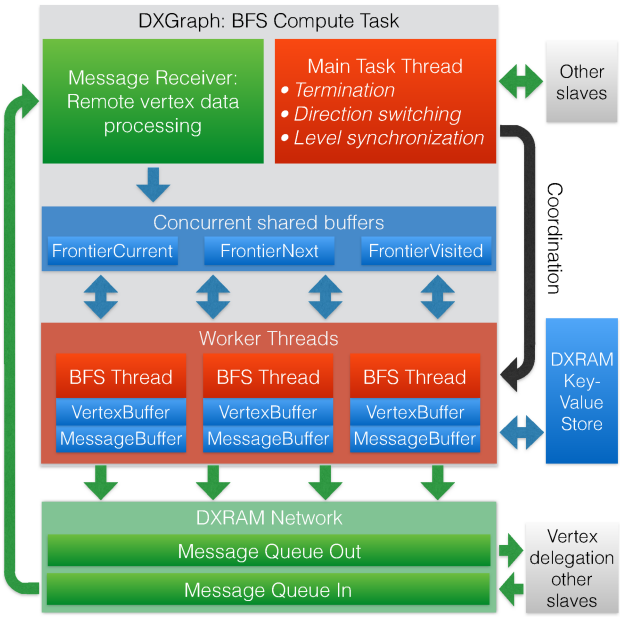

Figure 3: **Architectural overview of DXGraph's BFS implementation** Red indicates threading, green network (vertex) data flow and blue local (vertex) data flow

Algorithm 3: Simplified version of a single worker thread of the BFS implementation (top-down only)

```
   /* Each thread has shared access                    */
   Input: Frontier curfront, nextfront, visited; bfslevel
1  run = true ;        // loop, termination by main task
2  Vertex[] vertexBuffer ;          // Thread local buffer
3  while run do
4      if curfrontier.empty then
5          yield ;  // Wait for vertex IDs, hot standby
6      else
           /* Get vertex objects from storage          */
7          fillVertexBuffer(curfront, vertexBuffer);
       /* Vertices in curfront are always local        */
8      foreach vertex in vertexBuffer do
9          foreach neighbor in vertex.neighbors do
               /* Processing local neighbors only,
                  delegating remote ones               */
10             if IsStoredCurNode(neighbor) and
                visited.contains(neighbor) then
11                 neighbor.markVisited(bfslevel);
12                 visited.insert(neighbor);
13                 nextfront.insert(neighbor);
14             else
                   /* Remote node adds non visited
                      vertices to nextfront            */
15                 sendVertexToNodeOwner(neighbor);
```

flow. The bottom-up code is further explained below, but was removed in the pseudo-code for better readability.

A fixed but configurable number of worker threads is used on each node. The threads are kept on hot standby to avoid startup latencies and are stopped after the algorithm has terminated. Each worker thread has its own local vertex buffer for buffering a configurable number of vertices to enhance data locality and improve throughput. Another buffer for sending non-local vertex IDs to remote nodes is explained below. The current, next and visited BitVectors are allocated per node and shared among the threads on a single node.

For a prior loaded graph, a single vertex ID or list of vertex IDs is provided as root(s) to the task as input parameters. The algorithm is started on the slave owning the root vertex. The node marks it as visited and adds it to the current frontier. The worker threads are accessing the frontiers concurrently and "stealing" vertices from the current frontier resulting in implicit load balancing among them. For each locally buffered vertex, a thread checks its neighbors and determines if each neighbor is stored on the current node. If the neighbor is locally stored, it marks it as visited and adds it to the next frontier. Otherwise, it sends the vertex ID to the remote node owning the vertex by adding it to the vertex message buffer. The remote node receives a message with the vertex ID, checks if the vertex is already visited, marks it as visited and adds it to his next frontier. Additional steps to retrieve the actual vertex chunks from the storage are necessary before iterating the vertex's neighbor list. Delegate messages with vertex IDs are sent in batches to better utilize network bandwidth and batch processing on the remote node.

When the main task thread detects that his local current frontier is empty, it finishes the iteration by synchronizing with the other nodes (see section V-C4). With the exchanged information on this step, every node can determine if it has to terminate the breadth-first search and stop the worker threads or continue with the next level and swapping their current and next frontiers.

3) Top-Down with Bottom-Up, a Hybrid Approach: Our initial top-down only implementation was already performing very well, but in combination with a bottom-up approach, we were able to improve execution time even further (see section VI). Before each level iteration, every node checks if it has to run the upcoming iteration top-down or bottom-up. Provided by Beamer et.al [14], $m_f > \frac{m}{10}$ determines if we switch from top-down to bottom-up and $n_f < \frac{n}{14k}$ determines if we switch from bottom-up to top-down with m_f being the number of edges and n_f the number of vertices in the current frontier of all nodes. k specifies the graph's degree. All necessary information is exchanged on the multicast level synchronization with all other nodes ensuring that every node runs the same approach for the current level.

4) Synchronization: Level synchronous BFS requires synchronizing all participating threads of all nodes after each iteration level. We decided to implement an all-to-all barrier [18] using our efficient network subsystem and atomic counters to keep the latency between BFS iteration levels low. Furthermore, we combine this synchronization step with exchanging data for local calculation of top-down/bottom-up switching and BFS termination to avoid adding more messaging overhead. When reaching the barrier, each node sends his next frontier vertex count as well as edge count to all other nodes and waits on hot standby. Each node waits until it received this data from all other nodes. The last incoming message releases the barrier and the waiting thread is released immediately ensuring low delays.

VI. Evaluation

We evaluated DXGraph's BFS and compared it to equivalent implementations of the two state-of-the-art systems Grappa [8] and GraphLab [5]. We analyze memory consumption and overhead of the loaded graph data as well as execution time of the algorithm. In this paper, we did not evaluate loading times. All systems are loading and processing the graph with mechanisms for persistency and fault-tolerance disabled. Because of the preprocessing step, our loading phase is much faster than Grappa's or GraphLab's which are lacking this extra step. Furthermore, this aspect is not important as we are aiming for online processing and analysis with graph data generated and evolving by interactive user input.

The input graph data was generated by the Graph500's reference implementation of a Kronecker generator. The graph's *scale* is the logarithm base two of the number

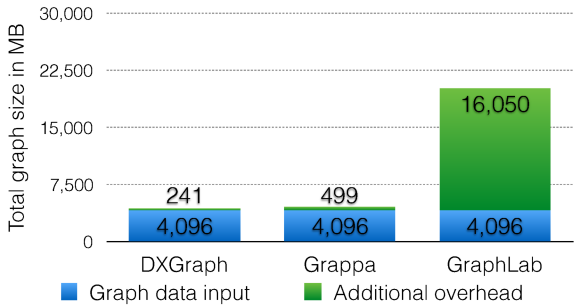

Figure 4: **Memory usage of loaded/constructed graph split into input graph size and additional overhead.**

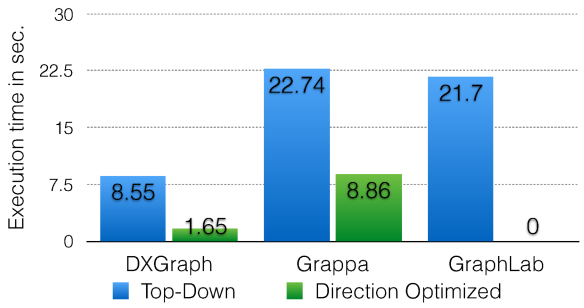

Figure 5: **Lowest execution times of the standard top-down and direction optimized BFS.**

of vertices, i.e. $numverts = 2^{scale}$. The *edgefactor* describes the ratio of the graph's edge count to its vertex count and delivers the total edge count of the graph with $numedges = 2^{scale} \cdot edgefactor$. The total size is calculated by $numedges \cdot 16$ with a vertex ID size of 8 bytes resulting in an edge size of 16 bytes. The generated graph contains empty/isolated vertices, self-loops and duplicate edges. According to the specification, these are stored with the generated data but can be filtered when constructing the graph. All three systems do not execute any filtering on the input data. Furthermore, graph data is randomly distributed for all systems without using partitioning algorithms as desribed in section IV-B.

We created a scale 24 and 27 graph using the generator. The largest connected subgraph for the scale 24 graph has 8,864,904 vertices and 536,865,232 non duplicate edges. For the scale 27 graph, it spans 63,035,883 vertices and 4,293,897,563 non duplicate edges. For both subgraphs, the non duplicate edge count covers over 99.99% of all edges for each input graph. The subgraph sizes were determined by counting vertices and edges using DXGraph's BFS implementation. Currently, we are limited to the scale 27 graph because the generator must be executed on a single node and requires the same amount of memory as the generated output data. However, we already started to switch to a UV2000 shared memory machine with 16 TB RAM provided by the computing center of our university which will allow us creating much larger graphs.

We used the scale 24 graph for comparing the systems because we were not able to load anything bigger on our cluster with Grappa. We are in contact with the developers and hope to resolve this to allow future evaluations with even bigger graphs. All tests were executed on four nodes of our cluster with Intel Xeon E5-1660 CPUs (6 cores with hyper-threading) and 64GB RAM connected by Gigabit Ethernet running Debian 8.4. For DXGraph, we used OpenJDK's Java 1.8 runtime.

A. Memory Overhead

Low memory overhead per vertex is crucial to utilize the available amount of memory efficiently. Thus, we are comparing the amount of memory used by the loaded and constructed graph on each system. Figure 4 shows the amount of memory on each system occupied by the constructed scale 24 graph before BFS execution. GraphLab's demand for space is approximately five times the size of the input graph (4096 MB) where Grappa and DXGraph add far less overhead. DXGraph's results are based on a compact vertex data representation as well as its highly efficient and low footprint memory management and custom allocator suited for many small objects [9].

B. Execution Time and MTEPS

Following the specification of the second kernel of the Graph500 benchmark, we have evaluated the execution time of the algorithm. Execution speed is classified in million traversed edges per second (MTEPS).

For Grappa and GraphLab, we executed the algorithm on each system multiple times with different random root vertices. DXGraph used the root vertex list generated by the kronecker generator with 64 random root vertices. Performance of the algorithm is influenced by the root vertex determining the spanned subgraph which might also lead to best or worst case performance. This effect can be observed especially with the direction optimized algorithm [14].

Figure 5 compares each system's peak performance (lowest execution times). We compared the top-down only implementations of all three systems as well as the direction optimized versions of DXGraph and Grappa (no implementation was available for GraphLab). For the standard top-down algorithm, DXGraph's implementation is about 2.5 times faster than Grappa's and GraphLab's. Furthermore, DXGraph's top-down implementation can even keep up with Grappa's direction optimized implementation. Comparing the lowest execution time of the direction optimized versions, DXGraph's Java implementation is 5 times faster than Grappa's C++ implementation.

Figure 6 shows all 64 execution times of DXGraph's BFS implementation. The blue data points are the runs of the standard top-down with an average execution time of 9.0 seconds and the green data points are the runs of the direction optimized approach with an average of 5.9 seconds. Both average times are still outperforming Grappa's and GraphLab's lowest execution times. For the top-down version, the first data point reflects the JVM's runtime optimization of the

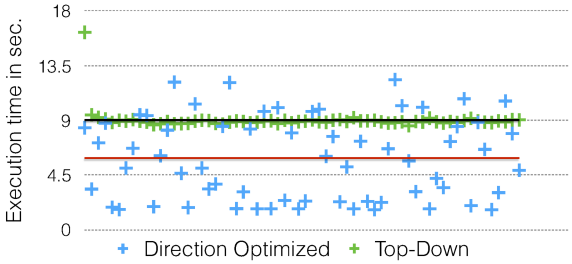

Figure 6: **Execution times of 64 top-down (green) and direction optimized (blue) BFS runs on DXGraph.** The black line marks the avarage execution time for all top-down runs and the red line for all direction optimized runs.

generated byte code. The performance of any run of the direction optimized variant depends on the root vertex but many show optimal or close to optimal execution times. Even for runs not performing well, only a few are exceeding the average execution time of the top-down only approach.

With a total number of 536,865,232 non duplicate and non self-loop traversed edges (as described in VI), an overall peak performance of 24.740 MTEPS is reached for GraphLab, 60.594 MTEPS for Grappa and 325.373 MTEPS for DXGraph. Furthermore, running the scale 27 graph with a total number of 4,293,897,563 non duplicate and non self-loop traversed edges, DXGraph's peak performance hits 323.579 MTEPS with a lowest iteration time of 13.27 seconds.

VII. Conclusions

In this paper, we proposed DXGraph, an extensible graph framework implemented on top of the in-memory key-value store DXRAM, as well as the additional layer DXCompute for executing computations on storage nodes. Furthermore, we implemented a standard top-down as well as direction optimized BFS algorithm as specified by the second kernel of the Graph500 benchmark using DXRAM with its key-value store and networking subsystem and the master-slave service of DXCompute. Our low memory footprint data structure BitVector allows highly parallel execution of the BFS algorithm with multiple threads on each node and low synchronization overhead. Efficient communication and low overhead synchronization during BFS runs are achieved using straight forward message passing with DXRAM's network subsystem. The comparison of equivalent BFS implementations executed on Grappa and GraphLab, two state-of-the-art graph processing systems, shows that DXGraph outperforms both systems with a peak throughput of over 323 million traversed edges per second, which is about five times the throughput of Grappa and thirteen times the throughput of GraphLab, on a graph with over 4 billion edges. In the future, we want to extend the evaluation of the BFS algorithm to further systems, such as Trinity and GraphX. We also want to scale out to more nodes with the goal to move to the cloud, also with Infiniband. Moreover, we want to increase the graph size to the terabyte scale which is also a good opportunity utilizing the vast cloud resources. Other essential algorithms like page rank will enhance DXGraph and allow us to compare not only to Grappa and GraphLab, but also to further systems as well. Moreover, we want to switch to online analytics on dynamic graphs.

References

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1804–1815, 2015.

[2] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," 2010.

[3] A. Gulli and A. Signorini, "The indexable web is more than 11.5 billion pages," in *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, 2005, pp. 902–903.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2010, pp. 135–146.

[5] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, pp. 716–727, 2012.

[6] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems*, New York, NY, USA, 2013, pp. 2:1–2:6.

[7] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2013, pp. 505–516.

[8] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Grappa: A latency-tolerant runtime for large-scale irregular applications," University of Washington, Tech. Rep., 2014.

[9] F. Klein, K. Beineke, and M. Schoettner, "Distributed range-based meta-data management for an in-memory storage," in *LNCS Europar Workshop Proceedings, 4th Big Workshop on Big Data Managements in Clouds*, 2015.

[10] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2010, pp. 303–314.

[11] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2," in *Proceedings of the 2006 International Conference on Parallel Processing*, 2006, pp. 523–530.

[12] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating cuda graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, 2011, pp. 267–276.

[13] R. Berrendorf and M. Makulla, "Level-synchronous parallel breadth-first search algorithms for multicore and multiprocessor systems," in *Proc. Sixth Intl. Conference on Future Computational Technologies and Applications (FUTURE COMPUTING 2014)*, 2014, pp. 26–31.

[14] S. Beamer, A. Buluc, K. Asanovic, and D. A. Patterson, "Distributed memory breadth-first search revisited: Enabling bottom-up search," EECS Department, University of California, Berkeley, Tech. Rep., 2013.

[15] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, 2014, pp. 333–342.

[16] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, 1998.

[17] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[18] O. Villa, G. Palermo, and C. Silvano, "Efficiency and scalability of barrier synchronization on noc based many-core architectures," in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2008, pp. 81–90.

Chapter 4

Concurrent Low-Latency Data Access for Parallel Java Applications

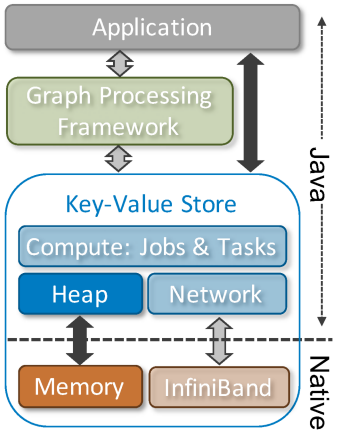

Figure 4.1: The "big picture" of this thesis with the relevant components for this chapter highlighted ("Application" layer and interface, "Memory" Java, and "Memory" Native).

This chapter discusses the second research question "Can the local storage provide low latency data access and scalability on highly concurrent local computations benefitting from data locality?" in a Java environment (see Section 1.3).

First, Section 4.1 presents the addressed requirements concerning a Java-based in-memory key-value storage that result from this major question based on the previously introduced context (see Section 1.1) and its challenges (see Section 1.2). The stage of work of the DXRAM storage system is presented in Section 4.2, with Section 4.3 elaborating on the major research question in detail. The resulting contributions of this work are presented in Section 4.4 followed by a copy of the publication:

Stefan Nothaas, Kevin Beineke and Michael Schöttner. "Optimized Memory Management for a Java-Based Distributed In-Memory System". In Proceedings of the 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). 2019. 10 pages.

4.1 Requirements

Disk-based solutions are not an option when considering highly interactive applications and their requirements presented in Section 1.2. Thus, all data must be kept in memory. This approach ensures a fundamentally low access-latency for all data which is mandatory, especially for typical graph algorithms with highly random access patterns.

To fulfill the proposed application requirements, the memory management must be optimized for the commonly used object size (< 128 bytes) commonly found in graph data-sets. The more objects stored per server, the higher the data locality for local computations. This increased per-server density lowers the likelihood of requesting data from remote servers and, thus, lowers overall and more expensive inter-server communication. Furthermore, the whole system requires overall less memory for storage which also lowers hardware costs.

However, concurrent access to local data needs to be synchronized to avoid user data and also metadata (e.g. of the allocator) corruption. With a focus on read-heavy workloads, get-operations dominate. The design of concurrency control, e.g., using locks, has to consider this to avoid performance penalties for the dominating operation type.

In the object-oriented language Java, data is naturally modeled using classes. With data stored as binary serialized data blobs in-memory, the memory management has to support fast, and memory efficient serialization of complex and even nested Java objects to memory.

On long-running applications (e.g., 24/7 services), memory fragmentation caused by rather seldom allocations and deletions cannot be avoided in the long term. With the majority of objects being rather small, external fragmentation is not critical and can be minimized by periodically compacting the memory, for example during low-load phases.

4.2 Stage of Work

To address the requirements presented in Section 4.1, the research regarding the memory management of this thesis was conducted on the Java-based DXRAM in-memory key-value storage system. Initially proposed by Dr. Florian Klein [63, 61], DXRAM's memory management already implemented a low-overhead allocator using Java's Unsafe class for managing many small objects in RAM efficiently. The allocator managed a large pre-allocated block of memory by splitting it into a configurable number of logical fixed-size segments.

The arena manager implemented concurrency management on a segment-basis by assigning application threads, creating new chunks, to different segments. Naturally, access to existing chunks (get, put and remove) had to access the segment the chunk was located in. A segment was locked before thread access and unlocked after the operation completed. The memory manager implemented support for CRUD operations, only.

CIDs were translated to native memory addresses using a paging-like address translation named CIDTable. By re-using CIDs, the tables were kept densely packed minimizing storage costs. The tables were stored using the allocator of the memory management.

Locking of chunks was optional and implemented as an external service. Chunks had to be locked/unlocked explicitly and could still be accessed or even modified without using the service. This design could lead to data races if the application developer did not use this service properly. The service supported per chunk read-write locks stored on-heap using a standard Java Map. This method is not very memory efficient and limits the total number of storable locks to 2^{31} (signed integer).

A chunk was implemented as a simple tuple consisting of a CID and a binary data blob. An external serialization had to be used to serialize chunks to binary data to allow storing them to the back-end storage.

4.3 Research Questions

The initial DXRAM implementation was already storing all data always in-memory to keep local data access latency low. Furthermore, it was already optimized for storing tiny objects efficiently making it suitable for storing large graph data-sets.

With a natural representation of data using Java objects, efficient de-/serialization is mandatory to store the objects as binary data outside of the Java heap. This off-heap storage also avoids performance penalties imposed by the Java garbage collection (e.g., expensive collection-phases). However, how can the memory management guarantee low-latency local data-access for applications/algorithms running on the storage? This requirement proposes a challenging task for supporting highly concurrent applications.

With applications using multiple threads to run algorithms in parallel, concurrency is the rule. However, this introduces data races which have to be considered. The initial DXRAM implementation addressed this by managing access to multiple segments using the arena manager. However, profiling has shown that this management introduced significant locking overhead and did not favor commonly used get-operations which are typical for most graph-based applications and algorithms (see Section 4.1).

Furthermore, this mechanism only protected the memory manager's metadata and did not avoid races on user data. The application programmer had to provide his/her solution for synchronizing concurrent access. However, this is a general issue which applies to any concurrent application and, thus, should be addressed by the memory management instead. With the current solution not addressing this adequately, how can the memory management provide

(local) data consistency guarantees for user data without sacrificing performance and the low per-object memory overhead? What is the best granularity of the synchronization/locking mechanism? How does the granularity affect the performance of the application and memory management when relying on less fine-granular locking or a per-object synchronization? Is it essential to distinguish these cases regarding performance?

4.4 Contributions

Our publication addresses these questions by an optimized and extended memory management [93]. It fulfills the requirements of concurrent (graph-based) applications while still providing a very low per-object memory overhead for many small objects. The contributions stated below which are not explicitly assigned to any author/contributor are by the author of this thesis.

First, a custom serialization was implemented to allow fast and efficient de-/serialization of complex and even nested Java objects. It enables the memory management to de-/serialize the objects directly from/to the native memory area without additional buffering (zero-copy). Later, this serialization framework was also used in DXNet (see Chapter 5) to provide a consistent interface for serializing data, chunks, and messages to send them to remote servers. This serialization was further extended by Dr. Kevin Beineke to support interrupting the de-/serialization process at any byte position as well as handling of over- and underflows on a ring buffer data structure (see Section 5.4.2).

Extensive benchmarking and profiling of the memory management revealed that the arena manager with its multi-level locking limited the performance on typical read-heavy workloads. First, we proposed a new design that replaced the arena manager with a less expensive and straightforward read-write lock mechanism. Furthermore, memory segmentation was removed as it did not provide any benefits to our typical parallel workloads but increased the overall complexity of the system. The new approach improved the overall performance significantly and was the first step towards developing an appropriate solution.

This solution included a new design for a low-overhead fine-granular per-chunk read-write lock. It ensures that the per-chunk memory overhead is not increased significantly but allows low-overhead fine granular locking for synchronizing concurrent access and modification of data. Furthermore, a chunk can also be used just like a lock, with or without storing user data, to allow coarse granular locking on the application level, e.g., a single lock on the root of a linked list.

Further proposals for concurrency optimizations were analyzed and confirmed to be less optimal by the bachelor thesis of Florian Hucke [51]. The evaluation in our publication shows, that our solution is significantly faster compared to Hazelcast and InfiniSpan, two state-of-the-art Java-based in-memory caches, and provides single-digit microseconds access latency even on highly concurrent workloads.

The API of the memory management was further adapted to the application domain to support batch allocations to speed up uploading of large data-sets. The set of operations was extended by a *resize*-operation to allow resizing of existing chunks (e.g., required for expanding/shrinking

arrays), a *reserve*-operation to reserve CIDs without allocating memory (e.g. required for loading phases when assigning/mapping CIDs to data to load), *direct memory access*-operations (to modify single fields/values without having to read an entire chunk) and a *pinning*-operation. The latter is mandatory to enable proper chunk management on future RDMA hardware access (see Section 7.3.2).

An interface for a concurrent defragmentation thread allows implementing and evaluating different defragmentation strategies (see Section 7.3.1).

The memory management of DXRAM was moved to and published as the separate open-source library DXMem which can now be used by any (concurrent) Java application requiring an efficient memory manager for storing many small objects with low-overhead concurrency management. Additionally, DXMem provides a built-in YCSB-like benchmark to quickly evaluate it with configurable workloads and an interactive command line tool for testing and debugging (loading and analyzing of heap dumps).

Stefan Nothaas wrote the paper and evaluated all systems presented in it. Prof. Dr. Michael Schöttner reviewed the paper several times and helped improve it. Dr. Kevin Beineke, Prof. Dr. Michael Schöttner, and Florian Hucke took part in many discussions about the design and performance analysis of DXMem.

Optimized Memory Management for a Java-Based Distributed In-Memory System

Stefan Nothaas
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
stefan.nothaas@hhu.de

Kevin Beineke
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
kevin.beineke@hhu.de

Michael Schoettner
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
michael.schoettner@hhu.de

***Abstract*—Several Java-based distributed in-memory systems have been proposed in the literature, but most are not aiming at graph applications having highly concurrent and irregular access patterns to many small data objects. DXRAM is addressing these challenges and relies on DXMem for memory management and concurrency control on each server. DXMem is published as an open-source library, which can be used by any other system, too.**

In this paper, we briefly describe our previously published but relevant design aspects of the memory management. However, the main contributions of this paper are the new extensions, optimizations, and evaluations. These contributions include an improved address translation which is now faster compared to the old solution with a translation cache. The coarse-grained concurrency control of our first approach has been replaced by a very efficient per-object read-write lock which allows a much better throughput, especially under high concurrency. Finally, we compared DXRAM for the first time to Hazelcast and Infinispan, two state-of-the-art Java-based distributed cache systems using real-world application-workloads and the Yahoo! Cloud Serving Benchmark in a distributed environment. The results of the experiments show that DXRAM outperforms both systems while having a much lower metadata overhead for many small data objects.

***Index Terms*—Memory management, Cache storage, Distributed computing**

I. Introduction

The ever-growing amounts of data, for example in big data applications, are addressed by aggregating resources in commodity clusters or the cloud [20]. This concerns applications like social networks [13], [23], [24], search engines [19], [29], simulations [30] or online data analytics [18], [33], [34]. To reduce local data access times, especially for graph-based applications processing billions of tiny data objects ($<$ 128 bytes) [16], [27], [32], backend systems like caches and key-value storages keep all data in-memory.

Many systems for big data applications, such as frameworks [22], [26], databases [3], [4], or backend storages/caches [5], [7], [8], [28], are written in Java. However, many of them cannot handle small data objects (32 - 128 byte) efficiently and introduce a considerable large metadata overhead on a per-object basis. Compared to traditional disk storage solutions, RAM is more expensive and requires sophisticated memory management. High concurrency in big data applications is the rule but adds additional challenges to ensure low access-times for local and remote access and to provide mechanisms for synchronizing concurrent access. Combined with today's low-latency networks, providing single-digit microseconds remote access times on a distributed scale, local access times must be kept low to ensure high performance which is challenging in general but especially in Java.

DXMem is the extended and optimized memory management of DXRAM. It provides low metadata overhead and low-latency memory management for highly concurrent data access. Data is stored in native memory to avoid memory and garbage collection overhead imposed by the standard Java heap. DXMem uses a fast and low-overhead 64-bit key to raw memory address mapping. Java objects are serialized to native memory using a custom lightweight and fast serialization implementation.

Furthermore, DXMem offers a low-overhead per-object read-write locking mechanism for concurrency management as well as memory defragmentation for long-running applications. On an average object size of 32 bytes, DXMem can store 100 million objects with just 22% additional overhead (§VI-A). On a typical big data workload with 32 byte objects, 95% get and 5% put operations, DXMem achieves a local aggregated throughput of 78 million operations per second (mops) with 128 threads which is an up to 28-fold increase compared to Hazelcast [7] and Infinispan [8], two Java-based state-of-the-art in-memory caches (§VI-B). Using the Yahoo! Cloud Service Benchmark [17], we compared DXRAM with DXMem to Hazelcast and Infinispan (§VI-C), too. The results show that DXRAM scales well with up to 16 storage servers and 16 benchmark clients on real-world read-heavy workloads with tiny objects outperforming the other two systems.

Our previous publication [21] **has addressed the following contributions**:

- The initial design of the low-overhead memory allocator
- The address translation (CIDTable) without per-chunk locks
- An arena-based memory segmentation for coarse-grained concurrency control and defragmentation

The contributions of this paper are:

- Reduced metadata overhead while supporting more storage per server (up to 8 TB, before 1 TB)

- Low-overhead Java object to binary data serialization (the old design supported binary data only)
- Optimized address translation (faster than the old design with translation cache)
- Efficient fine-grained locking for each stored object
- New experiments and comparisons with Infinispan, and Hazelcast

To evaluate the local memory manager performance of storage instances, we created a microbenchmark based on the design and workloads of the YCSB and implemented clients for the systems evaluated in this paper (§VI-B). DXMem is also published as a separate Java library that can be used by any Java application. DXRAM and DXMem are open-source and available at Github [6].

The remaining paper is structured as follows: Section II presents the target application domains and their requirements. Section III presents related work. We give a brief top-down overview of DXMem and its components in Section IV before explaining them in detail in a bottom-up approach in the following sections. Starting with Section V, we explain important details about memory management in Java before elaborating on DXMem's allocator in Section V-A. This section is followed by Section V-B which describes the CIDTable translating chunk ID to native memory addresses. The design of the fine-granular locks is presented in Section V-C The evaluation and comparison of DXMem to Hazelcast and Infinispan is presented in Section VI. Conclusions are located in Section VII.

II. Challenges and Requirements

This section briefly presents the target application domains which were already introduced in Section I. Often, Big data applications use batch-processing frameworks (e.g. Hadoop [26], Kafka, [22]) or are live systems (e.g. social networks [13], [23], [24] or search engines [19], [29]) serving **many concurrent requests of interactive users**. Many systems and applications are written in **Java**, which is popular because of its strong typing, sophisticated language features, platform independence, and rich libraries and have to address the following challenges and requirements.

Fast local response times. In-memory caches are used to mask slow disk access times for stored data. Some applications take this approach one step further by storing **all data always in-memory**.

Data distribution. Often, one commodity cluster node is not sufficient to store and process vast amounts of data.

Fast remote response times. Low remote latency on inter-node communication is ensured by **low-latency network interconnects**, e.g., InfiniBand should be considered which in turn demand low local latency not to become the bottleneck instead. However, with many applications and frameworks in Java, access to such low-level hardware is very challenging.

Fast and efficient (remote) object lookup. With billions of objects distributed across multiple nodes, object lookup becomes a challenge, too. Often, a key-value design combined with hashing is used to address this issue [3], [7], [8], and the standard API provides **CRUD operations** (create, read, update, delete).

Very small objects. Typical data models for big data applications include tables, sets, lists, and graph-structured data [31]. For the latter, storing billions of objects becomes a challenge because the **per-object overhead must be kept low**. With the limited amount of main memory, storing more objects per node does not only require fewer nodes to store all data but also increases locality and performance.

High concurrency. Simultaneously serving **many concurrent interactive user requests** or using many threads to lower execution times of algorithms, e.g., graph traversal, high concurrency is a must. On today's multi-core hardware, **concurrency support and optimizations** are inevitable. However, **with concurrency data races must be considered** and require mechanisms to synchronize data access on concurrent modification without limiting concurrency and increasing access latency too much.

III. Related Work

Common purpose memory management, algorithms, and allocators are widely studied in literature and are beyond the scope of this paper but have been discussed and evaluated in our previous publication [21]. This paper extends this foundation and focuses on a variety of changes to address the requirements and challenges imposed by our target application domain (§II). Thus and due to limited space, we focus only on relevant Java-based in-memory caches and storages which are designed for the same application domain.

Hazelcast [7] is a distributed in-memory cache and computing platform implemented in Java. It organizes data using implementations of standard Java collection interfaces, e.g., List, Queue, Map or Set. Hazelcast offers three storage options for objects: As serialized binary data in native memory (High-Density Memory Store) using a custom serialization similar to DXMem's, as Java objects on the Java heap or stored as both. It implements a peer-to-peer protocol to form a cluster of storage nodes. Ethernet using Java NIO is supported for remote node communication. Objects are stored to one of the 271 partitions distributed to storage nodes using hashing [12].

Infinispan [8] is a distributed in-memory key-value storage implemented in Java. Based on a peer-to-peer architecture, objects are hashed and stored using a cache interface extending the Java Map interface. It supports networking over TCP using Ethernet-based transport implementations, e.g., Netty. Objects are stored either on the Java heap or in native memory. For binary de/-serialization to/from native memory, InfiniSpan provides *Externalizers* which use the JBoss Marshalling framework. By default, objects are distributed to storage nodes using consistent hashing. The application has the option to distribute objects manually to optimize data access times.

Ignite [3] is a distributed in-memory data grid and processing platform implemented in Java. Objects are stored as key-values either to the Java heap or off-heap using a tiered storage model. Ignite supports ACID transactions for consistency and built-in distributed data structures. Clients

determine the location of objects by hashing them explicitly without involving additional servers of the system. Network communication is implemented using TCP sockets.

Apache Geode [5] (commercially GemFire) is a Java-based distributed cache. It offers a key-value interface by storing a distributed implementation of the Java Map in regions. Data such as metadata, keys, indices or values are stored on-heap, but values can also be stored off-heap in regions and are managed in slabs. A region, depending on the type, is stored on a single node locally (not distributed), divided into buckets for distribution (partitioned) and also replicated. A custom serialization is provided for storing data off-heap for improved performance over Java's *Serializable*. Data is exchanged peer-to-peer using TCP or UDP sockets over Ethernet.

Ehcache [11] is a Java-based cache implementation. A cache manager manages one or multiple caches consisting of one or multiple storage tiers. A storage tier defines where the data is stored: on-Java-heap, off-heap, on disk or clustered on a remote cache server. This tier-based storage allows applications to leverage different storage types by storing the hottest data closer to the application to ensure low access times. Data stored off-heap is serialized using a custom serialization. Ehcache supports eventual and strong consistency and also transactions for operations. On a multi-server setup, data can be distributed to multiple servers using a distributed hash table. The data is split into stripes and stored on the remote servers. Network communication is implemented using Java RMI.

DXRAM [14] is a Java-based distributed in-memory key-value storage using **DXMem** for managing objects stored locally on nodes. DXRAM shares many concepts with the other systems presented but also implements different and new ideas. For a distributed object-lookup, DXRAM implements a chord-like overlay with dedicated nodes for storing and providing the metadata which is a unique approach compared to the other systems not using dedicated servers for storing metadata for object lookup. DXMem stores all objects in native memory, like the other systems presented, to avoid the drawbacks of the Java garbage collection. However, DXRAM nor DXMem offer multiple storage types like additional on-heap or explicit disk storage. A custom object serialization allows efficient and fast serialization of primitive data-types and nested complex objects. When storing data off-heap, this is mandatory and also implemented by all systems presented in this Section. DXMem does not use hashing but instead implements a custom paging-like address translation. Thus, DXMem does not automatically distribute data to servers, but an application decides where to store the data which, of course, can be determined using a hashing algorithm as well. Using DXNet [15], DXRAM supports Ethernet over TCP using Java NIO and low-latency InfiniBand networks using native verbs which is not available on the other systems presented.

IV. DXMem: Architecture Overview

This section presents *DXMem*, the extended and optimized memory manager specifically designed to address the requirements and challenges presented in Section II. Figure 1 depicts

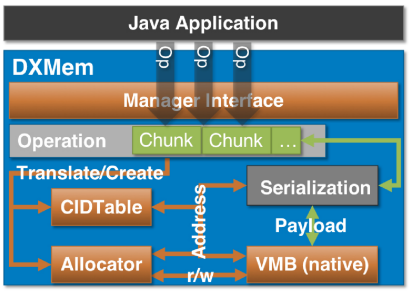

Fig. 1. Simplified DXMem Architecture

a simplified view of its architecture. DXMem implements a key-value store data model with a value stored as binary data of serialized Java objects, referred to as *chunks*, and a 64-bit key called *chunk ID (CID)* (§V-B). A custom allocator (§V-A) stores data outside of the Java heap to avoid the drawbacks explained in Section V. Instead of hashing, a custom paging-like address translation (CIDTable, §V-B) is implemented to provide fast and low memory overhead CID to memory address translation with a per-object read-write lock for strong consistency on multi-threaded applications (§V-B). Batch processing of multiple chunks per operation is supported to increase overall throughput. A concurrent memory defragmentation is addressing external fragmentation relevant for long-running applications.

DXMem provides a modular interface implementing different types of operations. It implements CRUD operations (*create, read (get), update (put) and delete (remove)*) which are typically used in key-value store APIs. All operations can be executed with batches of data to reduce processing times further. DXMem also implements operations to *lock* and *resize* existing objects, and to allow *pinning* of chunks for direct local access and RDMA operations with InfiniBand. DXMem is used with DXRAM but also published as a separate library at Github [6].

Our previous publication [21] presented the initial implementation for a low-overhead memory allocator and fast local object lookup (CIDTable). However, the initial proposal with its arena management using coarse-grained locks was unsuitable for highly concurrent applications. Neither it provided a Java object serialization interface nor locking of individual chunks and only implemented the basic CRUD operations. Batching of operations as well as an optimized CID translation algorithm was not available in the initial design.

V. Manual Memory Management in Java

In Java, memory management becomes a considerable issue when storing millions of small objects. By default, all objects are subject to Java's runtime garbage collection. It cleans up allocated but unreferenced objects automatically which has valuable benefits in most standard Java applications. However, when storing billions of small 32-byte objects, which are typical in our target application domain (§II), object instantiation becomes time-consuming, and the per-object metadata

required for storing all objects is relatively high (e.g. 12 byte header on a 64-bit Hotspot-JVM with compressed object pointers and a heap less than 4 GB [10]). Furthermore, garbage collection runs concurrently to the application, and its activity phases cannot be controlled. Thus, it can impose unintended performance penalties due to high latencies during collection phases. A higher degree of control is necessary to support high loads efficiently.

We address these requirements with our custom allocator (§V-A) which keeps the per-object memory-overhead low and (by default) does not impose any garbage collection. Data can be stored off-heap using Java's DirectByteBuffers (2 GB buffer limit), Apache DirectMemory [1] (retired) or the Unsafe [25] class. The latter uses intrinsics for memory access and is widely used for fast data exchange with native libraries or buffers of native I/O. Furthermore, the size of the allocated area is not limited by the maximum value of a positive Java integer (2^{31}).

Using Unsafe, we created a Virtual Memory Block (VMB) allocating a single continuous memory area (starting with address 0) which is used by our allocator (§V-A). All meta-data and application object-data is stored in the VMB and written/read using the methods provided by the Unsafe class. Chunks are read/written using a custom de/-serialization interface (§V-A).

A. Efficient Memory Allocator for Many Small Objects

To maximize the number of objects to store per node, we address the challenges from the previous Section V with a custom allocator. This allocator is designed explicitly for **low-metadata overhead and handling small objects with average sizes of 16-128 bytes efficiently**.

On initialization, our allocator uses the VMB (§V) to create one large free block which occupies nearly the entire VMB (size configurable). We now use **43 bit pointers** which allows addressing a total of 8 TB of main memory which is sufficient for commodity servers. At the end of the VMB, additional space is reserved for root pointers of the doubly-linked free-block lists.

There are two types of free blocks: *Untracked free blocks* are less than 14 bytes in size and are not tracked using a free block list. *Tracked free blocks* are managed by one of the free block lists. Each entry of this list describes the size of the free block in bytes up to the size of the next entry of the free block list. The lists track specific small free blocks of 14, 24, 36 and 48 bytes as well as all power of two sizes starting with 64 bytes and up to the max size of the VMB. **Blocks are not aligned to 64-bit bounds or multiples of a cache line size to avoid fragmentation.**

Every block is separated by a single byte called a *marker byte*. Each half of a marker byte (4 bits) describes the type (allocated, free tracked or untracked block) of the adjacent block to the left or the right of it. Allocated blocks may contain an additional compacted *length field* that stores a part of the payload's size following it. Free untracked blocks of at least 2 bytes contain a length field describing the block size. Free untracked blocks of at least 14 bytes contain two length fields (one at the front and one at the end of the block) and a pointer to the previous and next element of the free block list it is managed by. **This design allows us to keep the average per-object memory-overhead very low** compared to other memory allocators [21] and systems (§VI-A).

On allocation, the allocator selects a free block using a best-fit strategy. **The block is cut to size required to store the length field and the requested payload size to avoid internal fragmentation.** The remaining part of the free block (if available) is converted to a free block and, if a tracked block, added back to one of the free block lists accordingly. On deallocation, the allocator checks the blocks adjacent to the current one to free and merges it with every non-allocated block to lower external fragmentation. If resulting in a tracked free block, it is added to the appropriate free block list.

Often, applications can issue a single allocation request for multiple blocks of a single size or different sizes, e.g., when loading datasets. **The new allocator supports batch allocations reducing overall memory allocation times.** On a batch allocation, the allocator calculates the total amount of memory required and tries to allocate all blocks for the requested sizes in a single continuous area with a fallback option to single block allocation. The search for free blocks is reduced to a minimum (one) which also lowers fragmentation.

Java objects have to implement a custom serialization interface for reading from (*Importable*) and writing to (*Exportable*) an allocated native memory block. The object to im-/export specifies the primitive fields or im-/exportable objects to de-/serialize which enables efficient binary representations. Nested im-/exportable objects are also supported. **Our custom serialization allows DXMem to execute fast, transparent and low-overhead reading from and writing to native memory** compared to generic serialization, e.g., Java *Serializable*.

B. Low-Latency Address Translation

In distributed applications, dealing with bare memory addresses becomes uncomfortable once the stored data is moved either locally (e.g., defragmentation) or to another remote node (e.g., migration due to hotspots). Thus, hashing is used on many systems (§III) to create an indirection and assign unique IDs to objects. However, this commonly used approach comes with many drawbacks such as high memory overhead [21] or requiring additional processing time for re-hashing entries. We address these issues with a custom low-latency and low memory-overhead address translation.

In DXMem **each chunk is referenced using a 64-bit *chunk ID (CID)***. The upper 16-bits are the *node ID (NID)* of the creator node. The NID is assigned on node startup and allows identification of chunk origins in a distributed setup. The lower 48-bits are a per node creator locally unique value, called a *local ID (LID)*. This value is incremented independently between nodes with each chunk creation.

The *CIDTable*, provides a fast and efficient CID to memory address translation for chunk lookup and retrieval of allocated memory blocks in the allocator. The CIDTable is

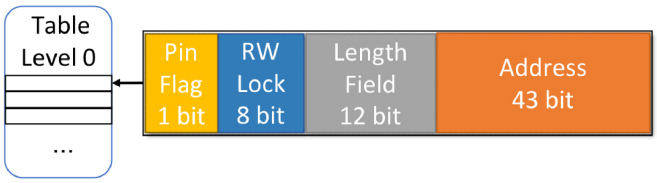

Fig. 2. Breakdown of CIDTable level 0 entry with embedded length field and RW lock.

stored in memory blocks managed by the allocator (§V-A) and consists of multiple sub-tables/pages. On chunk creation, a new CID is generated by using the current node's NID and incrementing an atomic local LID counter. After allocating a memory block for the chunk, the CID to (native) memory address mapping is inserted into the paging-like translation tables. On chunk access operations like get or put, the CID is translated to the stored memory address.

The CID is split into 5 fragments: a 16-bit NID fragment, and four 12-bit LID fragments. The translation steps are executed identically to memory paging on operating systems. The 43-bit memory address referencing the chunk data in the VMB is stored as part of the 8-byte entry in the level 0 table. Each table for a 12-bit LID fragment contains 4096 entries, and the table for the NID fragment contains 65,536 entries. When storing millions of objects, **the average per-object overhead added by the CIDTable is about 8 bytes**. Tables are not pre-allocated but created on demand to save memory. Once created, the tables are never deleted and profit from re-using LIDs (of deleted chunks) on create operations to keep all tables densely packed.

Furthermore, **all tables are aligned to 64-bit bounds to avoid multiple memory reads on misalignment**. This alignment becomes a very critical performance issue especially on CAS operations used for lock implementations (see next paragraph). The alignment does not increase the overall meta-data overhead significantly when considering the overall large size of a table (8 bytes per entry $\times$ 4096 entries = 32 kB).

For this section, this concludes the fundamental design of the CIDTable which has already been published in our previous publication [21]. The following Section V-C presents the extensions and optimizations which, like the fundamental design, are not just limited to the Java environment.

C. Fine-Granular Locks

Figure 2 shows the breakdown of a single CIDTable level 0 entry. 8 bits of the table level 0 entries are used to implement a **custom per chunk read-write lock based on CAS operations**. The level 0 table must be memory aligned to ensure low-latency on lock operations. This lock allows applications to lock single chunks on concurrent reads/writes to protect the payload from data races. By default, the CIDTable does not enforce any locking that prevents payload race conditions on the chunk data. For create-once read-only data (e.g., static index or entry point to sub-structures), **special read-only chunks can be created which do not use any locking mechanism** further decreasing access times. Furthermore, it is possible to ***pin*** **chunks which returns a pointer to the memory address** of the payload stored in the VMB. Locally, a pinned chunk can be accessed directly, or it can be registered with an InfiniBand HCA to allow RDMA reads from and writes to the payload.

To lower the overall memory overhead, we use up to 12 bits to directly embed the length field of allocated blocks in table level 0 entries. This embedded length field enables storing chunks with a maximum size of 2048 bytes without requiring an additional length field in the allocated blocks. If this size is exceeded, a split length field is stored in both the entry of the CIDTable and the allocated block. This design reduces the overall metadata overhead, especially for small chunks but **also speeds up chunk-data access** because no additional length field must be read from the allocated block area.

Often, batches of chunks, especially on graph-based applications with typical access patterns to adjacent vertices/edges with likely close-by chunk IDs, are requested and require multiple translations of nearby CIDs (e.g., on multi-get operations). To benefit from these access patterns, initially, we implemented an additional small thread local cache storing the 10 most recently translated level 0 table addresses using a simple list. This cache allowed subsequent translations to skip four translation steps if it was part of the same table level 0. This cache was speeding up translation by up to 20% on large batches with nearby CIDs, initially. However, after introducing various optimizations to the translation algorithm and the table alignment, **the costs for the cache lookup outweighed the costs of a full translation**. We determined that the translation cache added an average latency of approx. 5 µs not benefiting the overall low access times of the other components involved (see the breakdown in Section §VI-B). Thus, the translation cache was dropped.

VI. Evaluation

The following sections present the results of the following evaluations: the **local metadata overhead** of the memory management, the **local multi-threading performance** and **distributed performance**. We compare **DXMem**, used by **DXRAM**, to **Hazelcast Enterprise with HD memory** and **Infinispan**, two open-source Java-based in-memory caching systems. All systems share fundamental concepts, like providing a local native memory backend using Unsafe, but are suitable candidates because they differ regarding the allocator, address translation and handling of data in a distributed setup. These differences also apply to the other systems presented in Section III which we consider for future evaluations.

Our previous publication [21] already evaluated the memory allocator regarding memory overhead, the CIDTable regarding performance and in a distributed setup DXRAM regarding memory overhead. Due to significant changes in the design of the allocator and CIDTable, this publication re-evaluates DXMem's memory overhead and performance. DXRAM and

the Yahoo! Cloud Serving Benchmark (YCSB) is used to demonstrate DXMem's application in a distributed setup.

The benchmarks in Sections VI-A and VI-B were executed on one machine running Arch-Linux with Kernel version 4.18.10 with 64 GB RAM and an Intel Core i7-6900K CPU with 8 cores (HT) clocked at 3.2 GHz. For the benchmarks in Section VI-C, we used up to 32 servers of our university's cluster, each equipped with 128 GB RAM and two Intel Xeon Gold 6136 (3.0 GHz) 12 core CPUs connected with Gigabit Ethernet. For all benchmark runs, we limited the applications to run on the main CPU socket (with a total of 12 cores), only. The nodes run CentOS 7.4 with the Linux Kernel version 3.10.0-693. All benchmarks are executed with Java 1.8.

We used the YCSB [17] and its workload models for Sections VI-B and VI-C with the following **workloads based on real-world applications**:

1) YCSB-A: Objects with 10x 100 byte fields, 50% get 50% put access-distribution [17]
2) Facebook-B: Objects with 1x 32 byte field, 95% get 5% put access-distribution [27]
3) Facebook-D: Objects with 24x 32 byte fields, 95% get 5% put access-distribution [27]
4) Facebook-F: Objects with 1x 64 byte fields, 100% get access-distribution [16]

A. *Local Metadata Overhead*

In this section, we evaluate the **local memory overhead** of the memory managers of the three systems. The overhead describes the **additional amount of metadata added by the allocators as well as the lookup mechanisms** (e.g., hash table) to store the objects on a single server instance. Our previous publication [21] compared only the memory overhead of our allocator to other commonly used memory allocators available. Hazelcast Enterprise with HD memory and Infinispan are configured to use the off-heap backend storage. Any type of local data replication or eviction (if available) is disabled on all systems. All systems store all data in RAM loaded once. For each benchmark run, we started a single server storage instance and a YCSB loading client. After the load phase is finished, we used Hazelcast's health monitor, Infinispan's exposed bean objects on the JMX interface and DXRAM's storage monitoring data to determine the total amount of data stored in each system. All three systems were loaded with 100 million objects per benchmark run. Each run used one of the power-of-two sizes of 8 to 256 bytes. Larger object sizes were also tested but required a decreased object count to fit into the limited main memory. On some systems, this leads to inconsistent results diverting from the results of 8 to 256-byte objects and could not be extrapolated. Thus, these results were omitted from the evaluation.

The results are depicted in Figure 3 grouping the systems per object size (x-axis) as stacked bars indicating memory consumption (left y-axis). Each group consists of the three systems distinguished by markings (none or diagonal stripes). The line plot is colored accordingly for each system and shows the memory overhead in percent (right y-axis). The results

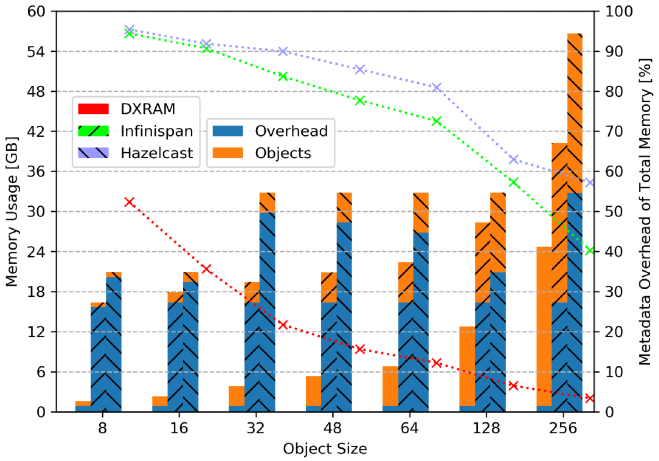

Fig. 3. Memory usage (object payload + metadata) and overhead of DXRAM/DXMem, Hazelcast and Infinispan with increasing object size.

show that **DXRAM achieves an overall very low metadata overhead on small object sizes**. Considering 32-byte objects (total object memory of 2.98 GB), DXRAM adds about 0.84 GB (22% overhead of total used memory), Infinispan about 16.4 GB (85% overhead of total used memory) and Hazelcast about 29.8 GB (91% overhead of total used memory) of memory for metadata. In comparison to DXRAM, Infinispan requires four-times and Hazelcast even 7.5-times more memory to store the same amount of data. Furthermore, Infinispan, as well as **DXRAM, provide constant overhead independent of the object size** for the evaluated object sizes compared to Hazelcast's which seems to be tied to object size ranges as it varies highly.

For DXMem, every object with a size up to 2048 bytes stored adds 8 bytes (omitting table alignment) for a CIDTable entry and one byte in the allocator for the marker byte, only. For objects larger than 2048 bytes, additional bytes with increasing object size (up to 3) are required to extend the length field to store larger objects (§V-B). Infinispan stores the data in an off-heap bucket of linked-list pointers similar to a standard Java *HashMap*. A server adds a fixed overhead which depends on the number of objects to store and adds another 8 bytes for a pointer per-object. Furthermore, variable per allocated object overhead of 25 bytes for header information and a linked-list pointer is added as well as another 36 bytes for additional housekeeping for the LRU list nodes [9]. Hazelcast's off-heap storage uses either a standard or pooled allocator. The standard allocator uses the OS's memory manager (malloc/free from glibc) and is less suitable for many small objects. The pooled allocator (used here) is Hazelcast's recommended custom allocator and uses a buddy allocation policy and 4 MB (default) page size. Block sizes for allocations are always rounded up to the nearest power-of-2 size. Additional metadata space is reserved for map components such as indices or offsets. According to the authors, it takes about 12.5% of the total native memory configured by default [2].

B. Multi-threaded Local Memory Access

This section presents the results of our custom benchmark to determine the **local multi-threading performance of the memory management under high loads on one server**. It adapts the design of the YCSB and its workload model and provides a similar database-layer interface to implement benchmark clients for different systems. It allows execution of an arbitrary amount of phases (common loading and benchmark phases) using create, read, update and delete operations.

For this evaluation, we execute a single load phase followed by one benchmark phase like the original YCSB. All three systems support local code execution on storage nodes and implement the client interface of the benchmark. Again, all systems store the loaded data in native memory and use a mechanism (e.g., hash table) to translate the key to an address for retrieving the value-data from native memory. The CRUD-operations of DXRAM are directly mapped to the benchmark client interface. For Hazelcast and Infinispan, the clients use a single (default) map for the operations (identical to their YCSB clients). To implement object serialization, we used DXRAM's *Importable/Exportable* interface, Hazelcast's *DataSerializable* and Infinispan's *AdvancedExternalizer*. The load phase creates and stores 10 million objects on each storage instance. The run phase executes 100 million operations for each workload.

The results for the different workloads are depicted in multiple figures: YCSB-A in Figure 4, Facebook-B in Figure 5, Facebook-D in Figure 6 and Facebook-F in Figure 7. The results for all systems are grouped per thread count (x-axis). Each group for each system (DXRAM, Infinispan, Hazelcast) for a specific thread count consists of two bars (left bar for get and right bar put operations) on all workloads except the Facebook-F (one bar with get operations, only). The bars of each group have identical markings (none or diagonal stripes). Each stacked bar depicts the operation's average latency as well as the 95th, 99th and 99.9th percentiles (color coded) on a logarithmic scale (y-axis on the left). The aggregated throughput (y-axis on the right) of each system (color coded) is presented as a line plot on the right next to each bar group.

Due to space constraints and the similarity of the results, we depict only the results of the Facebook-B workload for a detailed analysis. The results of the Facebook-B workload show that **DXRAM achieves very low single-digit microsecond average latencies** with up to 32 threads on local storage access. As to be expected, once over-provisioning the 8 core (HT) CPU, the average latency approx. doubles for both gets and puts when doubling the number of threads.

However, the 95th and 99th percentiles are lower (not visible in the figure) than the average latency, and the 99.9th percentiles are increasing significantly when highly over-provisioning the CPU (e.g., 64 and 128 threads). Thus, most operations (99.9%) are executed in less than the average latency, but a small amount (0.1%) have significantly higher latency. With 10 million objects and only 5% write distribution on 100 million operations, the likelihood of lock contention due to a thread write-locking a chunk and getting evicted while

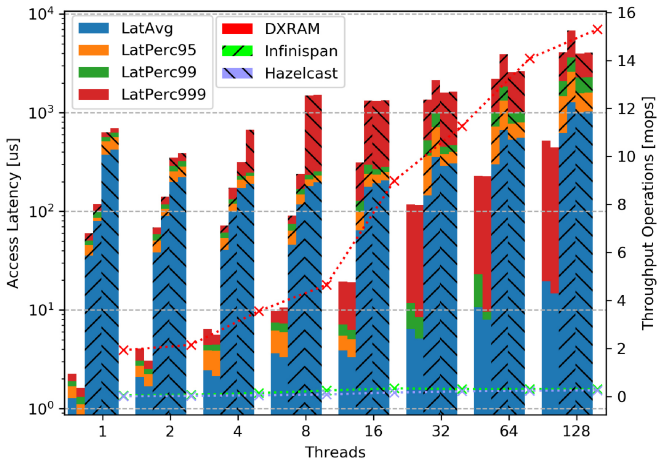

Fig. 4. Local memory multi-threading benchmark using workload YCSB-A with increasing thread count.

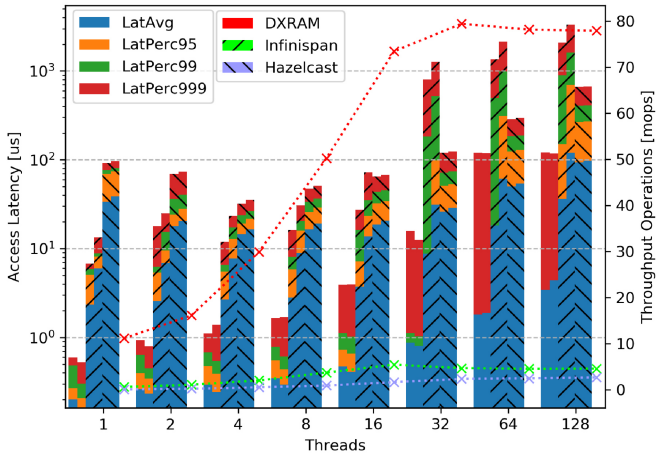

Fig. 5. Local memory multi-threading benchmark using workload Facebook-B with increasing thread count.

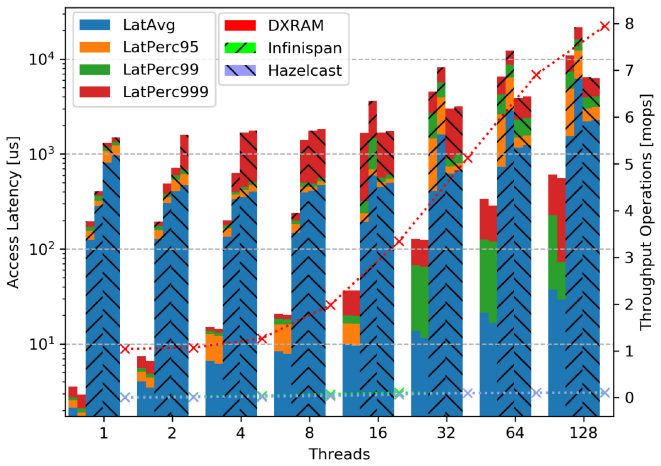

Fig. 6. Local memory multi-threading benchmark using workload Facebook-D with increasing thread count.

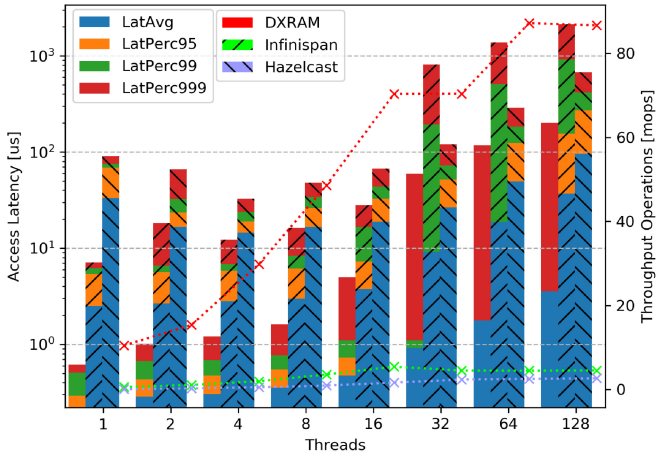

Fig. 7. Local memory multi-threading benchmark using workload Facebook-F with increasing thread count.

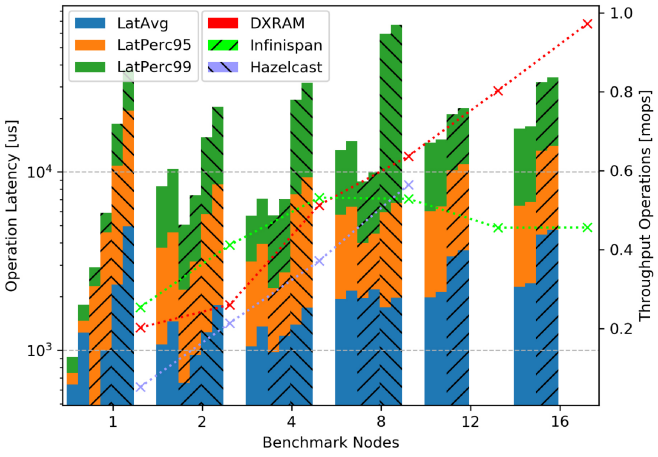

Fig. 9. Workload YCSB-A with increasing benchmark node count accessing an equal amount of storage nodes with 128 threads per node

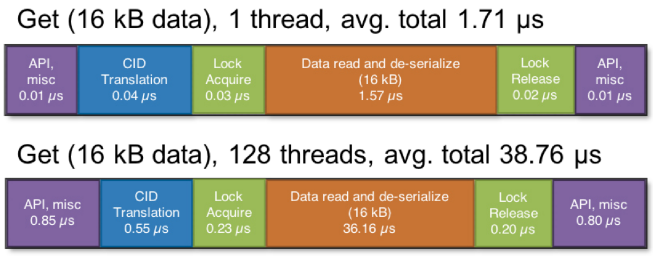

Fig. 8. Breakdown of get-operation (16 kB data) with 1 thread (low load) and 128 threads (high load) in DXRAM

other threads trying to read/write the same object is rather low. For verification, we executed the same workload with 100% read distribution and received similar results eliminating lock contention. We assume that there is another state when a thread is evicted that either increases its own or the latency of other threads which requires further analysis.

With increasing thread count, **DXRAM achieves a throughput saturating at 78 mops** with 32 threads. Compared to Infinispan, peaking at approx. 4.6 mops, and Hazelcast peaking at approx. 2.7 mops, this is a **17-fold and 28-fold increase**. With just a single thread, DXRAM already achieves 11 mops which are more than twice the peak throughput Infinispan and four times the peak throughput of Hazelcast.

For additional reference, Figure 8 depicts a breakdown of a get-operation with average times of the sub-components involved (16 kB of data and running with 1 and 128 threads). The breakdown shows that the majority of time is spent on reading and de-serializing data. **The CID translation and locks require just tenth's of nanoseconds** with a single thread and are still far below one microsecond even when over-provisioning the CPU with 128 threads.

C. Key-Value Storages

In this section, we present the results of the full systems evaluated in a distributed environment with up to 32 nodes using the YCSB with the defined workloads (§VI). The key to overall good performance is the combination of a low-latency and multi-threading capable local memory and network subsystem. All network subsystems are limited to Ethernet-based transports to provide equal conditions. Every workload on every system loads 1 million objects per node during the load phase. The run phase executes 100 million operations per node with Zipfian distribution on each workload. On long-running benchmark runs, we reduced to the number of operations to limit runtime to about 10 minutes to avoid unnecessary long-running benchmark runs. The Hazelcast Enterprise trial license (required for HD memory) we received limited our benchmarks to up to 8 servers nodes.

Half of the nodes on each benchmark are used for storage, and the other half runs remote clients. When referring to a specific number of nodes, we always consider that number of nodes for both servers and clients each, e.g., 4 nodes: 4 servers + 4 benchmark clients = 8 nodes total. The results for the different workloads are depicted in multiple figures: YCSB-A in Figure 9, Facebook-B in Figure 10, Facebook-D in Figure 11 and Facebook-F in Figure 12. The structure of the charts is similar to the ones presented in the previous Section VI-B. The results for all systems are grouped per node count (x-axis). The x-axis depicts the number of benchmark nodes and server nodes each used for a benchmark run (e.g., 4 benchmark nodes on the x-axis ran against 4 server nodes which equal 8 nodes in total). Each group for each system (DXRAM, Infinispan, Hazelcast) for a specific benchmark node count consists of two bars (left bar for get and right bar put operations) with identical markings (none or diagonal stripes). Each stacked bar depicts the operation's average latency as well as the 95th and 99th percentiles (color coded) on a logarithmic scale (y-axis on the left). The aggregated throughput (y-axis on the right) of each system (color coded) is presented as a line plot on the right next to each bar group.

Due to space constraints, we focus on the most notable aspects of the results. The results of the Facebook-B workload

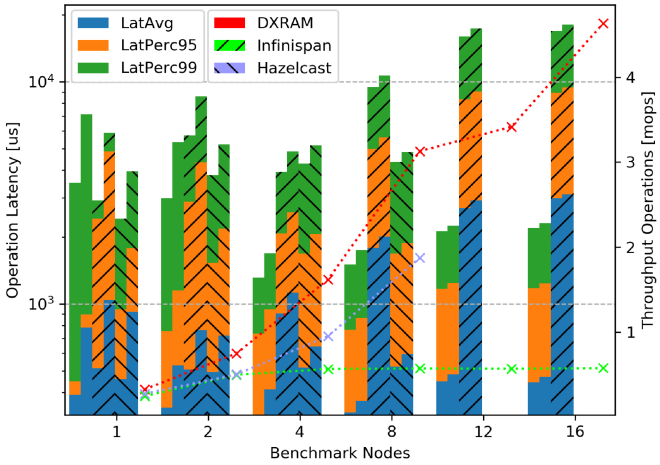

Fig. 10. Workload Facebook-B with increasing benchmark node count accessing an equal amount of storage nodes with 128 threads per node

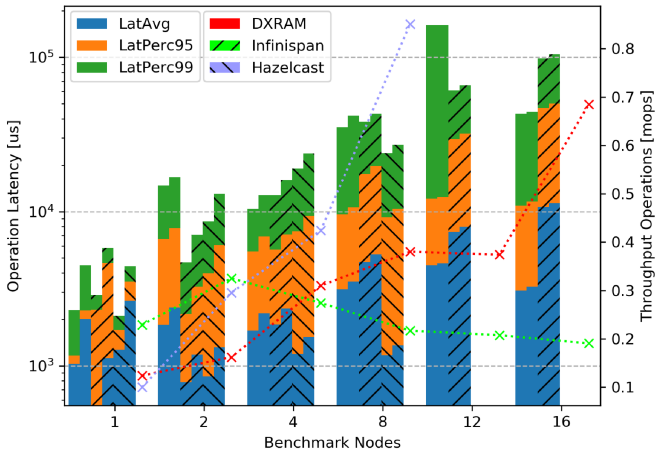

Fig. 11. Workload Facebook-D with increasing benchmark node count accessing an equal amount of storage nodes with 128 threads per node

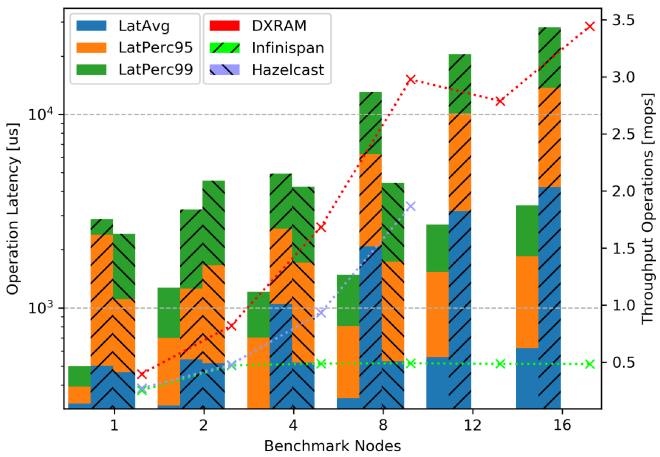

Fig. 12. Workload Facebook-F with increasing benchmark node count accessing an equal amount of storage nodes with 128 threads per node

(Figure 10) show that **DXRAM is capable of efficiently handling tiny objects and scales well with up to 16 nodes** with a total aggregated throughput of 4.6 mops. DXRAM's latencies are higher, especially the 99th percentiles, when using only 1-2 nodes compared to the other systems, but improve on larger scales and drop considerably starting with 4 nodes. Overall, Hazelcast scales well with up to 8 nodes, but its aggregated throughput (1.9 mops) is up to a factor of 1.6 lower compared to DXRAM's (3.1 mops). Infinispan's overall scalability is very limited with an aggregated peak performance of 0.57 mops reached at 8 nodes without further increasing with up to 16 nodes. The results of the YCSB-A reference workload (Figure 9) further support the assumption that Infinispan does not perform and scale well on workloads with tiny objects. Overall similar results for all systems are achieved on the Facebook-F workload (Figure 12).

However, DXRAM shows deficits on the Facebook-D workload (Figure 11). With 1-2 nodes, DXRAM is outperformed by the other systems. However, overall, DXRAM aims for scalability instead of high single node performance and continues to scale with up to 16 nodes, compared to Infinispan whose performance degrades starting with 4 nodes. Hazelcast shows great performance and scalability with up to 8 nodes peaking at an aggregated throughput of 0.85 mops compared to DXRAM with 0.38 mops and Infinispan with 0.21 mops. As the Facebook-D workload uses objects with 24 fields compared to the single field workloads Facebook-B, Facebook-F and the 10 field workload YCSB-A, we conclude that DXRAM requires additional optimizations to handle such a large number of fields better. The YCSB-A workload (Figure 9) shows that DXRAM can already handle 10 fields well. Furthermore, certain results (e.g., 1-2 nodes) across multiple workloads indicate that DXRAM's handling of concurrency on high loads still requires additional optimizations which we are planning to address by improving local thread management.

VII. Conclusions

We presented the extension and optimization of DXMem, a low-latency, and low metadata-overhead memory management for DXRAM, a highly concurrent Java-based distributed in-memory system. DXMem's allocator is designed for storing many small (32 - 128 byte) objects efficiently. The address translation table is extended by a per chunk low-latency read-write lock giving applications a fine-grained synchronization mechanism to control data races. Additionally, chunks can be pinned for direct memory access either locally or for RDMA operations using InfiniBand hardware. We compared DXRAM with DXMem, for the first time, to the two state-of-the-art Java-based key-value caches Hazelcast and Infinispan. Regarding memory overhead, DXMem achieves an at least four-times lower memory overhead on an average object size of 32 bytes compared to the other systems. Using real-world-based workloads to evaluate the local storage performance, DXRAM provides single-digit microsecond latency when not over-provisioning the CPU and outperforms Hazelcast and Infinispan with 78 mops on 128 threads 17-fold and 28-fold.

Using the YCSB in a distributed environment, DXRAM scales well on workloads with small objects and up to 16 server and 16 benchmark clients outperforming Hazelcast (1.6 fold) and Infinispan (5.4 fold). On a read-heavy workload with 32-byte objects, DXRAM achieves an aggregated throughput of 4.6 mops.

REFERENCES

[1] Apache DirectMemory. https://directmemory.apache.org/.

[2] Introduction to Hazelcast HD Memory. https://blog.hazelcast.com/introduction-hazelcast-hd-memory/.

[3] Apache ignite - database and caching platform. https://ignite.apache.org/.

[4] Cassandra. https://cassandra.apache.org/.

[5] Gemfire - in-memory data grid powered by apache geode. https://pivotal.io/pivotal-gemfire.

[6] Github operating systems research group heinrich-heine-university düsseldorf. https://github.com/hhu-bsinfo/.

[7] Hazelcast - an in-memory data grid. https://hazelcast.com.

[8] Infinispan. http://infinispan.org/.

[9] Infinispan - data container changes part 2. https://blog.infinispan.org/2017/01/data-container-changes-part-2.html.

[10] The java hotspot performance engine architecture. https://www.oracle.com/technetwork/java/whitepaper-135217.html.

[11] Official ehcache website. http://www.ehcache.org/.

[12] Sharding in hazelcast. https://docs.hazelcast.org/docs/latest-dev/manual/html-single/#sharding-in-hazelcast.

[13] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.

[14] K. Beineke, S. Nothaas, and M. Schoettner. High throughput log-based replication for many small in-memory objects. In *IEEE 22nd International Conference on Parallel and Distributed Systems*, pages 535–544, 2016.

[15] K. Beineke, S. Nothaas, and M. Schöttner. Efficient messaging for java applications running in data centers. In *International Workshop on Advances in High-Performance Algorithms Middleware and Applications (in proceedings of CCGrid18)*, 2018.

[16] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8:1804–1815, Aug. 2015.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[18] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Incremental page rank computation on evolving graphs. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 1094–1095, 2005.

[19] A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 902–903, 2005.

[20] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47:98 – 115, 2015.

[21] F. Klein, K. Beineke, and M. Schoettner. Memory management for billions of small objects in a distributed in-memory storage. In *IEEE Cluster 2014*, September 2014.

[22] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. In *NetDB 2011: 6th Workshop on Networking meetsDatabases*, 2011.

[23] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, 2010.

[24] X. Liu. Entity centric information retrieval. *SIGIR Forum*, 50:92–92, June 2016.

[25] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: The java unsafe api in the wild. *SIGPLAN Not.*, 50:695–710, Oct. 2015.

[26] S. Mehta and V. Mehta. Hadoop ecosystem: An introduction. In *International Journal of Science and Research (IJSR)*, volume 5, June 2016.

[27] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[28] Oracle. Oracle coherence. https://www.oracle.com/technetwork/middleware/coherence/overview/index.html.

[29] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, November 1999. Previous number = SIDL-WP-1999-0120.

[30] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. sson, D. van der Spoel, B. Hess, and E. Lindahl. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29:845–854, 2013.

[31] A. Ribeiro, A. Silva, and A. R. da Silva. Data modeling and data analytics: A survey from a big data perspective. *Journal of Software Engineering and Applications*, 8:617–634, 2015.

[32] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *CoRR*, abs/1111.4503, 2011.

[33] X. Wu, X. Zhu, G. Q. Wu, and W. Ding. Data mining with big data. *IEEE Transactions on Knowledge and Data Engineering*, 26:97–107, Jan. 2014.

[34] P. Zhao, Y. Li, H. Xie, Z. Wu, Y. Xu, and J. C. Lui. Measuring and maximizing influence via random walk in social activity networks. pages 323–338, Mar. 2017.

Chapter 5

Leveraging High-Speed and Low-Latency Networks in Java Applications

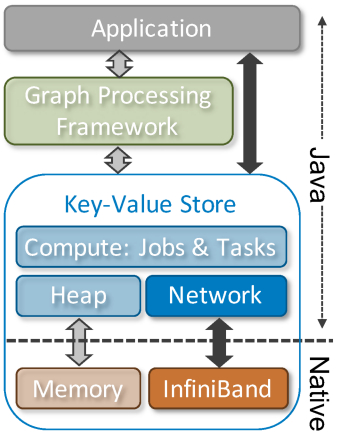

Figure 5.1: The "big picture" of this thesis with the relevant components for this chapter highlighted ("Application" layer and interface, "Network" Java, and "InfiniBand" Native).

This chapter discusses the third and final research question "Can the network support graph-based applications efficiently regarding low latency and handling of many small messages even on highly concurrent random remote access?" in a Java environment (see Section 1.3).

First, Section 5.1 presents the addressed requirements concerning Java applications and low-latency remote communication that result from the above research question based on the previously introduced context (see Section 1.1) and its challenges (see Section 1.2). The stage of work of the DXRAM storage system is presented in Section 5.2, with Section 5.3 elaborating on the major research question in detail. The resulting contributions of this work are presented in Section 5.4 followed by copies of the publications:

Stefan Nothaas, Fabian Ruhland and Michael Schöttner. "A Benchmark Suite to Evaluate InfiniBand Solutions for Java Applications". Published on arXiv e-prints. October 2019. arXiv:1910.02245. 10 pages.

Kevin Beineke, Stefan Nothaas and Michael Schöttner. "Efficient Messaging for Java Applications running in Data Centers". In Proceedings of the 18th International Symposium on Cluster, Cloud and Grid Computing (CCGRID). 2018. 10 pages. Copyright 2018 IEEE. `https://ieeexplore.ieee.org/document/8411076`

Stefan Nothaas, Kevin Beineke and Michael Schöttner. "Leveraging InfiniBand for Highly Concurrent Messaging in Java Applications". In Proceedings of the 18th International Symposium on Parallel and Distributed Computing (ISPDC). 2019. Copyright 2019 IEEE. `https://ieeexplore.ieee.org/document/8790899`

Stefan Nothaas, Kevin Beineke, Michael Schoettner. "Ibdxnet: Leveraging InfiniBand in Highly Concurrent Java Applications". Published on arXiv e-prints. December 2018. arXiv:1812.01963. 31 pages.

5.1 Requirements

By benefiting from data locality on single servers, algorithms can avoid having to request data stored on a remote server. However, keeping high data locality in a distributed environment is a difficult task. For example, temporal locality could be achieved by caching or replication of remote objects to the current server but is difficult to implement efficiently due to the highly random access in our application domain. Spatial locality can be exploited on the application level, e.g., for neighbors of a vertex of a graph, either by pre-fetching, migrating or keeping neighboured data stored on the same server. However, exploiting locality on the network level efficiently without any application context is not possible.

Huge data volumes require a distributed approach aggregating many servers. Naturally, this results in slower remote data exchange. More servers offer more compute resources but require fast and efficient communication to run and coordinate distributed computations.

The efficiency of a distributed approach depends on the network subsystem for communication with remote servers. Considering the requirements of our application domain (see Section 1.2), most data transferred is rather small due to the majority of objects being small. Such small transfers can be avoided with batching of data or operations, but in general, are limited to the possibilities offered by the algorithm. The highly random access patterns on graph algorithms result in random access to many remote servers. Data locality can limit but not avoid this entirely. Naturally, remote access increases with the degree of data distribution (number of servers).

As batching improves the overall throughput of offline analytics, it increases latency. To the contrary, online applications with user interactions demand low latency and instead neglect

throughput. Thus, from a networking perspective, the network subsystem has to provide, both, low latency and high throughput to support different application focuses.

Many threads can run on a single server for concurrent processing, but potentially have to exchange data with remote servers. The more a concurrent algorithm has to coordinate, the more communication with remote servers is required. Thus, the network subsystem must be capable of handling high concurrency with many threads sending and receiving data simultaneously. Naturally, this requirement does not account for embarrassingly parallel applications (e.g., MapReduce) with limited to none remote communication.

Often, the data transferred consists of objects of the stored data-set. Thus, serialization is required to convert the objects to a binary format for inter-server transfer. Furthermore, higher-level primitives for request-response patterns aid in implementing coordination of multiple servers.

To provide very low-latency remote data access, modern hardware such as InfiniBand offering single-digit microseconds latency is mandatory. All requirements mentioned thus far have to be considered in the context of low-latency hardware usage to leverage InfiniBand for highly concurrent messaging in Java. Furthermore, additional constraints imposed by the Java environment (e.g., interaction with native code and memory) have to be considered when programming such low-level hardware.

5.2 Stage of Work

To address the requirements presented in Section 5.1, the research regarding low-latency remote communication in Java applications in this thesis was conducted in the context of the Java-based DXRAM storage system. Initially designed and developed by Dr. Florian Klein, DXRAM's network subsystem was limited to Ethernet networks, only. It already implemented an API with higher-level communication primitives for asynchronous and synchronous messaging. Messaging objects were de-/serialized using a built-in and simple field-by-field to ByteBuffer serialization. Marc Ewert designed and implemented the initial concurrent back-end in his master thesis [34] and the network subsystem was deeply integrated into DXRAM.

5.3 Research Questions

With InfiniBand available in HPC and even in cloud environments, Java applications, like DXRAM, would benefit highly from low remote access-latency. However, how to utilize this hardware in Java, especially in the context of a key-value storage? Which solutions are already available to use InfiniBand hardware in Java applications? With such low latency on the hardware level, designing software that can truly leverage the power of the hardware is very challenging and raises the question if existing Java solutions are sufficient to exploit the performance of the hardware.

DXRAM, as well as most other Java-based in-memory systems, already provide a network subsystem based on Ethernet networks (see Section 2.2). With various socket-based wrappers available, we first analyzed transparent solutions and if they can provide adequate performance (see Section 5.4.1).

Our results show that these solutions are far from optimal compared to using custom verbs-based implementations. Furthermore, evaluations of DXRAM and the network subsystem pointed out significant weaknesses regarding the handling of small messages, serialization and buffer management, concurrency management and scalability. Naturally, this is not an optimal foundation to build support for low-latency hardware on. The API of the network submodule was already well designed and fitting our target application domain. However, the back-end required a re-design to be capable of leveraging the performance of InfiniBand hardware. However, we do not want to replace Ethernet with InfiniBand, but instead, support both interconnects (see Section 5.4.2).

By supporting two fundamentally different programming models, sockets and verbs, this raised many fundamental questions regarding the network subsystem: How to design the processing pipeline and data structures of the back-end to support these two fundamentally different programming models without negative impact on performance? How to design a shared abstraction layer that allows both models to achieve maximum performance? How to adapt the existing higher-level primitives?

Because InfiniBand hardware cannot be accessed directly in Java, these requirements are very challenging (see Section 5.4.1). When using the native verbs API, the Java-based network subsystem has to utilize a native C-library to communicate directly with InfiniBand hardware. This approach raises further questions regarding the performance as the design of the pipeline is very latency sensitive: How to design a low-latency processing pipeline for InfiniBand communication spanning from Java to native space and vice versa, and utilizing the native verbs API? This pipeline has to consider scalability on a local thread level as well as regarding many remote connections.

5.4 Contributions

This section presents the contributions addressing the previously stated research questions and how they were implemented with DXRAM's network subsystems DXNet. The contributions are published in several publications and described in the following sections. Section 5.4.1 presents the "Java InfiniBand Benchmark" suite to evaluate existing solutions to leverage InfiniBand in Java applications. Section 5.4.2 presents DXRAM's re-designed network subsystem DXNet. Section 5.4.3 presents the design of Ibdxnet, the InfiniBand transport for DXNet. Copies of the publications are attached after each section. The contributions stated in the following sections which are not explicitly assigned to any author/contributor are by the author of this thesis.

5.4.1 JIB-Benchmark: A Benchmark Suite to Evaluate Existing InfiniBand Solutions for Java Applications

The proposed publication [94] in this section presents the Java InfiniBand Benchmark (JIB) suite to evaluate existing solutions to use InfiniBand hardware in Java applications. This suite is the result of prolonged work of this thesis. Before InfiniBand development started, Michael Schlapa analyzed available libraries and solutions to use InfiniBand in Java applications in his master thesis [111]. His results served as a foundation for the decision that DXRAM required a custom solution for using InfiniBand because none of the existing solutions achieved satisfying performance. This decision was followed by further research, microbenchmarks, and prototypes by the author of this thesis and resulted in the mandatory re-design of DXRAM's network subsystem for InfiniBand use (see Section 5.4.2).

In general, the benchmarks and results are useful to any Java application that wants to use InfiniBand and, first, has to consider the numerous pros and cons of the available solutions. In his master thesis [108], Fabian Ruhland created the JIB-Benchmark suite which revises the initially proposed benchmarks by Michael Schlapa [111]. Additional knowledge obtained by Stefan Nothaas when developing Ibdxnet (see Section 5.4.3) allowed improving the benchmarks to provide more optimal results. Fabian Ruhland implemented a scripting framework with a fully automated pipeline to run the benchmark suite with all currently implemented benchmarks (overhead, uni-directional throughput, bi-directional throughput, and one-sided latency) and libraries (C-verbs, jVerbs, IPoIB, libvma, JSOR) followed by an evaluation of the data and generating of plots. Stefan Nothaas contributed various bugfixes and optimized Fabian Ruhland's code, refactored the one-sided latency benchmark and implemented an additional ping-pong benchmark.

Fabian Ruhland and Prof. Dr. Michael Schöttner took part in many discussions about the performance analysis of the benchmark results.

Stefan Nothaas wrote the paper and used the benchmark suite to re-evaluate all currently available solutions on 56 Gbit/s and 100 Gbit/s hardware provided by the "Centre for Information and Media Technology" (ZIM) at the University of Düsseldorf. Prof. Dr. Michael Schöttner and Fabian Ruhland reviewed the paper several times. The JIB-Benchmark suite is open source and available at Github [43].

A Benchmark to Evaluate InfiniBand Solutions for Java Applications

Stefan Nothaas
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
stefan.nothaas@hhu.de

Fabian Ruhland
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
fabian.ruhland@hhu.de

Michael Schoettner
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
michael.schoettner@hhu.de

***Abstract*—Low-latency network interconnects, such as InfiniBand, are commonly used in HPC centers and are even accessible with todays cloud providers offering equipped instances for rent. Most big data applications and frameworks are written in Java. But, the JVM environment alone does not provide interfaces to directly utilize InfiniBand networks.**

In this paper, we present the "Java InfiniBand Benchmark" to evaluate the currently available (and supported) "low-level" solutions to utilize InfiniBand in Java. It evaluates socket- and verbs-based libraries using typical network microbenchmarks regarding throughput and latency. Furthermore, we present evaluation results of the solutions on two hardware configurations with 56 Gbit/s and 100 Gbit/s InfiniBand NICs. With transparency often traded for performance and vice versa, the benchmark helps developers with studying the pros and cons of each solution and support them in their decision which solution is more suitable for their existing or new use-case.

***Index Terms*—High-speed networks, Distributed computing**

I. Introduction

RDMA capable devices have been providing high throughput and low-latency to HPC applications for several years [18]. With todays cloud providers offering instances equipped with InfiniBand for rent, such hardware is available to a wider range of users without the high costs of buying and maintaining it [25]. Many application domains such as social networks [20], [29], [31], search engines [24], [36], simulations [37] or online data analytics [21], [41], [42] require large processing frameworks and backend storages. Many of these are written in Java, e.g. big data batch processing frameworks [28], [33], databases [1], [2] or backend storages/caches [3], [4], [7], [35].

These applications benefit from the rich environment Java offers including automatic garbage collection and multi-threading utilities. But, the choices for inter-node communication on distributed applications are limited to Ethernet-based socket-interfaces (standard ServerSocket or NIO) on the commonly used JVMs OpenJDK and Oracle. They do not provide support for low-latency InfiniBand hardware. But, there are external solutions available each with pros and cons.

This raises questions if a developer wants to chose a suitable solution for a new use-case or an existing application: What's the throughput/latency on small/large payload sizes? Is the performance sufficient when trading it for transparency requiring less to no changes to the existing code? Is it worth considering developing a custom solution based on the native API to gain maximum control with chances to harvest the performance available by the hardware?

In this paper, we address these questions by presenting a "Java InfiniBand (JIB) benchmark" to evaluate existing solutions to leverage the performance of InfiniBand hardware in Java applications. The modular benchmark currently provides implementations to evaluate three socket-based libraries and implementations, IP over InfiniBand, libvma and JSOR, as well as two verbs-based implementations, native C-verbs and jVerbs. This paper focuses on the fundamental performance metrics of low-level interfaces and *not* on higher-level network subsystems with connection management, complex pipelines and messaging primitives, e.g. MPI. We discuss and evaluate these in a separate publication [34]. We used our benchmark to evaluate the listed solutions on two hardware configurations with 56 Gbit/s and 100 Gbit/s InfiniBand NICs. The contributions of this paper are:

- An overview of existing Java InfiniBand solutions
- An extensible and open source benchmark to easily evaluate solutions to use InfiniBand in Java applications
- Extensive evaluation of existing Java libraries with 56 Gbit/s and 100 Gbit/s hardware

The remaining paper is structured as follows: Section II discusses related work with socket-based (§II-A) and verbs-based (§II-B) libraries. Section III presents the JIB Benchmark Suite which is used to evaluate two verbs-based solutions and three socket-based solutions in the following Section IV regarding overhead (§IV-A), uni-directional (§IV-B) and bi-directional (§IV-C) throughput, as well as one-sided latency (§IV-D) and full round-trip-time using a ping-pong benchmark (§IV-E). Conclusions are presented in Section V.

II. Related Work

This section elaborates on existing "low-level" solutions/libraries that can be used to leverage the performance of InfiniBand hardware in Java applications. This does not include network or messaging stacks/subsystems implementing higher-level primitives such as the Massage Passing Interface, e.g. Java-based FastMPJ [22] providing a special transport to use InfiniBand hardware. To the best of our knowledge, there is no benchmark available to evaluate InfiniBand solutions in Java.

A. Socket-based Libraries

The socket-based libraries redirect the send and receive traffic of socket-based applications transparently over InfiniBand host channel adapters (HCAs) with or without kernel bypass depending on the implementation. Thus, existing applications do not have to be altered to benefit from improved performance due to the lower latency hardware compared to commonly used Gigabit Ethernet. The following three libraries are still supported to date and evaluated in Section IV.

IP over InfiniBand (IPoIB) [27] is not a library but actually a kernel driver that exposes the InfiniBand device as a standard network interface (e.g. *ib0*) to the user space. Socket-based applications do not have to be modified but use the specific interface. However, the driver uses the kernel's network stack which requires context switching (kernel to user space) and CPU resources when handling data. Naturally, this solution trades performance for transparency.

libvma [10] is a library developed by Mellanox and included in their OFED software package [11]. It is pre-loaded to any socket-based application (using *LD_PRELOAD*). It enables full bypass of the kernel network-stack by redirecting all socket-traffic over InfiniBand using unreliable datagram with native C-verbs. Again, the existing application code does not have to be modified to benefit from increased performance.

Java Sockets over RDMA (JSOR) [40] redirects all socket-based data traffic in Java applications using native verbs, similar to libvma. It uses two paths for implementing transparent socket streams over RDMA devices. The "fast data path" uses native verbs to send and receive data and the "slow control path" manages RDMA connections. JSOR is developed by IBM on only available in their proprietary J9 JVM.

The following libraries are also known in literature but are not supported or maintained anymore.

The **Sockets Direct Protocol (SDP)** [23] redirects all socket-based traffic of Java applications over RDMA with kernel-bypass. It supported all available JDKs since Java 7 and was part of the OFED package until it was removed with OFED version 3.5 [12].

Java Fast Sockets (JFS) [39] is an optimized Java socket implementation for high speed interconnects. It avoids serialization of primitive data arrays and reduces buffering and buffer copying with shared memory communication as its main focus. However, JFS relies on SDP (deprecated) for using InfiniBand hardware.

Speedus [17] is a native library that optimizes data transfers for applications especially on intra-host and inter-container communication by bypassing the kernel's network stack. It is also advertised to support low-latency networking hardware for inter-node communication. But, the latest available version to date (2016-09-08) does not include such support.

B. Verb-based Libraries

Verbs are an abstract and low-level description of functionality for RDMA devices (e.g. InfiniBand) and how to program them. Verbs define the control and data paths including RDMA operations (write/read) as well as messaging (send/receive). RDMA operations allow reading or writing directly from/to the memory of the remote host without involving the CPU of the remote. Messaging follows a more traditional approach by providing a buffer with data to send and the remote providing a buffer to receive the data to.

The programming model differs heavily from traditional socket-based programming. Using different types of asynchronous queues (send, receive, completion) as communication endpoints. The application uses different types of work-requests for sending and receiving data. When handling data to transfer, all communication with the HCA is executed using these queues. The following libraries are verbs implementations that allow the user to program the RDMA capable hardware directly. The first two libraries presented are evaluated in Section IV.

C-verbs are the native verbs implementation included in the OFED package [13]. Using the Java Native Interface (JNI) [30], this library can be utilized in Java applications as well in order to create a custom network subsystem [22] [34]. Using the Unsafe class [32] or Java DirectByteBuffers, memory can be allocated off-heap to use it for sending and receiving data with InfiniBand hardware (buffers must be registered with a protection domain which pins the physical memory).

jVerbs [38] are a proprietary verbs implementation for Java developed by IBM for their J9 JVM. Using a JNI layer, the OFED C-verbs implementation is accessed. "Stateful verb methods" (*StatefulVerbsMethod* Java objects) encapsulate the verb to call including all parameters with parameter serialization to native space. Once the object is prepared, it can be executed which actually calls the native verb. These objects can be re-used for further calls with the same parameters to avoid repeated serialization to native space and creating new objects which would burden garbage collection.

Jdib [26] is a library wrapping native C-verbs function calls and exposing them to Java using a JNI layer. According to the authors, various methods, e.g. queue pair data exchange on connection setup, are abstracted to create an easier to use API for Java programmers. The fundamental operations to create protection domains, create and setup queue pairs, as well as posting data-to-send to queues and polling the completion queue seem to wrap the native verbs and do not introduce additional mechanisms like jVerbs's stateful verb calls. We were not able to obtain a copy of the library for evaluation.

III. A Benchmark for Evaluating InfiniBand Libraries for Java

To evaluate and compare the different libraries available, a common set of benchmarks had to be implemented for two programming languages (C and Java) and two programming models (sockets and verbs). Existing solutions like the iperf [8] tools for TCP/UDP or the ibperf tools included in the OFED package [13] do not cover all libraries we want to evaluate and do not implement all necessary benchmark types.

In this paper, we want to evaluate most of the available and still maintained libraries (§II) in a fundamental point-to-point setup regarding throughput and latency. Like other benchmarks

(e.g. OSU [14]), we want to determine the maximum throughput on uni-directional and bi-directional communication (e.g. application pattern asynchronous "messaging"), as well as one-sided latency and full round-trip-time (RTT) with a ping-pong communication pattern (e.g. application pattern "request-response"). These benchmarks allow us to determine the fundamental performance of the presented solutions and are commonly used to evaluate network hardware or applications [8], [13], [14]. The evaluation of higher-level primitives, e.g. collectives, and all-to-all communication is not possible with fundamental low-level interfaces. These require a higher-level networking stack with connection management and a complex pipeline which is not the focus of this paper.

The Java InfiniBand Benchmark (JIB) provides implementations of the listed benchmarks for two verbs-based solutions (C-verbs, jVerbs) and three socket-based solutions (IPoIB, libvma, JSOR). It is open source and available at Github [9]. Each benchmark run is configurable using command line parameters such as the benchmark type (uni-/ bi-directional, one-sided latency or ping-pong), the message size to send/receive and the number of messages to send/receive. For convenience, we refer to the payload size sent as messages independent of how it is transferred (e.g. sockets, verbs RDMA or verbs messaging). The context and all buffers are initialized before the benchmark is started. Afterwards, the current instance connects to the remote specified via command line parameters. Once the connection is established, a dedicated thread is started for sending data and another thread for receiving. Today, we can expect this to run on a multicore system with at least two physical cores to ensure that the send and receive thread can be run simultaneously to avoid blocking one another. The benchmark instance sends the specified number of messages to the remote and measures the time it takes to send the messages. Furthermore, we utilize the performance counters of the InfiniBand HCA to determine the overhead added by any software defined protocol which is especially relevant for the socket-based libraries (§IV-A).

For socket-based libraries, the benchmark is implemented in Java using TCP sockets with the ServerSocket, DataInputStream and DataOutputStream classes. Sending and receiving data is executed synchronously in a single loop on each thread. No further adjustments are required because all libraries redirect the normal send and receive calls of the socket libraries. With IPoIB, we use the address of the exposed *ib0* device. The libvma library is pre-loaded to the benchmark using *LD_PRELOAD*. In order to use JSOR, we run the benchmark in the J9-JVM and provide a configuration file specifying IP-addresses whose traffic is redirected over the RDMA device.

The verbs-based benchmarks are implemented in C and Java. Both implementations use RC queue pairs for RDMA and message operations. UD queue pairs can also be used for message operations but this option is currently not implemented. On RDMA operations, we did not include immediate data with a work request which would require a work completion on the remote (optional for signalling incoming RDMA operations on the remote). When sending RDMA operations to the HCA to determine the maximum throughput, we do not repeatedly add one work request to the send queue, then poll the completion queue waiting for that single work request to complete. This pattern is commonly used in examples [16] and even larger applications [15] but does not yield optimal throughput because the queue of the HCA runs empty very frequently. To keep the HCA busy, the send queue must be kept filled at all times. Thus, we fill up the send queue to the maximum size configured, first. Then, we poll the completion queue and once at least one completion is available and processed, we immediately fill the send queue again. Naturally, this pattern cannot be applied to the ping-pong latency benchmark executing a request-response pattern.

This data path is implemented in both, the C-verbs and jVerbs implementation. The C implementation uses the verbs implementation included in the OFED package and serves as a reference for comparing the maximum possible performance. To establish a remote connection, queue pair information is exchanged using a TCP socket. The jVerbs implementation has to implement the operations of the data path using the previously described stateful verbs methods. Thus, the sending of data on the throughput benchmark had to be altered slightly. A single stateful verb call for posting work requests to the send queue always posts 10 elements. Hence, new work requests are added to the send queue once at least 10 work completions were polled from the completion queue. We create all stateful verbs calls before the benchmark and re-use them to avoid performance penalties. On connection creation, queue pair information is exchanged with the API provided by jVerbs which is using the RDMA connection manager.

IV. Evaluation

In this Section, we present the results of the evaluation of the socket-based libraries/implementations **IPoIB**, **libvma** and **JSOR** and the verbs-based libraries **C-verbs** and **jVerbs** using our benchmark suite (§III). We analyze and discuss basic performance metrics regarding throughput and latency using typical benchmarks with a two node setup with 56 Gbit/s and 100 Gbit/s interconnects. A summary of the benchmarks executed with each library/implementation is given in Table I. Due to space constraints, we limit the discussion of the results to selected conspicuities of the plotted data.

Library/Benchmark	OV	Uni-dir	Bi-dir	Lat	PingPong
C-verbs rdmaw		x	x	x	
C-verbs rdmar		x	x	x	
C-verbs msg	x	x	x	x	x
jVerbs rdmaw		x	x	x	
jVerbs rdmar		x	x	x	
jVerbs msg	x	x	x	x	x
IPoIB	x	x	x	x	x
JSOR	x	x	error	x	x
libvma	x	x	x	x	x

TABLE I
OVERVIEW OF LIBRARIES EVALUATED WITH BENCHMARKS AVAILABLE. ABBREVIATIONS: OV = OVERHEAD, RDMAW = RDMA WRITE, RDMAR = RDMA READ, MSG = MESSAGING VERBS

We use the term “message” (msg) to refer to the unit of transfer which is equivalent to the data payload. The size of a message does refer to the payload size only and does not include any additional protocol or network layer overhead. Each throughput focused benchmark run transfers 100 million messages and each latency focused benchmark run transfers 10 million messages. Starting with 8 kB message size, the amount of messages is incrementally halved to avoid unnecessary long running benchmark runs. We evaluated payload sizes of 1 byte to 1 MB in power-of-two-increments. When discussing the results, we focus on the message rate on small messages with payload sizes less than 1 kB and on the throughput on middle sized and large messages starting at 1 kB.

The throughput results are depicted as line plots with the left y-axis describing the throughput in million messages per second (mmps) and the right y-axis describing the throughput in MB/s. For the latency results, the left y-axis describes the latency in µs and the right y-axis the throughput in mmps. The dotted lines always represent the message throughput while the solid lines represent either the throughput in MB/s or the latency in µs, depending on the benchmark. For the overhead results, a single y-axis describes the overhead in percentage in relation to the amount of payload transferred on a logarithmic scale. On all plot types, the x-axis depicts the size of the payload in power-of-two increments from 1 byte to 1 MB. Each benchmark run was executed three times and the min and max as well as average of the three runs are visualized using error bars.

The following releases of software were used for compiling and running the benchmarks: Java 1.8, OFED 4.4-2.0.7, libvma 8.7.5, IBM J9 VM version 2.9, gcc 8.1.0. We ran our experiments on the following two setups with two nodes each:

1) Mellanox ConnectX-3 HCA, 56 Gbit/s InfiniBand, MTU size 4096, Bullx blade with Dual socket Intel Xeon E5-2697v2 (2.7 GHz) 12 core CPUs, 128 GB RAM, CentOS 7.4 with the Linux Kernel version 3.10.0-693, SBE-820C with built-in switch
2) Mellanox ConnectX-5 HCA, 100 Gbit/s InfiniBand, MTU size 4096, Supermicro blade with Dual socket Intel Xeon Gold 6136 (3.0 GHz) 12 core CPUs, 128 GB RAM, CentOS 7.4 with the Linux Kernel version 3.10.0-693, Super Micro EDR-36 Chassis with built-in switch

Flow steering must be activated for libvma to redirect all traffic over InfiniBand by setting the parameter *log_num_mgm_entry_size* to *-1* in the configuration file */etc/modprobe.d/mlnx.conf* for the InfiniBand kernel module. Otherwise, libvma falls back to sockets over Ethernet.

In the following subsections, we focus the analysis and discussion on the results with 100 Gbit/s hardware. Selected Figures depicting the results with 56 Gbit/s hardware are also included if they provide additional insights and value for discussion and comparision. Due to automated execution of the benchmarks, the naming in the figures differs slightly, e.g. ”JSocketBench(msg)“ refers to IPoIB. The other names are self-explanatory.

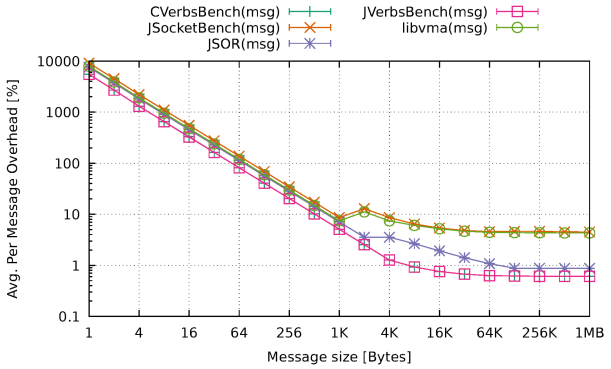

Fig. 1. Average per message overhead in percentage in relation to the payload size transferred.

A. Overhead

In this Section, we present the results of the overhead measurements of the described libraries/implementations. As overhead, we consider the additional amount of data that is sent along with the payload data of the user. This includes any data of any network layer down to the HCA. We measured the amount of data emitted by the port using the performance counter *port_xmit_data* of the HCA.

IPoIB and libvma are implementing buffer/message aggregation when sending data. Applications on high load sending many small messages benefit highly from increased throughput and the overall per message overhead is lowered. However, in order to determine the general per message overhead, we used the pingpong benchmark which does not allow aggregation due to its nature. The results of both types (sockets/verbs) are depicted in a single figure (see Figure 1).

We try to give a rough breakdown of the overhead involved with each method evaluated. A precise breakdown is rather difficult with just the raw amount of data captured from the ports as re-transmission of packages are also captured (e.g. RC queue pairs or custom protocols based on UD queue pairs).

The results in Figure 1 show that the overhead for msg operations of C-verbs and jVerbs are identical. For a single byte of payload, an additional 54 bytes are required which corresponds to two 27 byte headers which are part of the low-level InfiniBand protocol. Required by the RC protocol, one package is used for sending the ping and the other package to receive the pong. The metadata consists of a local routing header (8 bytes), base transport header (12 bytes), invariant CRC (4 bytes) and variant CRC (2 bytes) [19]. This makes a total of 26 bytes which is close to the measured 27 bytes (errors due to *port_xmit_data* yielding values in octets). For RDMA operations, an additional RDMA extended transport header (16 bytes) is added which makes a total of 42 bytes of “metadata“ for such a packet. Naturally, this overhead cannot be avoided. As expected with jVerbs using the native verbs directly without adding another software protocol layer, the overhead added is identical to C-verbs’s. The overhead stays constant which leads to an overall decreasing per message

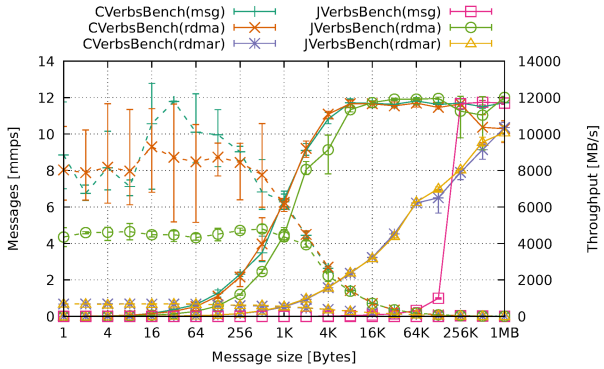

Fig. 2. **Uni-directional** throughput, **verbs-based** libraries with different transfer methods, increasing message size, **100 Gbit/s**.

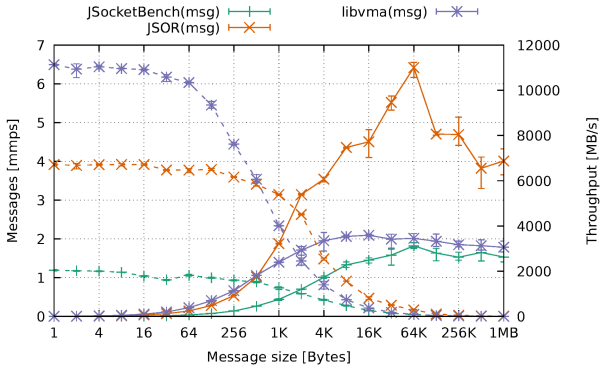

Fig. 3. **Uni-directional** throughput, **socket-based** libraries, increasing message size, **100 Gbit/s**.

overhead with increasing payload size. Starting with 8 kB payload size, the overhead ratio drops below 1%.

The overhead of the socket-based solutions is overall slightly higher. Again, considering 1 byte messages, JSOR adds an additional 7500 %, libvma 7900 % and IPoIB 9100 % overhead to each pingpong transfer. libvma and IPoIB rely on UD messaging verbs which add a datagram extended transport header (8 bytes) to the InfiniBand header and include additional information to allow IP-address based routing of the packages. The IPoIB specification describes an additional header of 4 octets (4 bytes) and IP header (e.g. IPv4 20 bytes + 40 bytes optional) which are added alongside the message payload [27]. libvma adds an IP-address (4 bytes) and Ethernet frame header (14 bytes) [10]. Remaining data is likely committed towards a software signalling protocol. Regarding JSOR, we could not find any information about the protocol implemented (closed source).

B. *Uni-directional Throughput*

This section presents the throughput results of the uni-directional benchmark. Starting with the verbs-based results depicted in Figure 2, jVerbs RDMA write message throughput (4.3 - 4.6 mmps) is about half of C-verbs's RDMA write throughput (7.9 - 9.3 mmps) for small payload sizes up to 512 byte. The RDMA write performance of C-verbs is nearly double the throughput of C-verbs messaging but with high jitter. Starting at 1 kB payload size, jVerbs's RDMA write throughput stays clearly below both C-verbs's RDMA write and message send throughput. Interesting to note that C-verbs's messaging is significantly better, though highly jittery, on small messages up to 512 bytes and middle sized messages up to 4 kB. Both C-verbs operations saturate throughput with 8 kB payload size and low jitter at around 11.7 GB/s. We could not determine the reason for the very poor performance of jVerbs's msg verbs on both 56 Gbit/s and 100 Gbit/s hardware.

The results of socket-based libraries are depicted in Figure 3. On small payload sizes up to 256 bytes, IPoIB achieves a throughput of approx. 1 - 1.2 mmps. With increasing payload size, the throughput starts saturating at 32 kB message size and peaks at 64 kB with 3.1 GB/s throughput. The results of libvma

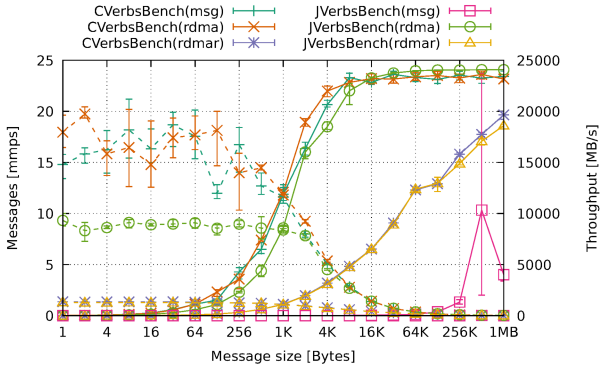

Fig. 4. **Bi-directional** throughput, **verbs-based** libraries with different transfer methods, increasing message size, **100 Gbit/s**.

show an highly increased throughput of 6.0 to 6.5 mmps for up to 64 byte messages. Overall throughput for middle and large sized messages surpasses IPoIB's peaking at 3.5 GB/s with 8 kB messages but also starting to decrease down to 3.0 GB/s when increasing the message size up to 1 MB. JSOR achieves a significantly lower throughput of 3.8 - 3.9 mmps for up to 128 byte messages. However, JSOR provides a much higher throughput starting at 1 kB message size compared to IPoIB and libvma. Throughput peaks at 64 kB message size with 11 GB/s but drops down to approx 6.5 GB/s with 512 kB messages afterwards. As described in Section IV, we determined that JSOR's performance degrades considerable on payload sizes of 128 kB and greater which required us to increase the RDMA buffer size (to 1 MB). However, this problem could not be resolved entirely.

The results on 56 Gbit/s hardware for both, verbs and sockets, show an overall and expected lower throughput but not any notable differences. Thus and due to space constraints, the figures are omitted. But, results for libvma were not available because the benchmarks failed repeatedly with a non fixable "connection reset" error by libvma.

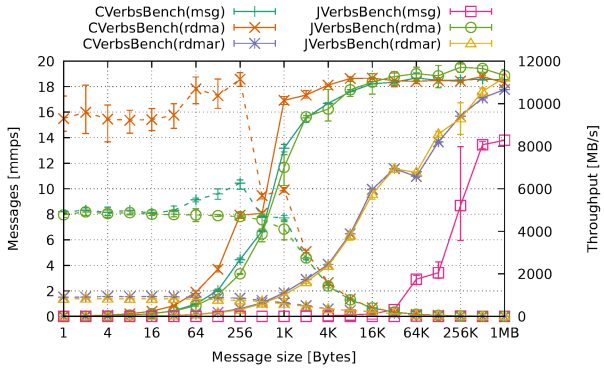

Fig. 5. **Bi-directional** throughput, **verbs-based** libraries with different transfer methods, increasing message size, **56 Gbit/s**.

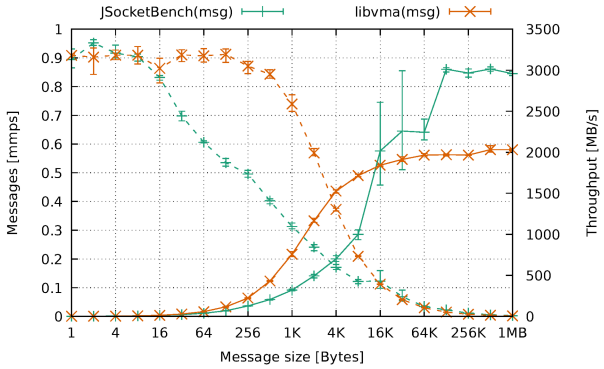

Fig. 6. **Bi-directional** throughput, **socket-based** libraries, increasing message size, **100 Gbit/s**.

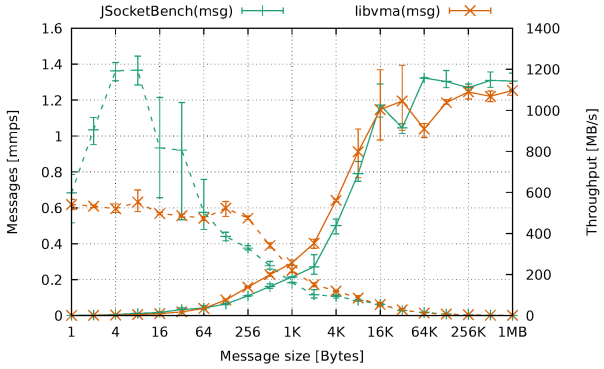

Fig. 7. **Bi-directional** throughput, **socket-based** libraries, increasing message size, **56 Gbit/s**.

C. *Bi-directional Throughput*

This section presents the throughput results of the bi-directional benchmark. With full-duplex communication supported, we expect roughly double the throughput of the uni-directional results in general. Figure 4 depicts the results of the verbs-based implementations and, as expected, all implementations show roughly double the message rate on small messages and roughly double the throughput on large messages compared to the uni-directional results (§IV-B).

C-verbs RDMA writes are still jittery but yield the best performance with 15.8 - 19.7 mmps for 1 - 128 byte payload size, 23.1 GB/s peak performance at 32 kB payload size. This is followed by C-verbs messages with 11.5 - 18.6 mmps for 1 - 512 bytes, 23.3 GB/s peak performance at 8 kB payload size. jVerbs RDMA writes perform worse on payload sizes up to 1 kB (8.3 - 9.3 mmps) but with less jitter than C-verbs RDMA writes. Saturation on large messages starts at around around 16 kB with 23.7 GB/s with a peak performance of 24.0 GB/s at 128 kB message size. RDMA reads of both verbs interfaces are nearly on par. The incomprehensible poor performance of jVerbs msg verbs, as already seen on the uni-directional benchmark results (§IV-B), is also present here.

The 56 Gbit/s results are depicted in Figure 5 and show an overall similar but as expected lower performance regarding throughput. On small messages, the RDMA write performance of C-verbs is less jittery and jVerb's not significantly lower compared to the 100 Gbit/s results. The RDMA write performance of C-verbs clearly outperforms jVerb's sometimes slight jittery performance on 128 byte to 1 kB messages.

Figure 6 depicts the socket-based results of the bi-directional benchmark. Due to unresolvable errors causing disconnects, especially on message sizes smaller 512 bytes, we cannot provide results for JSOR. This seems to be a known problem [6] but increasing the send and receive queue sizes as described does not resolve this issue. Furthermore, the benchmark does not terminate anymore on message sizes greater than 32 kB. The proposed solution to increase the buffer size does not resolve this problem either [5].

The results available show a very low overall performance for IPoIB and libvma compared to their results on the uni-directional benchmark (§IV-B). IPoIB achieves an aggregated throughput of 0.89 to 0.95 mmps for just up to 16 byte messages with a considerable drop in performance for small messages afterwards. But, throughput increases with increasing message size starting with middle sized messages saturating and peaking at 128 kB message with 3.0 GB/s aggregated throughput. Further notable are high fluctuations for 16 kB to 64 kB. On small messages, libvma can at least provide a constant performance for small messages up to 128 bytes with 0.9 mmps. Throughput increases with increasing message size with saturation starting at approx. 32 kB messages with 1.9 GB/s aggregated throughput. A lower peak performance than IPoIB is reached at 512 kB messages with 2.0 GB/s.

The results on 56 Gbit/s hardware in Figure 7 show high fluctuations on up to 64 byte messages with IPoIB and also on 8 kB to 32 kB messages with libvma.

D. *One-sided Latency*

This section presents the results of the one-sided latency benchmark to determine the latency of a single operation. Section IV-E further discusses full RTT latency for a ping-pong communication pattern. Results are separated by socket-

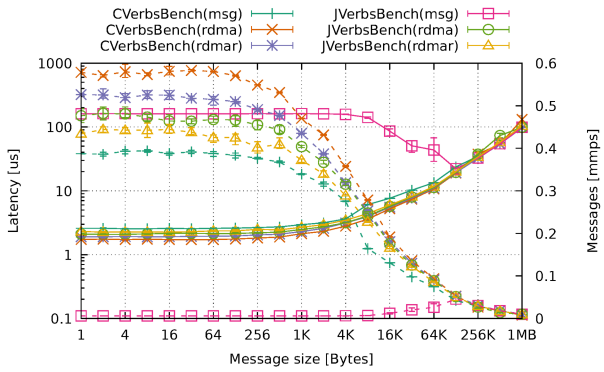

Fig. 8. Average **one-sided latency**, **verbs-based** libraries with different transfer methods, increasing message size, **100 Gbit/s**.

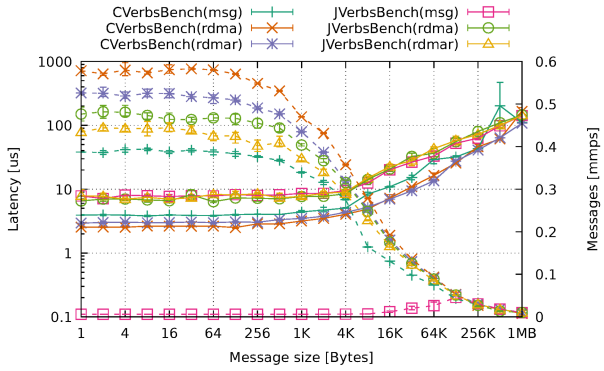

Fig. 9. 99.9th percentile **one-sided latency**, **verbs-based** libraries with different transfer methods, increasing message size, **100 Gbit/s**.

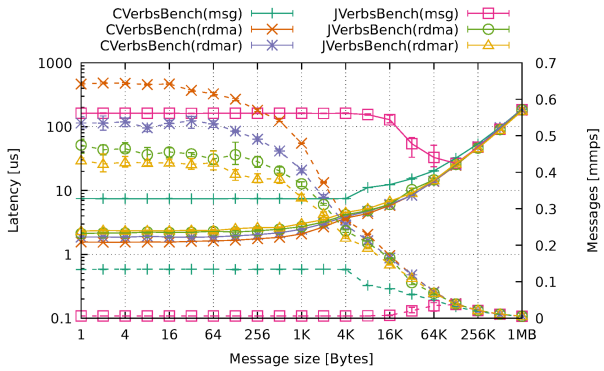

Fig. 10. Average **one-sided latency**, **verbs-based** libraries with different transfer methods, increasing message size, **56 Gbit/s**.

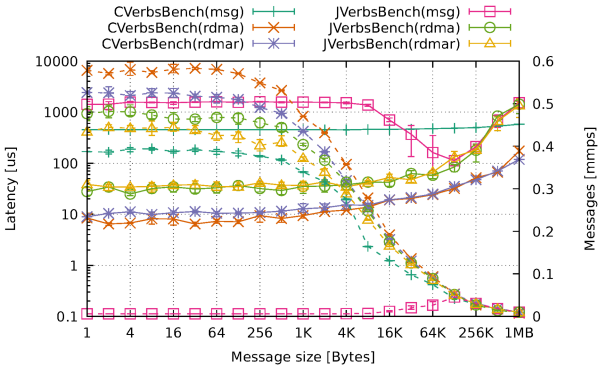

Fig. 11. 99.99th percentile **one-sided latency**, **verbs-based** libraries with different transfer methods, increasing message size, **100 Gbit/s**.

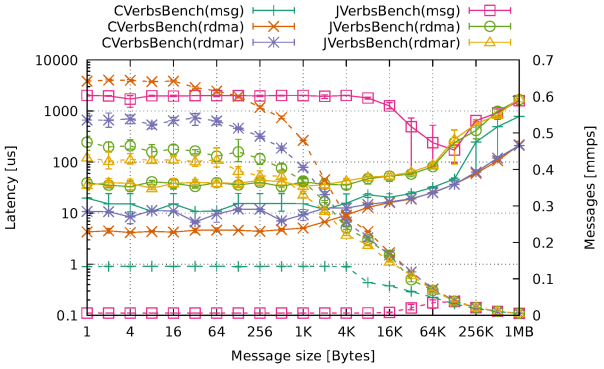

Fig. 12. 99.99th percentile **one-sided latency**, **verbs-based** libraries with different transfer methods, increasing message size, **56 Gbit/s**.

based and verbs-based, and include the average latency as well as the 99.9th percentiles.

Figure 8 shows the average latencies and Figure 9 the 99.9th percentiles of the verbs-based benchmarks. The results of all native verbs-based communication as well as jVerbs RDMA write and read are as expected providing an average close to 1 µs latency. From lowest to highest: C-verbs RDMA write, C-verbs RDMA read, jVerbs RDMA write, jVerbs RDMA read and C-verbs msg. As expected, jVerbs adds some overhead leading to a minor increase in latency.

But, jVerbs msg shows unexpected average latency results. Up to 4 kB message size, which equals the used MTU size, the latency is very high and constant at approx. 160 µs. With further increasing message size, the latency is lowered and approximates the average latencies of the other transfer methods. At 128 kB message size, it goes along with the other results. This is also present on the results on 56 Gbit/s hardware for jVerbs msg (see Figure 10).

To further analyze this issue, we depicted the 99.9th percentiles in Figure 9. The other transfer methods are showing expected behaviour and overall low latency. But, jVerbs msg also shows very low latencies around 8 µs latency for 99.9th of all messages transferred. Thus, just a small amount of messages yields latencies higher than 8 µs. This can be verified

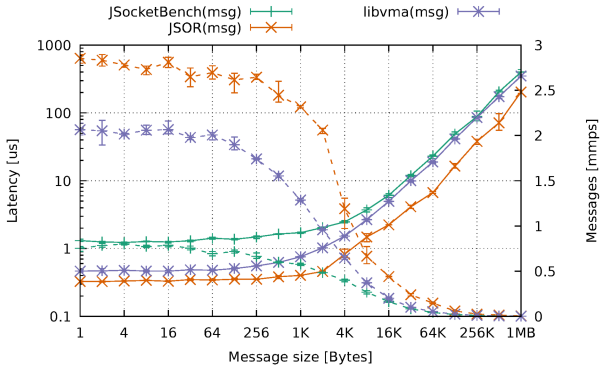

Fig. 13. Average **one-sided latency**, **socket-based** libraries, increasing message size, **100 Gbit/s**.

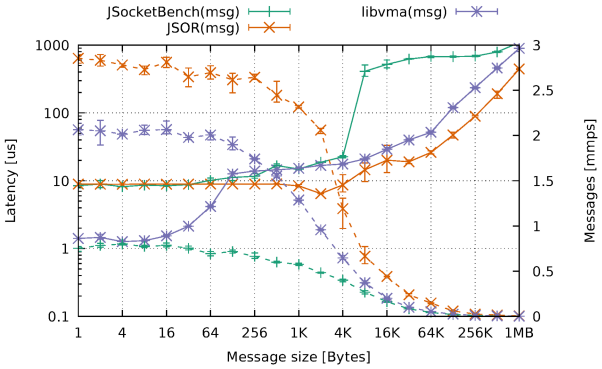

Fig. 14. 99.9th percentile **one-sided latency**, **socket-based** libraries, increasing message size, **100 Gbit/s**.

by analyzing the 99.99th percentiles (i.e. 1,000 worst out of 10 million) depicted in Figure 11. The results show a latency of approx. 1500 µs confirming that there is a very small amount of messages with very high latency causing a rather overall high average latency. This conclusion can also be drawn for jVerbs's results on 56 Gbit/s hardware (see Figure 12).

Two further issues regarding C-verbs msg can be observed. The first is a nearly constant 450 µs on the 99.99th percentile results on 100 Gbit/s hardware (see Figure 11). But, the average and 99.9th percentile results of C-verbs msg are as expected. This means that 1,000 out of 10,000,000 messages show a latency of 450 µs or worse. This is caused by the RC protocol yielding occasional RNR NAKs on high loads when sending data and no corresponding work requests for receiving are queued on the remote at that moment. This is proven by the value of the hardware counter *rnr_nak_retry_err* which shows that about 3,100 send requests are NAK'd during one benchmark run. For the sender, each NAK enforces a small delay to wait for resources to become available again. This behaviour cannot be avoided in such a benchmark scenario.

The second issue with C-verbs msg is a rather high average latency of 7.5 µs for up to 4 kB messages on 56 Gbit/s hardware. However, the RDMA write/read latency results on 56 Gbit/s hardware are similar to the ones on 100 Gbit/s hardware. The 99.9th and 99.99th percentiles (10 to 15 µs) prove that the base latency on that hardware configuration with 56 Gbit/s speed is unexpectedly high.

For the socket-based solutions, the average latencies in Figure 13 show that JSOR performs best with an average per operation latency of 0.3 - 0.4 µs for up to 1 kB messages. With further increasing payload size, latency increases as expected. libvma shows similar results with a slightly higher latency of 0.4 - 0.5 µs for small messages. IPoIB follows with a further increased average latency of 1.3 to 1.5 µs for small messages. These results, especially JSOR's, seem unexpected low at first glance. But, when considering the socket-interface, it does not provide means to return any feedback to the application when data is actually sent to the remote. With verbs, one polls the completion queue and as soon as the work completion is

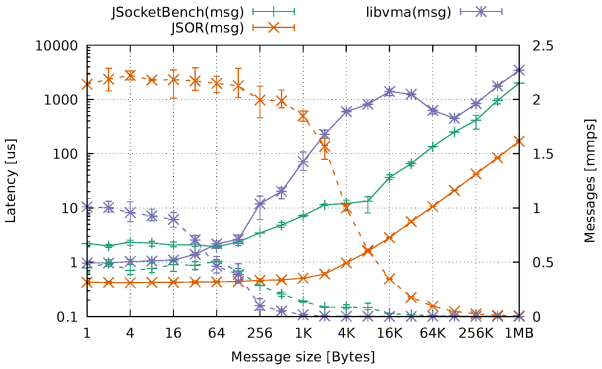

Fig. 15. Average **one-sided latency**, **socket-based** libraries, increasing message size, **56 Gbit/s**.

received, it is guaranteed that the local data is sent and received by the remote. A socket send-call however, does not guarantee that the data is sent once it returns control to the caller. Typically, a buffer is used to allow aggregation of data before putting it on the wire. JSOR, libvma and IPoIB implement message aggregation before actually sending out any data. This is further proven by the ping-pong benchmark which does not allow any aggregation to be applied by the backend (§IV-E). On 56 Gbit/s hardware (see Figure 15), JSOR and IPoIB show similar results with a slightly higher latency but libvma shows significantly worse results for 256 byte to 64 kB message sizes compared to running on 100 Gbit/s hardware.

The 99.9th percentiles in Figure 14 show further interesting aspects not just limited to message aggregation. libvma starts with very low 99.9th latencies for up to 16 byte messages indicating a rather low threshold for aggregation benefitting small messages by keeping the total of worst latencies low. However, with 16 to 128 byte messages, the latency increases considerably. JSOR and IPoIB show similar results for small messages up to 1 kB. JSOR's stays even constant with 9 µs up to 512 byte messages. This indicates a flush threshold based on the number of messages instead of a total buffer size. But, the latency of IPoIB starts to increase already starting with 64 byte message size and even jumps significantly up to over 400

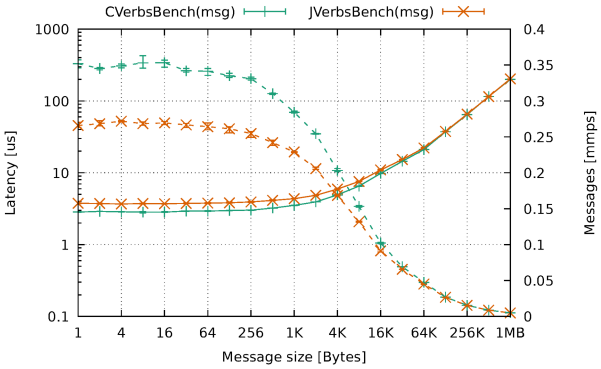

Fig. 16. Average **ping-pong** latency, **verbs-based** libraries with different transfer methods, increasing message size, **100 Gbit/s**.

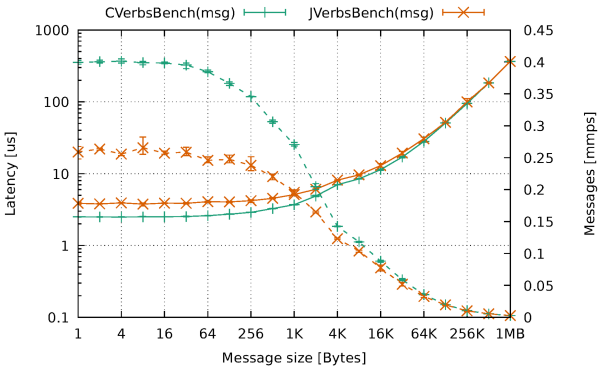

Fig. 17. Average latency **ping-pong**, **verbs-based** libraries with different transfer methods, increasing message size, **56 Gbit/s**.

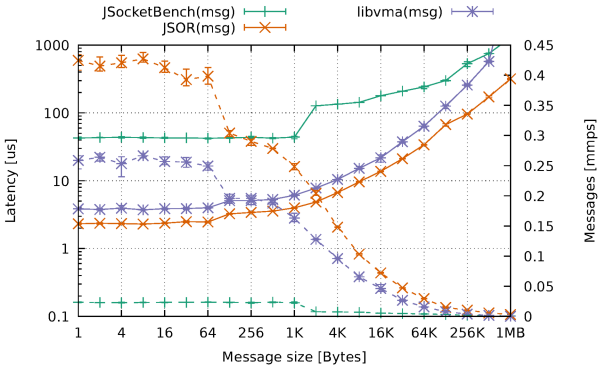

Fig. 18. Average **ping-pong** latency, **socket-based** libraries, increasing message size, **100 Gbit/s**.

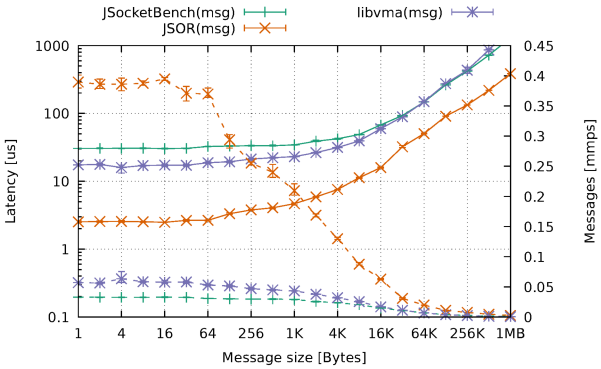

Fig. 19. Average **ping-pong** latency, **socket-based** libraries, increasing message size, **56 Gbit/s**.

µs with 8 kB message size. This might indicate that additional allocations are involved for large(r) messages increasing the overall worst latencies significantly. On 56 Gbit/s hardware, JSOR's and IPoIB's results are similar with a slightly higher latency. But, libvma's 99.9th percentile latency is significantly increasing with 256 byte messages and even reaching 600 ms on multiple sizes greater than 4 kB.

E. Ping-pong Latency

In this section, we present the results of the ping-pong latency benchmark. Due to the nature of the communication pattern, the methods of transfer are limited to messaging operations for verbs-based implementations. Using RDMA operations is also possible but requires additional data structures and control logic which is currently not implemented. The average latencies, i.e. full round-trip-times, are depicted in Figure 16 (Figure 17 for 56 Gbit/s results). C-verbs messaging achieves an average latency of 2.8 to 3.2 µs for up to 512 byte messages. jVerbs shows similar behaviour but with a slightly higher latency of 3.7 to 4.1 µs. Further increasing the message size of both, C-verbs and jVerbs increases the latency as expected. The 99.9th percentiles results on 56 Gbit/s and 100 Gbit/s hardware do not show any abnormalities and their Figures are omitted due to space constraints.

The average latencies of the socket-based methods are depicted in Figure 18 (Figure 19 for 56 Gbit/s results). Both, JSOR and libvma show low average latencies of 2.3 to 3.5 µs and 3.7 to 5.3 µs for message sizes up to 512 bytes. With increasing message sizes, the average latency also increases as expected. A small "latency jump" of around 1 µs is notable on both libraries from 64 byte to 128 byte message size. IPoIB shows a similar behaviour but with a significant higher average latency of 42.7 to 43.2 µs up to 1 kB message size. The same "latency jump" can be seen from 1 kB to 2 kB message size. This increase of latency might be caused due to switching to a different buffer size which might involve additional buffer allocation. The 99.9th percentile results show a similar behaviour without further abnormalities. Their figures are omitted due to space constraints.

V. Conclusions

We presented our JIB-Benchmark to evaluate three socket-based and two verbs-based solutions to leverage InfiniBand in Java applications. We believe that such a benchmark, not available thus far, will help the community and developers interested in using InfiniBand in their Java applications to find a suitable solution for their applications. The benchmark is open source and can be extended to evaluate further li-

braries. We evaluate the available solutions on two hardware configurations with 56 Gbit/s and 100 Gbit/s InfiniBand NICs. As expected, the socket-based solutions provide a transparent solution requiring low effort to get additional performance from InfiniBand hardware for existing socket-based software without requiring any changes. But, this comes at the price that the full potential of the hardware cannot be exploited, especially on bi-directional communication. Compared to the performance of Gigabit Ethernet, latency is at least halved on 56 Gbit/s hardware and can even be as low as 2-5 µs for small messages. Regarding throughput, one can get an at least ten-fold increase and it is even possible to saturate 56 Gbit/s hardware on uni-directional communication.

To leverage the true potential of the hardware, the verbs-based solutions are a must. Overall, jVerbs is performing very well and brings nearly native performance on RDMA operations to the Java space with a few minor performance differences. But, the inexplicable limited performance of jVerbs messaging verbs does not allow any meaningful usage in applications. With C-verbs, the full potential of the hardware can be exploited on all communication methods. Thus, one has to decide whether to stay entirely in Java space but having to rely on the proprietary JV9 JVM or having the freedom to write a custom network subsystem using C-verbs with JNI which is more time consuming and challenging.

Our personal recommendations regarding the evaluation: we consider libvma a good solution to benefit from some of the performance of InfiniBand hardware without having to invest a significant amount of time and work and not depending on a proprietary JVM. But, we think that it is worth spending additional work and time on implementing a custom network subsystem based on C-verbs to leverage the true performance of InfiniBand hardware if required for a target application.

REFERENCES

[1] Apache ignite. https://ignite.apache.org/.

[2] Cassandra. https://cassandra.apache.org/.

[3] Gemfire. https://pivotal.io/pivotal-gemfire.

[4] Hazelcast. https://hazelcast.com.

[5] Ibm. rdma communication appears to hang. https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/diag/problem_determination/rdma_jsor_hang.html.

[6] Ibm. rdma connection reset exceptions. https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/diag/problem_determination/rdma_jsor_connection_reset.html.

[7] Infinispan. http://infinispan.org/.

[8] iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/.

[9] Jib-benchmarks github. https://github.com/hhu-bsinfo/jib-benchmarks/.

[10] libvma github. https://github.com/Mellanox/libvma/.

[11] Mellanox. https://www.mellanox.com/.

[12] Ofed 3.5 release notes. https://downloads.openfabrics.org/OFED/release_notes/OFED_3.5_release_notes.

[13] Openfabrics alliance. https://openfabrics.org/.

[14] Osu micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[15] Ramcloud source code. https://github.com/PlatformLab/RAMCloud.

[16] Rdmamojo. blog by dotan barak. https://www.rdmamojo.com.

[17] Speedus. http://speedus.torusware.com/index.html.

[18] Top500 list.

[19] Infiniband architecture specification volume 1, release 1.3. http://www.infinibandta.org/, 2015.

[20] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.

[21] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Incremental page rank computation on evolving graphs. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 1094–1095, 2005.

[22] R. R. Exposito, S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. Fastmpj: a scalable and efficient java message-passing library. *Cluster Computing*, 17:1031–1050, Sept. 2014.

[23] D. Goldenberg, T. Dar, and G. Shainer. Architecture and implementation of sockets direct protocol in windows. *2006 IEEE International Conference on Cluster Computing*, pages 1–9, 2006.

[24] A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 902–903, 2005.

[25] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47:98 – 115, 2015.

[26] W. Huang, H. Zhang, J. He, J. Han, and L. Zhang. Jdib: Java applications interface to unshackle the communication capabilities of infiniband networks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, pages 596–601, 10 2007.

[27] V. Kashyap. Ip over infiniband (ipoib) architecture. https://www.ietf.org/rfc/rfc4392.txt, April 2006.

[28] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. In *NetDB 2011: 6th Workshop on Networking meetsDatabases*, 2011.

[29] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, 2010.

[30] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[31] X. Liu. Entity centric information retrieval. *SIGIR Forum*, 50:92–92, June 2016.

[32] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: The java unsafe api in the wild. *SIGPLAN Not.*, 50:695–710, Oct. 2015.

[33] S. Mehta and V. Mehta. Hadoop ecosystem: An introduction. In *International Journal of Science and Research (IJSR)*, volume 5, June 2016.

[34] S. Nothaas, K. Beineke, and M. Schoettner. Ibdxnet: Leveraging infiniband in highly concurrent java applications. *CoRR*, abs/1812.01963, 2018.

[35] Oracle. Oracle coherence. https://www.oracle.com/technetwork/middleware/coherence/overview/index.html.

[36] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, November 1999. Previous number = SIDL-WP-1999-0120.

[37] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. sson, D. van der Spoel, B. Hess, and E. Lindahl. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29:845–854, 2013.

[38] P. Stuedi, B. Metzler, and A. Trivedi. jverbs: Ultra-low latency for data center applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 10:1–10:14. ACM, 2013.

[39] G. L. Taboada, J. Touriño, and R. Doallo. Java fast sockets: Enabling high-speed java communications on high performance clusters. *Comput. Commun.*, 31:4049–4059, Nov. 2008.

[40] S. Thirugnanapandi, S. Kodali, N. Richards, T. Ellison, X. Meng, and I. Poddar. Transparent network acceleration for java-based workloads in the cloud. https://www.ibm.com/developerworks/library/j-transparentaccel/, January 2014.

[41] X. Wu, X. Zhu, G. Q. Wu, and W. Ding. Data mining with big data. *IEEE Transactions on Knowledge and Data Engineering*, 26:97–107, Jan. 2014.

[42] P. Zhao, Y. Li, H. Xie, Z. Wu, Y. Xu, and J. C. Lui. Measuring and maximizing influence via random walk in social activity networks. pages 323–338, Mar. 2017.

5.4.2 DXNet: A Transport Agnostic Network Subsystem for Highly Concurrent Java Applications

This section presents the contributions of the proposed publication of DXNet [12]. DXNet is a network stack that implements event-based messaging with asynchronous and synchronous messaging primitives and is optimized for highly concurrent Java applications. DXNet focuses on sending and receiving of small messages with a fast and efficient serialization of Java objects. DXNet's transport interface allows implementing transports for different network interconnects. Currently, DXNet supports socket-based Ethernet [17] as well as verbs-based InfiniBand (see Section 5.4.3). This publication presents the results of a close collaboration of Dr. Kevin Beineke and Stefan Nothaas who worked on DXNet.

DXNet is based on DXRAM's initial network subsystem. The architecture was revised by Dr. Kevin Beineke and Stefan Nothaas to address the requirements of the target application domain. Dr. Kevin Beineke replaced inefficient data structures, optimized thread handling, and buffer processing to improve overall throughput and latency. Stefan Nothaas further refactored DXNet to a hardware agnostic network stack, introduced a modular transport layer, extended the built-in benchmark for multi-server communication and optimized DXNet's flow control mechanism for the InfiniBand transport. Optimizations of the latter were adapted on the Ethernet transport by Dr. Kevin Beineke. With the new transport interface, DXNet was optimized for the InfiniBand transport implementation (see Section 5.4.3).

The Ethernet transport was refactored and maintained by Dr. Kevin Beineke and is not further discussed in this thesis. The InfiniBand transport was developed and maintained by Stefan Nothaas. To ensure low-latency and high throughput, especially for small messages, on high-speed networks like 10 GBit/s Ethernet and 56 Gbit/s InfiniBand, Dr. Kevin Beineke and Stefan Nothaas worked in close collaboration to continuously improve DXNet making it one of the fastest messaging systems for concurrent Java applications.

The following paragraphs describe the specific apportionment of work for this publication to the best of the knowledge of both authors and is an excerpt from Dr. Kevin Beineke's thesis [11].

"Dr. Kevin Beineke designed and implemented the concurrent de-/serialization including workflow optimizations and thread communication, the data structure pooling, the loopback, and Java.nio transports and the basic DXNet benchmark. Furthermore, most of the ORB and CUB were designed by Dr. Kevin Beineke, with contributions by Stefan Nothaas. Other lock-free data structures were inspired by the ORB and implemented by both authors as well as the parking strategy which emerged in an incremental process."

"Stefan Nothaas initiated many optimizations by designing Ibdxnet and discovering bottlenecks in DXNet's core shared by all transports which were hardly detectable with slower networks. Furthermore, Stefan Nothaas introduced the idea of using lock-free data structures to improve the performance of DXNet. Despite contributing performance optimizations and debugging, Stefan Nothaas also implemented interfaces for the serialization and accessing direct ByteBuffers with Java's Unsafe class. Additionally, Stefan Nothaas invested much time in structuring the code and a statistics module to aid in investigating DXNet's performance."

”Prof. Dr. Michael Schöttner took part in many discussions about the design and evaluation of DXNet.“

”Dr. Kevin Beineke structured and wrote most of the paper, including all figures but Figures 2 and 3 whose initial design was contributed by Prof. Dr. Michael Schöttner. Prof. Dr. Michael Schöttner also helped in improving comprehensibility and reviewed the paper several times. Stefan Nothaas contributed in writing the initial versions of section I and II and designing a figure of DXNet's architecture which was used for Figure 1 and 5. Stefan Nothaas reviewed the paper several times as well and helped improve it.”

Efficient Messaging for Java Applications running in Data Centers

Kevin Beineke, Stefan Nothaas and Michael Schöttner

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
E-Mail: Kevin.Beineke@uni-duesseldorf.de

***Abstract*—Big data and large-scale Java applications often aggregate the resources of many servers. Low-latency and high-throughput network communication is important, if the applications have to process many concurrent interactive queries. We designed DXNet to address these challenges providing fast object de-/serialization, automatic connection management and zero-copy messaging. The latter includes sending of asynchronous messages as well as synchronous requests/responses and an event-driven message receiving approach. DXNet is optimized for small messages (< 64 bytes) in order to support highly interactive web applications, e.g., graph-based information retrieval, but works well with larger messages (e.g., 8 MB) as well. DXNet is available as standalone component on Github and its modular design is open for different transports currently supporting Ethernet and InfiniBand. The evaluation with micro benchmarks and YCSB using Ethernet and InfiniBand shows request-response latencies sub 10 µs (round-trip) including object de-/serialization, as well as a maximum throughput of more than 9 GByte/s.**

***Keywords*—Message passing; Ethernet networks; InfiniBand; Java; Data centers; Cloud computing;**

I. Introduction

Today, many interactive applications are built upon very large graphs, e.g., social networks [1], comparing molecular structures in bioinformatics [2] or mobile network state management systems [3]. These graphs consist of billions of small data objects which are typically held in memory to provide low latency access. But, as data volumes grow fast it becomes necessary to aggregate many servers or move to expensive super computers. Usually, big data applications are executed in cloud data centers or on high performance clusters which provide fast networking with 10 GBit/s and beyond. Distributed and parallel processing of in memory data based on very fast networks requires the software stack to be designed carefully, especially if latency is important.

Many big data applications are written in Java and benefit from platform independence and a rich selection of libraries supporting the programmer in designing distributed and parallel applications [1], [4]–[8]. This includes many possibilities to exchange data between Java servers, ranging from high-level Remote Method Invocation (RMI) [9] to low-level byte stream processing using Java sockets [10] or the Message Passing Interface (MPI) for HPC applications [11]. DXNet does not aim at replacing any of those solutions but to rather complement the spectrum.

DXNet is a network library for Java-based applications which has originally been designed for DXRAM a distributed in-memory key-value store and DXGraph a graph processing framework built on top of DXRAM. We provide DXNet as a standalone library through GitHub [12] as we think it could be useful for many other Java-based big data applications.

The contributions of this paper are:

- the DXNet architecture (highly concurrent and transport agnostic);
- zero-copy, parallel de-/serialization of Java objects;
- lock-free, event-driven message handling;
- evaluations with 5 GBit/s Ethernet (Microsoft Azure) and 56 GBit/s Infiniband networks.

The evaluation shows that DXNet efficiently handles high loads with dozens of application threads concurrently sending and receiving messages. Synchronous request/response patterns can be processed in sub 10 µs RTT (Round-Trip Time) with Infiniband transport (including object de-/serialization). And, high throughput is achieved even with smaller payloads, e.g., bandwidth saturation with 1-2 KB payload on InfiniBand and 256 byte payload on Ethernet.

The structure of the paper is as follows: after discussing related work, we present an overview of DXNet in Section III. In Section IV, we describe the lock-free Outgoing Ring Buffer followed by the concurrent serialization in Section V. The next section explains the event-driven processing of incoming data. Sections VII and VIII present thread parking strategies and transport implementation aspects. Evaluation results are discussed in Section IX, followed by the conclusions.

II. Related Work

DXNet combines high-level thread and connection management and a concurrent object de-/serialization with lock-free, event-driven message handling and zero-copy data transfer over Ethernet and InfiniBand (extensible). To the best of our knowledge, no other Java-based network library provides these communication semantics. Because of space constraints, we compare DXNet with the most relevant related work, only.

Distributed shared memory (DSM) is re-gaining attraction due to networks supporting **RDMA** but is not an option for most existing Java applications as DSM requires a different application architecture and an integration in the heap management of the Java Virtual Machine (JVM) [13].

Java's RMI [9] provides a high level mechanism to transparently invoke methods of objects on a remote machine, similar to Remote Procedure Calls (RPC). Parameters are automatically de-/serialized and references result in a serialization of the object itself and all reachable objects (transitive closure) which can be costly [14]. Missing classes can be loaded from

remote servers during RMI calls which is very flexible but introduces even more complexity and overhead. The built-in serialization is known to be slow and not very space efficient [14], [15]. Furthermore, method calls are always blocking.

Manta [16] improves runtime costs of RMI by using a native static compiler. **KaRMI** [17], a drop-in replacement for Java RMI, is implemented in Java without any native code supporting standard Ethernet. KaRMI also replaces Java's built-in serialization reducing overhead and improving overall performance. DXNet does not provide transparent remote method calls but an efficient parallel serialization which avoids copying memory. DXNet is primarily designed for parallel applications and high concurrency, RMI for Web applications and services.

MPI is the state-of-the-art message passing standard for parallel high performance computing and provides very efficient message passing for primitive, derived, vector and indexed data types [18]. As MPI's official support is limited to C, C++ and Fortran, Java object serialization is not provided. Nevertheless, MPI is available for Java applications through implementations of the MPI standard in Java [19] or wrappers of a native library [20].

MPI-2 introduced multi-threading for MPI processes [18] enabling well-known advantages of threads. Prior to MPI-2, intra-node parallelization demanded the execution of multiple MPI processes (and the use of more expensive IPC). To enable multi-threading, the process has to call `MPI_init_thread` (instead of `MPI_init`) and to define the level of thread support ranging from single-threaded execution over funneled and serialized multi-threading to complete multi-threaded execution (every thread may call MPI methods at any time). A lot of effort has been put into the last mode to provide a high concurrent performance [21], [22]. Still, the performance is limited compared to a message passing service designed for multi-threading [21].

One of DXNet's main application domains are ongoing applications with dynamic node addition and removal (not limited to), e.g., distributed key-value stores or graph storages. The MPI standard defines the required functionality for adding and removing processes (over Berkeley Sockets with `MPI_Comm_join` or by calling `MPI_Open_port` and `MPI_Comm_accept` on the server and `MPI_Comm_connect` on the client). Unfortunately, most recent MPI implementations are still not supporting these features entirely [23], [24]. Furthermore, job shutdown and crash handling is also limited [24]. MPI is particularly suitable for spawning jobs with finite runtime in a static environment. DXNet, on the other hand, was designed for up- and downscaling and handling node failures. In [25], DXNet was used in the in-memory key-value store DXRAM to examine crash behavior and scalability.

High level mechanisms for typical **socket-like interfaces** supporting Gigabit Ethernet (and higher) are provided by Java.nio [26], [27], Java Fast Sockets (JFS) [28] or High Performance Java Sockets [29]. DXNet uses Java.nio to implement a transport for commonly used Ethernet networks.

III. Overview

DXNet relieves programmers from connection management, provides transferring Java objects (beyond plain Java.nio stream sockets) and allows the integration of different underlying network transports, currently supporting reliable verbs over InfiniBand and TCP/IP over Ethernet. In this section, we give a brief overview of the interfaces and functionality of DXNet (see Fig. 1). Further details can be found in the GitHub repository [12].

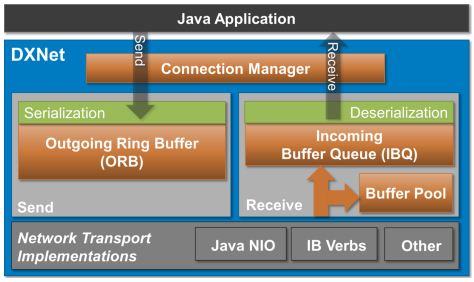

Figure 1. Simplified DXNet Architecture

A. Basic Functionality

Automatic connection management. DXNet abstracts physical network addresses, e.g., IP/Port for Ethernet or GUID for InfiniBand, by using nodeIDs. The aforementioned node address mappings are registered in the library and are mutable for server up- and downscaling. A new connection is opened automatically when a message needs to be sent to another server which is not connected thus far. In case of errors, the library will throw exceptions to be handled by the application. Connections are closed based on a recently used strategy, if the configurable connection limit is exceeded, or in case of network errors which may be reported by the transport layer or detected using timeouts, e.g., absent responses.

Sending messages. DXNet sends messages asynchronously to one or multiple receivers but also provides blocking requests (to one receiver) which return when the corresponding response is received (association of response and requests is transparently managed by DXNet). **Messages are Java objects and serialized** by using DXNet's fast and concurrent serialization (providing default implementations for most commonly used objects, see Section V). The serialization writes directly into the Outgoing Ring Buffer (ORB) which aggregates messages for high throughput (see Section IV) and is allocated outside of the Java heap. Sending data is performed by a decoupled transport thread based on event signaling. DXNet also includes a flow control mechanism, which is not further described here.

Receiving messages. When incoming data is detected by the network transport, it requests a pooled native memory buffer (avoiding to burden the Java garbage collector) and copies the data into the buffer (see Section VI and Fig. 1). The buffer containing the received data is then pushed to the Incoming Buffer Queue (IBQ), a ring buffer storing references on buffers which are ready to be deserialized (see Section VI). The buffer pool and the IBQ are shared among all connections. The buffers of the IBQ are pulled and processed asynchronously by dedicated threads. Message processing includes parsing message headers, creating the message objects and deserializing the payload data. Finally, the **received message is passed back to the application (as a Java object)** using a pre-registered callback method.

A brief overview of DXNet's API is shown in Table I.

TABLE I. DXNET'S APPLICATION INTERFACE

`new DXNet(config,nodeMap)`	initialize/configure (max. connections, server address mappings etc.)

`MyMessage extends Message/Request/Response`
`  exportObject(exporter)`
`  importObject(importer)`
`  sizeOfObject()`
	define message (serializable Java object) by implementing three methods
	serialize message with predefined methods from exporter
	deserialize message with predefined methods from importer
	return payload length

`sendMessage(message)`	send message asynchronously (receivers defined in message instance)
`sendSync(request,timeout)`	send request/response synchronously
`MyReceiver implements MessageReceiver` `  onIncomingMessage(message)`	receive messages/requests as Java objects pre-registered callback handler function

B. High Throughput and Low Latency

A key objective of DXNet is to provide high throughput and low latency messaging even for small messages found in many graph applications, for instance. We achieve this with a thread-based and event-driven architecture using lock-free synchronization, zero-copy, and zero-allocation.

Multithreading. All processing steps like serialization, deserialization, message transfer and processing are handled by multiple threads which are decoupled through events allowing high parallelism.

Lock-free event signaling. Dispatching processing events between threads is implemented using lock-free synchronization allowing low-latency signaling. CPU load is managed without impairing latency by parking currently idling threads.

Fast serialization. DXNet implements fast serialization of complex data structures and writes data directly into an ORB. The ORB can be accessed by many threads in parallel and ORBs are not shared between different connections increasing concurrency even more. The processing of incoming messages is also highly scalable because of the event-driven architecture.

Zero copy. DXNet does not copy data for messaging (except de-/serialization). For TCP/IP, we rely on Java's DirectByteBuffers and for InfiniBand on verbs pinning the buffers used by DXNet.

Zero allocation. DXNet uses object pooling wherever possible avoiding time-consuming instance creation and, even more important, not burdening the Java garbage collector which may block an application in case of low memory for multiple seconds.

C. Network Transport Interface

DXNet supports different underlying reliable network transports. The integration of a new transport protocol requires implementing just five methods:

- signal data availability on connection (callback);
- pull data from ORB and send it;
- push received data to IBQ;
- setup a connection;
- close a connection.

IV. LOCK-FREE OUTGOING RING BUFFER

The Outgoing Ring Buffer (ORB) is a key component for outgoing messages and essential for providing high throughput and low latency. The latter is achieved by a highly concurrent approach based on lock-free synchronization.

Each connection has one dedicated ORB allowing concurrent processing of different connections. The ORB itself allows many application threads serializing their outgoing objects concurrently and directly into the ORB. The ORBs are allocated outside of the Java heap in native memory allowing zero-copy sending by the network transport. Directly serializing Java objects into the ORB is more efficient than serializing each object in a separate buffer and combining them later by copying these buffers. The ORB preserves message ordering as given by the application threads and aggregates smaller messages in order to achieve high throughput. We decided to use lock-free synchronization for concurrency control which is more complex but more efficient with respect to latency compared to locks.

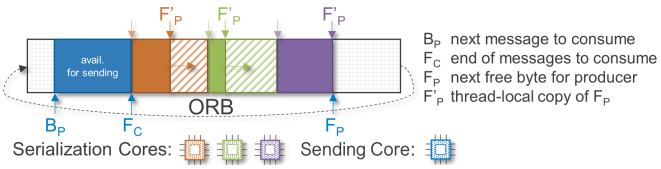

Figure 2. ORB for parallel serialization and aggregating outgoing messages.

A. Basic Lock-Free Approach

The ORB has a configurable but fixed size and is accessed concurrently by several producers (application threads) and one consumer (dedicated transport thread for sending messages). The configurable buffer size limits the maximum number of messages/bytes to be aggregated. For our experiments (see Section IX), we used 1 MB and 4 MB ORBs.

Fig. 2 shows the ORB with three application threads producing data (serialization cores). All pointers move forward from left to right with a wrap around at the end. The white area between F_P and B_P is free memory.

Messages available for sending (fully serialized) are found by the consumer (sending core) between B_P and F_C. The consumer sends aggregated messages and moves B_P forward accordingly but not beyond F_C. All messages between F_C and F_P are not yet ready for sending as parallel serialization is still in progress.

F_P is moved forward concurrently (if the buffer has enough space left) by the producers using a Compare-and-Set (CAS) operation, available in Java through `Unsafe` (see Section IV-C). Therewith, each producer can concurrently and safely store the position of F_P in a local variable F'_P and adjust F_P by its message size. All F'_P pointers (thread-local variables) are used by the associated producer for writing its serialization data concurrently at the correct position in the ORB. The light-colored arrows in Fig. 2 show the starting point of each serialization core (producer) whereas the solid-colored

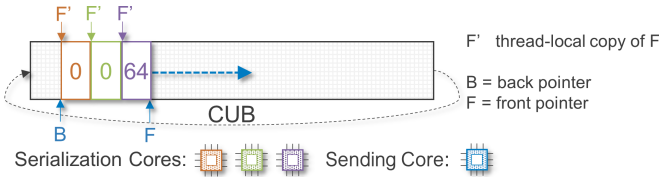

Figure 3. Catch-Up Buffer (CUB). Allowing faster producers returning early and not wasting CPU cycles for waiting.

ones show the current position. In the example, the purple producer finished its serialization first and the green and orange producers are still working.

F_C is moved forward by producers when messages are fully serialized. In Fig. 2, the purple producer finishes before the orange and green ones but cannot set F_C to F_P because the two preceding messages (from the other producers) have not been completely serialized yet. Each producer can easily detect unfinished preceding messages by comparing its starting point (light-colored arrow) with F_C. Obviously, the purple starting point is not equal to F_C. A naive solution lets fast producers wait for slower ones. As we do not want to impact latency we cannot use locks/conditions here. An alternative solution is to busy-poll until all preceding messages have been serialized. Finally, F_C can be set forward and the thread can return.

B. Optimized Lock-Free Solution

The basic solution already avoids the overhead of locks, but with increasing number of parallel serializations the probability of threads having to wait for slower ones increases. The busy-polling can easily overload the CPU. Reducing the polling frequency of producers by sleeping (> 1 ms) or parking (> 10 µs) increases latency too much. Instead, we propose a solution which avoids having fast producers waiting for slower ones by leaving a notice and returning early to the application. This notice includes the message size so that slower producers can move F_P forward for the faster ones. But, message ordering must be preserved.

Our solution is based on another configurable fixed-size ring buffer called Catch-Up Buffer (CUB). As mentioned before, we allocate one ORB for each connection which is now complemented by one associated CUB (e.g., with 1000 entries) for every ORB. The CUB is implemented using an integer array, each entry for one potential left-back notice from faster producers. An entry will be 0 if there is no notice or > 0 representing the message size if a producer finished faster than its predecessors. In the latter case, a slower producer will move forward F_P by the left-back message size.

Fig. 3 shows a CUB corresponding to the ORB shown in Fig. 2. The front pointer F is moved concurrently using a CAS operation (similar to F_P in the ORB). The colored F' are the thread-local copies needed by the producers to leave back a potential notice at the correct position in the CUB. The 64 is a notice from the purple producer (its message size, filled purple box in Fig. 2.) who finished fastest and returned already to the application. The green and orange producers are still working (0 = no notice). If the green producer would now finish before the orange one it would also fill in its message size and return immediately.

If the orange producer finishes next, it moves forward F_C in the ORB as well as B in the CUB (leaving no notice). The green one will do the same, but twice as it will detect the notice (64) after committing its serialization and thus move forward F_C in the ORB by 64 bytes and also B (now pointing to F in the CUB, indicating we are done).

It is important that the order of entries in the ORB and the CUB is consistent, meaning, we need to move forward F and F_P, as well as B and F_C synchronously. We do this, by storing each of those two indexes in one 64-bit long variable in Java and, as the CAS operation is working atomically on 64-bit longs, we can avoid locks.

Two more challenges remain, namely large messages which cannot be serialized at once and a potential ORB overflow during the serialization (both discussed in Section V).

C. Native Memory

The ORB is allocated in native memory (off Java heap) allowing the underlying network transports to send messages without copying them. The class `Unsafe` provides basic methods for memory allocation, memory copy and reading/writing primitives from/into native memory. Furthermore, `Unsafe` is very fast because of extensive optimizations and is widely used in third-party libraries [30].

We favor Unsafe over DirectByteBuffers [27] for two reasons. First, access is faster (e.g., missing boundary checks we already handle on higher level). Second, Unsafe is more versatile because it allows accessing memory which was allocated in C/C++ code (e.g., used for InfiniBand).

V. Serialization

DXNet is designed to send and receive Java objects which need to be de-/serialized from/into a byte stream of messages. The built-in serialization of Java (interface `Serializable`) as well as file-based solutions are too slow and have a large memory footprint [31] (because of automatic un-/marshaling and the use of separators). Other binary serializer like Kyro [32], for instance, either do not support writing directly into native memory or interruptible processing which is needed by DXNet (see Section V-A and V-B). We propose a new serializer addressing all these limitations while still being intuitive to use. The programmer has to implement two interfaces `Importable` and `Exportable`. The former requires implementing the method `importObject()`, the latter `exportObject()` and both `sizeOfObject()`.

A. Export

Exporter. The serialization (or export) of Java objects requires an `exporter` which is passed to `exportObject()`. The exporter class provides default method implementations for the serialization of all primitives, *compact numbers* and Strings and can be extended for supporting custom types (all types can also be arranged in arrays). Compact numbers are coded integers using a variable number of bytes as needed to reduce space overhead.

The exporter writes directly into the ORB by using Unsafe (see Section IV). It stores the start position within the ORB, the size of the ORB and the current position within the message.

Exporting an object involves two steps: exporting the message header (see Fig. 4) which has a fixed size and exporting the variable-sized payload by calling `exportObject()`.

DXNet uses its `default exporter` for serialization which is optimized for performance. It is complemented by

Figure 4. Message header. Cat.: message, request or response; X: exclusive or not (ordering).

two other exporters (described below) for handling messages which do not fit in the ORB without copying buffers.

Buffer overflow. If the end of the ORB will be reached during the serialization of an object, DXNet switches to the `overflow exporter`. The overflow exporter performs a boundary check for each data item of an object and writes bytes with a wrap-around to the beginning of the ORB, if necessary. The resulting message is sent as two pieces over the network stream avoiding copying data.

Large messages. Serialized objects resulting in messages larger than the ORB must be written iteratively. First, the entire unused section of the ORB (see Fig. 2) is reserved and filled with the first part of the message. If the back pointer is reached, the export is interrupted and its current state is stored in an `unfinished operation` instance to allow resuming serialization as soon as there is free space in the ORB again.

Unfinished operation. The instance stores the interrupt position within the message and the rest of the current operation. Depending on the operation, the rest is either a part of a primitive which can be stored in a `long` within the unfinished operation or an object with partly uninitialized fields whose reference can be stored.

Resume serialization after an interrupt. To continue the serialization, the method `exportObject()` is called again (threads return after being interrupted during serialization) and all previously successfully executed export operations are automatically skipped until the position stored in the unfinished operation is reached. The rest of the object is serialized from there (might be interrupted, again). For exporting large messages, the `large message exporter` is used, which extends the overflow exporter.

B. Import

All incoming messages are written into native memory buffers taken from the incoming buffer pool and are pushed to the IBQ (see Section VI). Each buffer contains received bytes (one or several messages) from the connection stream. The underlying network independently splits and aggregates packets resulting in a buffer beginning and ending at any byte within a message. DXNet is able to serialize split messages without copying buffers.

The import works analogously to the export. Messages are deserialized directly from native memory by using Unsafe (message header and payload). The fast `default importer` is complemented by three other importers (described below) for handling split messages. This requires to handle three situations: buffer overflow (tail of message/header missing), buffer underflow (head of a message/header is missing) and both combined.

Buffer overflow. When the buffer's end will be reached before the message is complete, we switch to the `overflow importer`. It does boundary checks and uses the unfinished operation (see Section V-A) when necessary. Furthermore, the serialization is aborted with an `IndexOutOfBoundsException` handled by DXNet

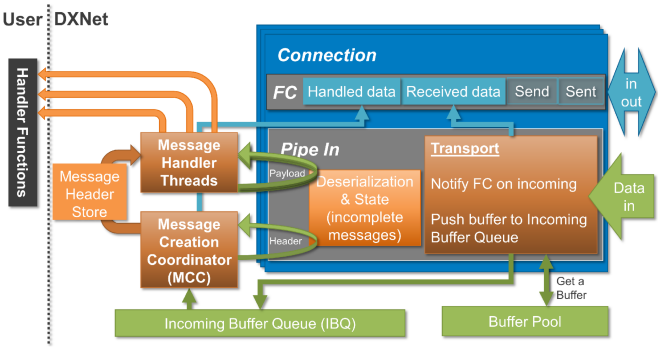

Figure 5. Receiving and processing messages. Green: Native memory access.

avoiding returning invalid values for succeeding operations.

Buffer underflow. This situation occurs after a buffer overflow (on the same stream). It is known apriori and handled by the `underflow importer`, which uses the unfinished operation instance (passed from the overflow importer) containing all information necessary to continue deserialization.

Buffer under- and overflow. When a message's head and tail is missing (likely for large messages), the message is handled by the `underoverflow importer`.

C. Resumable Import and Export Methods

Messages may be split caused by DXNet's buffering or the underlying network. In order to avoid copying buffers, we require both import and export methods to be interruptible and idempotent as they may be called multiple times for one object (to avoid blocking threads, see Sections V-A and V-B). DXNet's importer and exporter methods are sufficient for most object types, but custom object structures must be aware of this and avoid functions causing side effects (e.g., I/O access).

VI. Event-Driven Processing of Incoming Data

Fig. 5 gives an overview of the parallel event-driven processing of incoming data. Like for the ORB, we use multi-threading, lock-free synchronization, zero-copy and zero-allocation to provide high throughput and low latency.

Receiving process. The network transport pulls a buffer from the incoming buffer pool when new data can be received and fills it accordingly. The buffer is then pushed to the IBQ and processed by the **Message Creation Coordinator** thread (MCC) by deserializing the message headers. The message headers are pushed to the **message header store** afterwards. Multiple message handler threads concurrently create the message objects, deserialize the messages' payloads and pass the received Java objects to the application using its registered callback methods. When all data of a buffer has been processed, it is released and pushed back into the incoming buffer pool.

Incoming buffer pool. The buffer pool provides buffers, allocated in native memory, in different configurable sizes (e.g., 8×256 KB, 256×128 KB and 4096×16 KB). The transport pulls buffers using a worst-fit strategy as the amount of bytes ready to be received on the stream is unknown. It can also scale-up dynamically, if necessary.

The buffer pool management consists of three lock-free ring buffers optimized for access of one consumer and N producers (similar to the ORB but without the CUB, see Section IV).

A. Parallel Message Deserialization

Filled buffers are pushed by the transport thread into the IBQ. The IBQ is a basic ring buffer for one consumer and one producer and is synchronized using memory fences. The IBQ may be full and require the transport thread to park for a short moment and retry (see Section VII).

High throughput requires a parallel deserialization. As the received messages of the incoming stream can be split over several incoming buffers (see Section V-B), the buffer processing must be in-order and we need a two-staged approach to enable concurrency. The MCC thread pulls the buffer entries from the IBQ, deserializes all containing message headers (using relevant state information stored in the corresponding connection object) and pushes them into the message header store. Message payload deserialization based on the message headers can then be done in parallel by the message handler threads. This approach is efficient as the time-consuming payload deserialization and message object creation is parallelized.

The deserialization of split messages' payload (last message in buffer, which is not complete) must be in-order as well because all preceding parts of a message must be available to continue the deserialization of a split message. We address this situation by the MCC detecting and deserializing not only the header but the payload fraction within the current buffer, as well, for the split message. The rest of the message in the next buffer can be read by a message handler, again.

Split message headers are not a problem as deserialization of message headers is always done by the MCC which can store incomplete message headers within the connection object and continue with the next buffer.

Message header store. As mentioned before, the MCC pushes complete message headers to the message header store. The latter is implemented as a lock-free ring buffer for N consumers and one producer. Synchronization overhead is reduced by the MCC buffering the small message headers and pushing them in batches into the message header store. The batch size is limited but configurable, e.g., 25 headers.

Message header pool. Message headers are pooled, as well, in another single consumer, multiple producers lock-free ring buffer. Furthermore, message headers are pushed and pulled in batches. To reduce the probability of multiple message handler threads returning message headers at the same time, the batch sizes differ for every message handler.

Returning of buffers. A pooled buffer must not be returned before all its messages haven been deserialized. Because of the concurrent deserialization and split messages, we use the MCC incrementing an atomic counter for every message header pushed to the message header store (more precisely, the counter is increased once for every batch of message headers). Accordingly, the message handlers decrement the counter for every deserialized message. When all messages have been deserialized, the buffer can be safely returned to the pool.

We could run out of buffers during high throughput, if the MCC deserializes headers faster than the message handler threads can handle. Although we can scale up the number of incoming buffers, we prefer to throttle the MCC when a predefined number of used buffers is exceeded to reduce the memory consumption. Another benefit of limiting the amount of incoming buffers is that all buffer states like the message counters, the buffers' addresses or the unfinished operations which are filled for incomplete messages can be allocated once and reused for every incoming buffer to be processed.

Message Ordering. DXNet allows applications to mark messages and thus ensure message ordering on a stream/connection. All marked messages are guaranteed to be processed by the same message handler. All other steps preserve message ordering by default. For achieving maximum throughput, marking all messages is not advisable.

VII. Thread Parking Strategies

Lock-free programming allows low-latency synchronization but can easily overload a CPU by uncontrolled polling using CAS operations. DXNet implements a multi-level flow control with explicit message flow regulation and implicit throttling if memory pools drain and queues fill-up. We address three thread situations: blocked (the thread waits for another thread/server finishing its work because a pool is empty or queue full), colliding (failing CAS operation because another thread entered a critical section faster) and idling (the thread has nothing to do and waits for another thread/server committing new work).

Blocked thread. When blocked, the thread can park to reduce the CPU load because it is too fast compared to other threads/servers. However, the thread should not park for a long period to avoid restraining other threads/servers. Experiments showed that a sane park period is between 10 and 100 µs. Java allows minimum parking times of around 10 to 30 µs for a thread with `LockSupport.parkNanos()` for Linux servers with x86 CPUs.

Colliding thread. When colliding, the thread will repeat the CAS operation with updated values until successful because the thread is about to commit something and this should be done as fast as possible. However, reducing the collision probability (e.g., the ORB optimization described in IV-B) reliefs the CPU significantly.

Idling thread. This situation occurs, if a thread has nothing to do at the moment, e.g., a transport thread polls an empty ORB, the MCC polls an empty IBQ or a message handler polls an empty message header store. However, new work events can arrive within nanoseconds. Latency is minimized when threads do not park or yield, but only as long as the CPU is not overloaded. In case of CPU overload situations, parking threads can reduce latency.

We address this with an overprovisioning detection combined with an adaptive parking approach (10 to 30 µs), if the number of active threads (application threads and network threads) reaches a threshold, e.g., four times the number of cores, see also Section IX-A for the evaluation.

Idling for longer periods, e.g., applications not exchanging messages for a longer period of time, must be addressed, too. DXNet detects this, e.g., a network thread idling for one second (configurable time), and starts parking threads, if idling, reducing CPU load to a minimum.

VIII. Transport Implementations

DXNet has an open architecture supporting different network transport technologies. Currently, we have transport implementations for TCP/IP over Ethernet (using Java.nio), reliable verbs over Infiniband (based on JNI), and Loopback (for evaluation). Because of space constraints, we will only sketch some important aspects of these transports.

The `Ethernet` transport (EthDXNet) implementation is based on Java.nio and maps DirectByteBuffers to the ORB allowing to send data without copying it in user-space. Furthermore, two channels are opened for every connection to avoid channel duplication and for providing a side-band flow control channel for each connection. Channel duplication may occur when two servers create connections to each other simultaneously and must be avoided. The second channel allows exchanging flow control messages necessary to maximize throughput on a connection by using the back-channel.

The `InfiniBand` transport accesses the IBDXNet library (C++) using JNI. IBDXNet utilizes ibverbs to implement direct communication using the InfiniBand HCA. IBDXNet uses one dedicated send and one dedicated receive thread, both processing outgoing/incoming data in native memory. Context switching from C++ to Java was designed carefully and is highly optimized to avoid latency.

The `Loopback` transport is used for the experiments in this paper allowing to study the performance of DXNet without any bottlenecks from a real network. Data is not sent over a network device nor the operating system's loopback device (latency would be considerably high) but is directly copied from the ORB to a pooled incoming buffer. Furthermore, the Loopback transport simulates a server sending and receiving messages at highest possible throughput allowing to evaluate DXNet's performance.

IX. Evaluation

We evaluate the proposed concepts using Loopback and three different networks: 1 GBit/s Ethernet, 5 GBit/s Ethernet and 56 GBit/s InfiniBand. The Loopback is used to evaluate DXNet's concepts without any limitations of an underlying network.

Loopback and 5 GBit/s Ethernet tests were run in Microsoft's Azure cloud in Germany Central with up to 18 virtual machines from the type Standard_DS13_v2 which are memory optimized servers with 8 cores (Intel Xeon E5-2673), 56 GB RAM and shared 10 GBit/s Ethernet connectivity (two instances per connect). We deployed a custom Ubuntu 14.04 image with 4.4.0-59 kernel and Java 8. The tests with 1 GBit/s Ethernet and InfiniBand were executed on our private cluster servers with 64 GB RAM, Intel Xeon E5-1650 CPU and Ubuntu 16.04 with kernel 4.4.0-64.

We use a set of micro benchmarks for the evaluations in Sections IX-A and IX-B which send messages or requests of variable size with a configurable number of application threads. All throughput measurements refer to the payload size which is considerably smaller than the full message size, e.g., a 64-byte payload results in 115 bytes to be sent on IP layer when using Ethernet. Additionally, all runs with DXNet's benchmarks are **full-duplex** showing the aggregated performance for concurrently sending and receiving messages/requests.

A. Loopback

As mentioned before, we want to evaluate the efficiency of DXNet's concepts without any network limitations. Fig. 6 shows message processing times and throughputs for different message sizes when using the Loopback transport on a typical cloud server (Standard_DS13_v2). Messages up to 2 KB can be processed in around 500 ns. Larger messages require

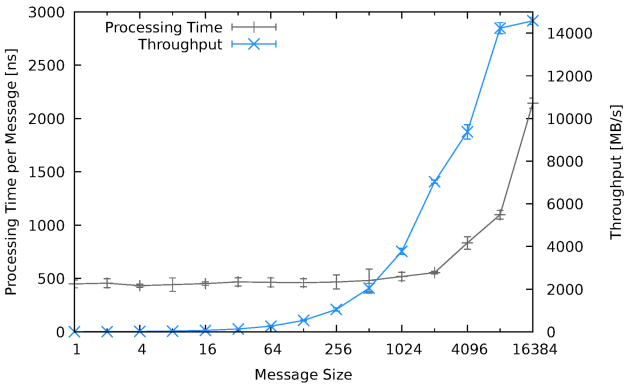

Figure 6. 10^7 Messages, 1 App. Thread, 4 Message Handlers.

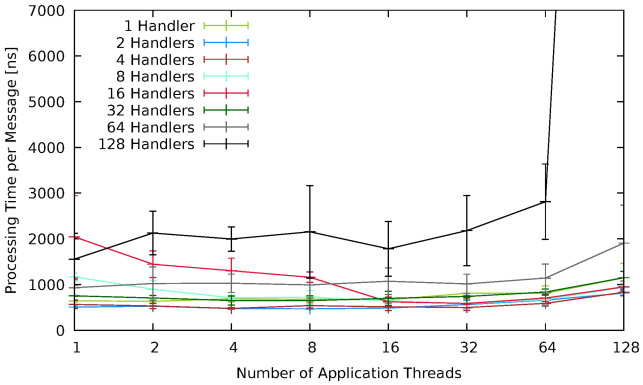

Figure 7. 10^7 Messages, 1024 Bytes Payload.

increasing processing times, as expected. The throughput increases linearly with the message size up to 8 KB messages and is capped at around 14 GByte/s aggregated throughput for sending and receiving of larger messages. The Linux tool `mbw` determined a memory bandwidth of 7.19 GByte/s for a 16 GB array and 16 KB block for the used servers which explains the maximum throughput (saturation of the available memory bandwidth).

In Fig. 6, we studied messages with up to 16 KB payload size as DXNet is primarily designed to perform well with small messages. We also tested larger messages (larger than the ORB, configured with 4 MB here) and measured a message throughput of around 5.4 GByte/s with 8 MB messages. The throughput is lower as application threads and transport thread work sequentially for larger messages (see Section V-A). However, if the application needs to often handle large messages, throughput can easily be improved by using a larger ORB.

DXNet is designed to efficiently support concurrent application threads sending and receiving messages in parallel. Fig. 7 shows that the processing time for 1 KB messages is stable from one to 64 and only slightly increases with 128 application threads. Additionally, Fig. 7 shows the performance with a varying number of message handlers peaking with two to four. Obviously, 128 application threads and 128 message handlers overstress the CPU (8 cores) significantly. The results for all other constellations are as expected showing DXNet's capability of efficiently handling hundreds of concurrent threads.

We also evaluated request-response latency by measuring the **RTT**, which includes sending a request, receiving the

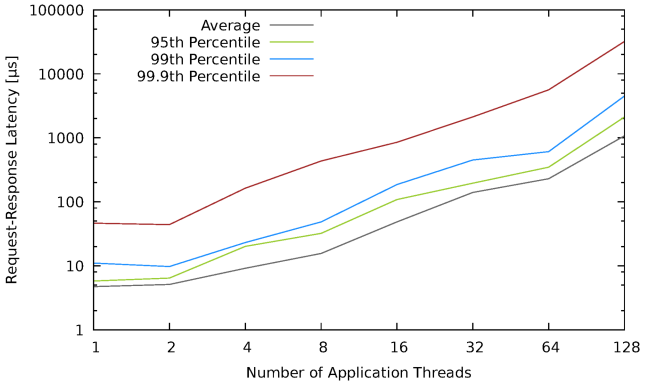

Figure 8. 10^6 Requests, 2 Message Handlers, 1 Byte Payload.

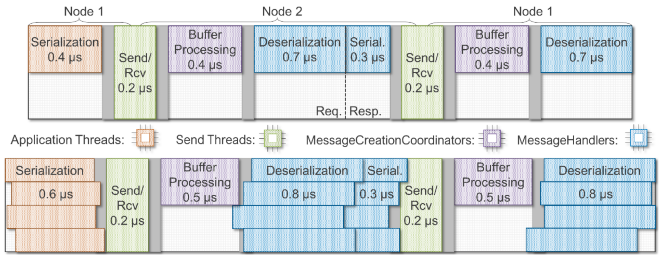

Figure 9. Breakdown of Request-Response Latency for 1024-byte Requests. One application thread (on top) and four (at the bottom). Grey bars indicate inter-thread communication.

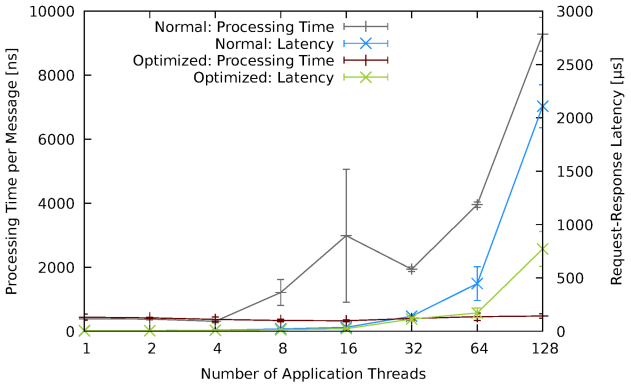

Figure 10. 10^7 Message or 10^6 Requests, 2 Message Handlers, 1 Byte Payload.

request, sending the corresponding response and receiving the response. Fig. 8 shows the latency for small requests with increasing number of application threads. The average RTT with one and two application threads is under 5 µs. With up to eight threads the RTT increases slower than the number of threads because requests can be aggregated for sending. With more threads the increase rate is higher.

Fig. 9 shows the breakdown of request-response latency for one and four application threads and 1024-byte requests. This is a best-effort approximation as time measurement is costly and influences the processing. As expected de-/serialization accounts for the majority of the RTT and deserialization is slower than serialization because of the message object allocation and creation. With more application threads or asynchronous messages, all depicted steps are executed in parallel.

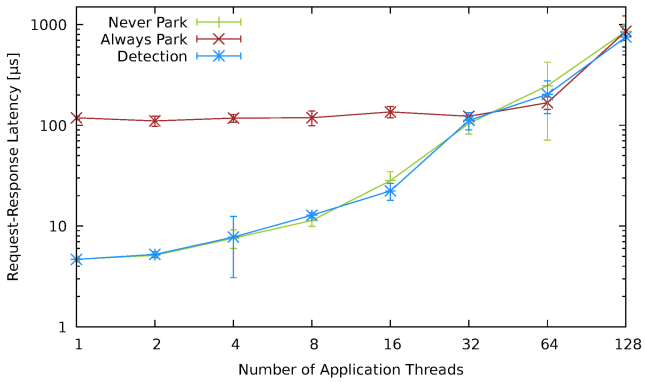

Figure 11. 10^6 Requests, 2 Message Handlers, 1 Byte Payload.

Optimized Outgoing Ring Buffer. The benefits of the CUB, discussed in Section IV, can be seen in Fig. 10. Without the optimization the message processing time increases significantly with more than four application threads sending messages (with 128 threads nearly 20 times higher). Furthermore, the RTT diverges considerably with more than 32 application threads as well.

Overprovisioning Detection. Fig. 11 shows the importance of the thread parking strategy (see Section VII). The RTT is 25 times larger when using one application thread and always parking network threads. All three strategies match with 32 threads and diverge a little with more threads. The never park strategy is at disadvantage with many threads (128) and the RTT is around 100 µs larger than with the adaptive approach.

The evaluation with Loopback transport shows the high throughput and low latency of DXNet. Furthermore, DXNet offers a high stability when used with many threads sending and receiving messages in parallel.

B. *Comparing Network Transports*

Fig. 12 shows the message processing time and throughput for all three network transports (Ethernet and Loopback on cluster and cloud instances) with varying payload size. As expected, InfiniBand has the lowest processing overhead and highest throughput of all physical devices.

The comparison between the 1 GBit/s Ethernet of the private cluster and 5 GBit/s Ethernet in Azure cloud reveals interesting insights. Obviously, message throughput is higher in the cloud for large messages. But, message throughput is higher and processing time is lower on the cluster for messages smaller than 64 bytes which is most likely caused by the virtualization overhead of cloud servers. Loopback is also considerably faster on cluster instances (< 300 ns processing time and > 16 GByte/s throughput).

Fig. 13 shows the request-response latency and throughput for requests sent by four application threads. Again, 1 GBit/s Ethernet on our cluster performs better for small payloads (< 1024) than 5 GBit/s Ethernet in the cloud. For larger requests the bandwidth becomes more and more important favoring the cloud network. Both Ethernet networks are far off the latencies InfiniBand achieves. For small request (< 512 byte payload) the RTT is consistently under 10 µs and rises to only 16 µs for 16 KB requests. Hence, the throughput is much higher with InfiniBand as well.

The evaluation with three physical transports confirms the

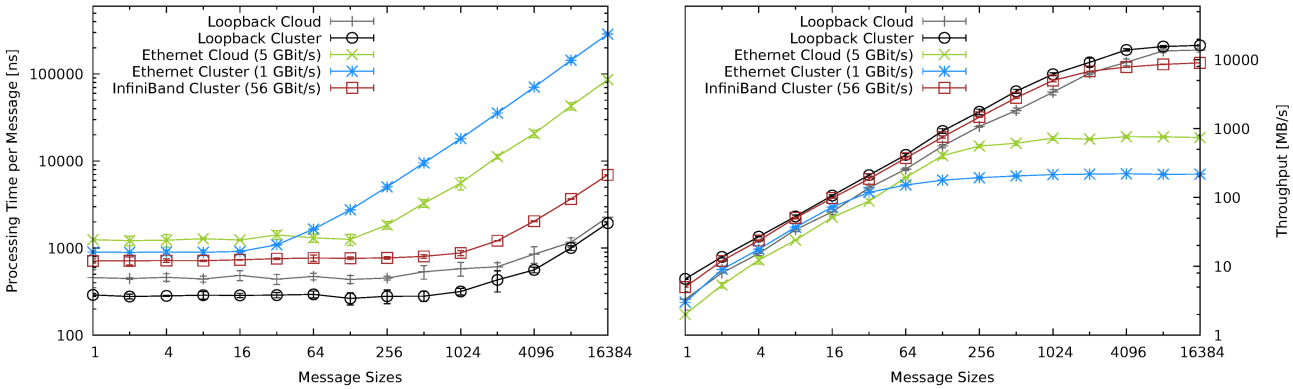

Figure 12. 10^8 Messages, 1 App. Thread, 2 Message Handlers.

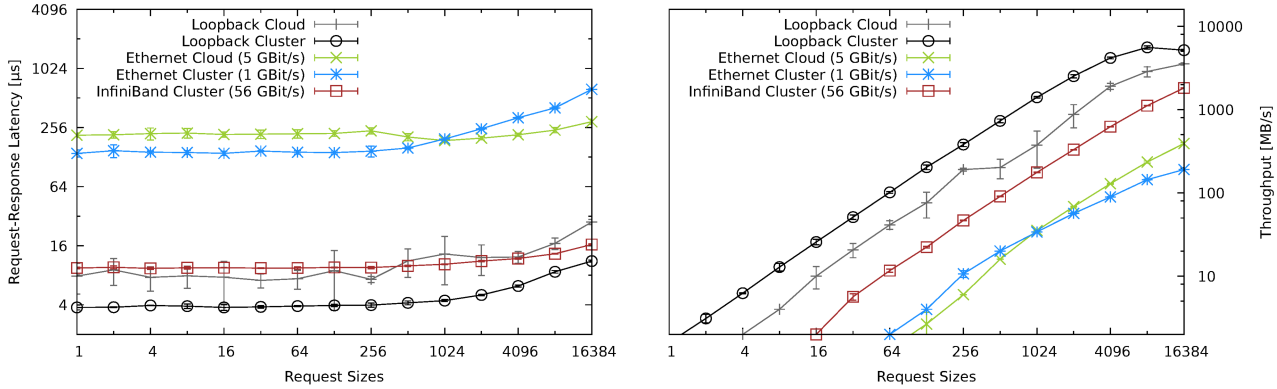

Figure 13. 10^7 Requests, 4 App. Threads, 2 Message Handlers.

results gathered with Loopback and DXNet performs strong especially with InfiniBand (RTT < 10 µs, throughput > 9 GByte/s full-duplex).

C. Yahoo! Cloud Serving Benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) was designed to quantitatively compare distributed serving storage systems [33]. The benchmark offers a set of simple operations (reads, writes, range scans) and a tabular key-value data model to evaluate online storage systems regarding their elasticity, availability and replication. Furthermore, YCSB is easily extensible for new storage systems and new workloads. For our evaluation, we used the in-memory key-value store DXRAM [34] which utilizes DXNet and created an individual workload: one 64-byte object per key, 10^6 keys, uniform distribution, 90 % read and 10 % write operations, 10^7 operations. The tests were run in the Microsoft Azure cloud with one storage server and an increasing number of client servers (maximum 16) which each hosted up to 80 client threads.

Fig. 14 shows the average operation latency and throughput with 10 to 1280 client threads. The operation latency starts at around 230 µs which is in line with previous latency measurements. The latency grows slowly up to 480 client threads but then exponentially indicating server congestion. The throughput rises up to 640 client threads with more than one million operations per second and remains stable with more client threads.

The evaluation with YCSB shows DXNet's high performance for a client-server scenario (one server can serve more than 1000 clients).

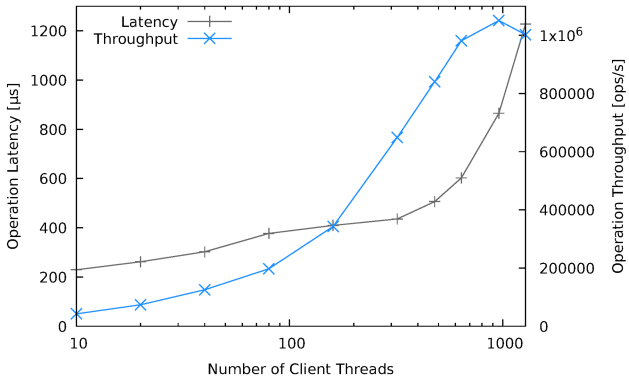

Figure 14. 6 Message Handlers

X. Conclusions

Many big data applications as well as large scale interactive applications are written in Java and aggregate the resources of many servers in a cloud data center, high performance cluster or private cluster. Efficient network communication is very important for these application domains. RMI while being comfortable to use is not fast enough for these applications. Plain sockets are difficult to handle especially if efficiency and scalability need to be addressed. MPI was designed for spawning processes with finite runtime in a static environment. Thus, multi-threading performance and support for adding/removing nodes to an existing environment are limited.

In this paper, we proposed DXNet, a Java open-source network library complementing the communication spectrum.

DXNet provides fast parallel serialization for Java objects, automatic connection management, automatic message aggregation and an event-driven message receiving approach including a concurrent deserialization. DXNet offers high-throughput asynchronous messaging as well as synchronous request/response communication with very low latency. Finally, its architecture is open for supporting different transport protocols. It already supports TCP with Java.nio and reliable verbs for Infiniband. DXNet achieves high performance and low latency by using lock-free data structures, zero-copy and zero-allocation. The proposed ring buffer and queue structures are complemented by different thread parking strategies guaranteeing low latency by avoiding CPU overload.

Evaluations on a private cluster and in the Microsoft Azure cloud show message processing times of sub 300 ns resulting in throughputs of up to 16 GByte/s which saturate the memory bandwidth of a typical cloud instance. For the request/response pattern, DXNet is able to provide sub 10 µs RTT latency using the InfiniBand transport (sub 4 µs over Loopback). Finally, DXNet is also able to efficiently handle highly concurrent processing of many small messages resulting in throughput saturations for Ethernet with 256 bytes payload and InfiniBand with 1-2 KB payload.

The InfiniBand transport IBDXNet is work in progress and final results will be published separately (throughput: >10.4 GByte/s). Future work also includes more experiments at larger scales including comparisons with other network middlewares, as well as evaluations using a 100 GBit/s InfiniBand network.

REFERENCES

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, pp. 1804–1815, Aug. 2015.

[2] M. S. Engler, M. El-Kebir, J. Mulder, A. E. Mark, D. P. Geerke, and G. W. Klau, "Enumerating common molecular substructures," *PeerJ Preprints*, vol. 5, p. e3250v1, Sep. 2017.

[3] P. Satapathy, J. Dave, P. Naik, and M. Vutukuru, "Performance comparison of state synchronization techniques in a distributed lte epc," in *IEEE Conf. on Network Function Virtualization and Software Defined Networks*, 2017.

[4] S. Ekanayake, S. Kamburugamuve, and G. C. Fox, "Spidal java: High performance data analytics with java and mpi on large multicore hpc clusters," in *Proceedings of the 24th High Performance Computing Symposium*, 2016, pp. 3:1–3:8.

[5] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[6] "Cassandra," http://cassandra.apache.org, accessed: 2018-03-14.

[7] "Interactive query with apache hive on apache tez," http://hortonworks.com/hadooptutorial/supercharging-interactivequeries-hive-tez/, accessed: 2018-03-14.

[8] "Impala - cloudera," https://www.cloudera.com/products/open-source/apache-hadoop/impala.html, accessed: 2018-03-14.

[9] S. Microsystems, "Java remote method invocation specification," https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html, accessed: 2018-03-14.

[10] Oracle, "Package java.net," https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html, accessed: 2018-03-14.

[11] S. Mintchev, "Writing programs in javampi," University of Westminster, Tech. Rep. MAN-CSPE-02, Oct. 1997.

[12] K. Beineke, S. Nothaas, and M. Schoettner, "Dxnet project on github," https://github.com/hhu-bsinfo/dxnet, accessed: 2018-03-14.

[13] W. Zhu, C.-L. Wang, and F. C. M. Lau, "Jessica2: a distributed java virtual machine with transparent thread migration support," in *Proceedings. IEEE International Conference on Cluster Computing*, 2002, pp. 381–388.

[14] S. P. Ahuja and R. Quintao, "Performance evaluation of java rmi: A distributed object architecture for internet based applications," in *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '00, 2000, pp. 565–569.

[15] M. Philippsen, B. Haumacher, and C. Nester, "More efficient serialization and rmi for java," *Concurrency: Practice and Experience*, vol. 12, pp. 495–518, 2000.

[16] J. Maassen, R. V. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman, "Efficient java rmi for parallel programming," *ACM Trans. Program. Lang. Syst.*, vol. 23, pp. 747–775, Nov. 2001.

[17] C. Nester, M. Philippsen, and B. Haumacher, "A more efficient rmi for java," in *Proc. of the ACM 1999 Conf. on Java Grande*, 1999, pp. 152–159.

[18] M. P. I. Forum, Ed., *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center, 2015, 2015. [Online]. Available: https://books.google.de/books?id=Fbv7jwEACAAJ

[19] A. Shafi, B. Carpenter, and M. Baker, "Nested parallelism for multi-core hpc systems using java," in *Journal of Parallel and Distributed Computing*, 2009, pp. 532–545.

[20] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and X. Li, "mpijava: A java interface to mpi," http://www.hpjava.org/mpiJava.html, accessed: 2018-03-14.

[21] G. "Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur, "Enabling concurrent multithreaded mpi communication on multicore petascale systems," in *"Recent Advances in the Message Passing Interface"*, 2010, pp. 11–20.

[22] H. V. Dang, S. Seo, A. Amer, and P. Balaji, "Advanced thread synchronization for multithreaded mpi implementations," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 314–324.

[23] R. Latham, R. Ross, and R. Thakur, "Can mpi be used for persistent parallel services?" in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin Heidelberg, 2006, pp. 275–284.

[24] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi, "Using mpi in high-performance computing services," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13, 2013, pp. 43–48.

[25] K. Beineke, S. Nothaas, and M. Schoettner, "Fast parallel recovery of many small in-memory objects," in *International Conference on Parallel and Distributed Systems (ICPADS)*, vol. 23, in press.

[26] Oracle, "Java i/o, nio, and nio.2," https://docs.oracle.com/javase/8/docs/technotes/guides/io/index.html, accessed: 2018-03-14.

[27] R. Hitchens, *Java NIO*. Sebastopol, CA, USA: O'Reilly Media, 2009.

[28] G. L. Taboada, J. Touriño, and R. Doallo, "Java fast sockets: Enabling high-speed java communications on high performance clusters," *Comput. Commun.*, vol. 31, pp. 4049–4059, Nov. 2008.

[29] G. L. Taboada, J. Tourino, and R. Doallo, "High performance java sockets for parallel computing on clusters," in *Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.

[30] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom, "Use at your own risk: The java unsafe api in the wild," *SIGPLAN Not.*, vol. 50, pp. 695–710, Oct. 2015.

[31] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat, "Pickling state in the javatm system," in *Proc. of the 2nd Conf. on USENIX Conf. on Object-Oriented Technologies*, 1996, pp. 19–19.

[32] "Kryo - java serialization and cloning: fast, efficient, automatic." https://github.com/EsotericSoftware/kryo, accessed: 2018-03-14.

[33] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[34] K. Beineke, S. Nothaas, and M. Schoettner, "High throughput log-based replication for many small in-memory objects," in *IEEE 22nd International Conference on Parallel and Distributed Systems*, 2016, pp. 535–544.

5.4.3 Ibdxnet: An InfiniBand Network Subsystem for DXNet

This section presents two publications, one paper [92] and an extended report [90], proposing the design of Ibdxnet. Ibdxnet is a transport for the DXNet messaging system to allow highly concurrent Java applications to leverage the performance of low-latency InfiniBand hardware. Ibdxnet consists of a native library with a dedicated subsystem to drive InfiniBand hardware using the C-verbs library.

Our publications present a scalable pipeline for low-latency and high-throughput sending and receiving of data using a ring-buffer data-structure. The author had to design and develop InfiniBand support for DXRAM's network subsystem from scratch using the C-verbs library to ensure the highest degree of control over the hardware possible.

With documentation regarding efficient InfiniBand development being sparse and many (toy-) examples available, which explain basic concepts only, development and optimization to reach optimal performance were very challenging. Context switching from Java to native space and vice versa imposes further latency considerations impacting performance (e.g., garbage collection). Several key concepts which are essential for performance were, to the best of our knowledge, either unknown or previously unpublished.

This thesis addressed all these challenges. Thus, DXNet with the IB transport using Ibdxnet provides low-latency and high throughput for highly concurrent Java applications. In a concurrent environment and on worst-case all-to-all communication, DXNet with the IB transport even outperforms the well established MPI implementation MVAPICH2 (written in C). With DXNet and Ibdxnet being open-source and published as separate libraries, any Java application can benefit from this high degree of abstraction with single-digit latency and high throughput.

In his master thesis [111], Michael Schlapa contributed initial research and evaluation of available libraries to use InfiniBand hardware from Java applications. His thesis provided the foundation for further prototypes and discussions on how to design and implement InfiniBand support for DXRAM's network subsystem.

The results lead to the design and implementation of a custom networking subsystem aiming at providing low latency and high throughput when using InfiniBand in Java applications. Michael Schlapa developed small prototype applications using the C-verbs library which served as a reference for Stefan Nothaas to start his research.

Stefan Nothaas designed and implemented the native C++ library Ibdxnet which involved several redesigns and refactoring iterations during the development process as documentation and programming examples for InfiniBand hardware were sparse. To create a transparent and abstract API for the application using Ibdxnet, DXNet had to undergo major refactoring phases to support multiple types of transport (Ethernet and InfiniBand) sharing a common core with data structures and parts of the processing pipeline as well as the DXNet API (see Section 5.4.2).

Dr. Kevin Beineke and Prof. Dr. Michael Schöttner took part in many discussions about the design and evaluation of DXNet with Ibdxnet. As already described in detail in Section 5.4.2, the core of DXNet was a close collaboration of Dr. Kevin Beineke and Stefan Nothaas.

Stefan Nothaas wrote the paper and report and evaluated the systems presented. Stefan Nothaas created all figures of the report excluding Figures 1 and 10 which were contributed by Dr. Kevin Beineke. Prof. Dr. Michael Schöttner and Dr. Kevin Beineke helped in improving comprehensibility and reviewed the report and paper several times.

Leveraging InfiniBand for Highly Concurrent Messaging in Java Applications

Stefan Nothaas
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
stefan.nothaas@hhu.de

Kevin Beineke
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
kevin.beineke@hhu.de

Michael Schoettner
Department of CS Operating Systems
Heinrich-Heine-Universität
Duesseldorf, Germany
michael.schoettner@hhu.de

***Abstract*—In this paper, we describe the design and implementation of Ibdxnet, an InfiniBand transport to enable high throughput and low-latency messaging for concurrent Java applications with transparent serialization of Java objects using DXNet. Ibdxnet applies best practices by implementing a dynamic and scalable pipeline with RC QPs and messaging verbs using the ibverbs library. A carefully designed JNI layer ensures minimal overhead to connect the native Ibdxnet library to the Java counterpart without impacting performance. Existing as well as new multi-threaded Java applications can use DXNet's event-based architecture concurrently to send and receive messages and requests transparently over InfiniBand with the Ibdxnet transport. We compared DXNet with Ibdxnet to the InfiniBand supporting MPI implementations FastMPJ and MVAPICH2. DXNet's performance for middle sized and large messages keeps up with FastMPJ's and MVAPICH2's. For small messages, DXNet clearly outperforms both systems especially in a multi-threaded environment. Furthermore, we compared the two key-value storages DXRAM, which uses DXNet with Ibdxnet, and RAMCloud, which uses a custom network subsystem based on ibverbs, using the YCSB with two workloads. On a graph data workload, DXRAM outperforms RAMCloud with a five times higher throughput of 7.96 mops on 40 nodes.**

***Index Terms*—High-speed networks, Big data applications, Distributed computing**

I. Introduction

The ever growing amounts of data, for example in big data applications, are addressed by aggregating resources in commodity clusters or the cloud [23]. This concerns applications like social networks [9], [27], search engines [35], simulations [36] or online data analytics [17], [44], [46]. Storing all data in-memory or caching major parts reduces local data access times significantly especially for graph-based applications processing billions of very small data objects ($<$ 64 bytes) [13], [32], [42].

A lot of big data applications are written in Java or rely on Java-based frameworks, e.g. Hadoop [30], Apache Kafka [25], Hazelcast [19] or InfiniSpan [1]. Typically, these applications use many threads to fully utilize all available cores on each server. There are embarrassingly parallel algorithms working well on batch processing frameworks, e.g. based on MapReduce. However, many algorithms have data dependencies and require low-latency remote data access in order to allow scalability with the number of nodes. Examples are machine learning algorithms (e.g. neural networks, deep learning) [43], data-mining programs (e.g. PageRank [35], random walk, graph coloring) as well as graph-based bioinformatics applications. As mentioned above, graphs are composed of potentially trillions of small data objects. Thus, the algorithm exchanges many small network packets with remote nodes often resulting in challenging all-to-all communication patterns.

Java applications as well Java-based big-data frameworks use mainly TCP sockets over Ethernet provided by Java NIO. Java NIO sockets are fast but cannot exploit the potential of high-speed networks like InfiniBand. One transparent approach is to use IP over InfiniBand (IPoIB) [24] which is known to bring some benefits but far from the full potential of InfiniBand [45]. This can only be achieved by using native InfiniBand verbs. We analyzed the few available solutions, such as a built in verbs implementation in the JVM or libraries redirecting socket traffic over InfiniBand but none can provide optimal performance especially for high concurrency and many small network packets (§II).

In this paper, we present Ibdxnet, an InfiniBand-based transport implementation for the Java-based network subsystem DXNet [12]. DXNet provides high throughput and low-latency asynchronous and synchronous messaging for many concurrent threads accessing the network [13], [16], [18]. DXNet offers sending and receiving of messages with transparent serialization of messaging objects, automatic connection management, is optimized for high concurrency on all operations by implementing lock-free synchronization and is implemented in Java. Ibdxnet implements a native subsystem to interface with the ibverbs library for sending and receiving data over InfiniBand hardware. A carefully designed JNI layer is used to efficiently connect it to the transport implementation of DXNet in Java.

We compared DXNet with Ibdxnet to MVAPICH2 and FastMPJ, two MPI implementations supporting InfiniBand (§VI). We used typical uni- and bi-directional network benchmarks to evaluate throughput and latency with messages sizes up to 1 MB. DXNet outperforms FastMPJ already in single-threaded mode, especially on small messages and is on par with the performance of MVAPICH2. Using multiple application threads and message handlers, DXNet's throughput can be further increased outperforming MVAPICH2 up to twofold.

Furthermore, we compared the Java-based in-memory key-

value storage DXRAM, which uses DXNet as its network subsystem, to the in-memory key-value storage RAMCloud. We used the Yahoo! Cloud Service Benchmark (YCSB) with two workloads to evaluate the pure network performance of both systems (§VII). DXRAM outperforms RAMCloud on both workloads and shows scalability with up to 40 nodes.

The contributions of this paper are:

- The design and implementation of a fast InfiniBand transport for distributed and parallel Java applications not requiring any modification to the JVM
- Automatic, scalable and concurrent connection management allowing transparent dynamic up- and downscaling
- Extensive experiments and comparisons to MVAPICH2, FastMPJ and within key-value stores (DXRAM and RAMCloud) showing the benefits of the proposed solution

With DXNet and the transport Ibdxnet being open source and available at Github, we provide a network subsystem with InfiniBand for simple and fast data exchange using messages and requests for new and existing Java applications.

The remaining paper is structured as follows: Section II presents related work followed by Section III giving a brief introduction to DXNet with its key-features. Section IV gives an overview of the architecture and describes the details of the ibverbs-based transport Ibdxnet in further sub-sections. Section V describes briefly how the native Ibdxnet library connects to the Java transport interface in DXNet. Section VI presents the results of the experiments with MVAPICH2 and FastMPJ. Section VII presents the results of the YCSB comparing DXRAM (using DXNet) to RAMCloud. Conclusions are located in Section VIII.

II. Related Work

We consider two categories for related work: low(er)-level network stacks and networking middleware with higher-level primitives and programming models. Because of limited space, we focus only on solutions which support InfiniBand.

Before developing Ibdxnet and the InfiniBand transport for DXNet, we evaluated available (low-level) solutions for leveraging InfiniBand hardware in Java applications. This includes using NIO sockets with **IP over InfiniBand (IPoIB)** [24], the **Socket Direct Protocol (SDP)** [22], IBM's implementation of the verbs API in Java called **jVerbs** [39], IBM's **Java sockets over RDMA (JSOR)** [41], Mallanox's **libvma** [2] library for sockets and native C-verbs with **ibverbs**. All solutions were evaluated with uni- and bi-directional microbenchmarks on a 56 GBit/s InfiniBand network. The results show that socket-based solutions and jVerbs cannot provide an adequate base performance in throughput and latency compared to native C-verbs implementations, especially on small message sizes and on bi-directional communication. Furthermore, every solution requires some compromises like a proprietary environment (jVerbs, JSOR) or is not maintained anymore (SDP since OFED 3.5). These reasons motivated a custom implementation using the ibverbs library. Due to limited space, the results are published in a separate publication.

MVAPICH2 [31] is an MPI implementation built onto the MPICH source supporting various network interconnects, such as Ethernet, iWARP, Omni-Path, RoCE and InfiniBand. MVAPICH2 includes features like RDMA fast path or RDMA operations for small message transfers and is widely used on many clusters over the world [3]. **Open MPI** [4] is an open source implementation of the MPI standard (currently full 3.1 compliance) also supporting interconnects, such as Ethernet using TCP sockets, RoCE, iWARP and InfiniBand.

mpiJava [10] implements the MPI standard by a collection of wrapper classes that call native MPI implementations, such as MVAPICH2 or OpenMPI, through JNI. The wrapper-based approach provides efficient communication relying on native libraries. However, it is not thread-safe and, thus, is not able to take advantage of multi-core systems using multi-threading.

FastMPJ [20] uses Java Fast Sockets and ibvdev to provide an MPI implementation for parallel systems using Java. Initially, **ibvdev** [21] was implemented as a low-level communication device for **MPJ Express** [38], a Java MPI implementation of the mpiJava 1.2 API specification. ibvdev implements InfiniBand support using the low-level verbs API and can be integrated into any parallel and distributed Java application. FastMPJ optimizes MPJ Express collective primitives and provides efficient non-blocking communication. Currently, FastMPJ supports issuing MPI calls using a single thread, only.

We also considered using an MPI implementation like MVAPICH2 as a DXNet transport implementation. However, we encountered limitations that currently make this approach unfeasible: dynamic scaling, e.g. adding and removing additional nodes during runtime, is not supported by current implementations. Bootstrapping a process which uses MPI as a network subsystem without an MPI communicator creates isolated MPI worlds which cannot be connected afterwards with the currently available implementations. Support for multi-threading in the same process is only supported by MVAPICH2, but not implemented efficiently (§VI).

III. DXNet: Concurrent Messaging for Java Applications

DXNet [12] is a network library for Java targeting, but not limited to, highly concurrent big data applications. DXNet implements **asynchronous event-driven messaging** with a simple to use application interface. **Messaging** describes **transparent sending and receiving of complex (even nested) data structures** with implicit serialization and de-serialization. Furthermore, DXNet provides a built-in primitive for concurrent and transparent **request-response communication**.

DXNet is optimized for supporting highly multithreaded sending and receiving of small messages by using **lock-free data structures, fast concurrent serialization, zero copy and zero allocation**. It implements a custom **lock-free Outgoing Ring Buffer (ORB** to efficiently handle outgoing data and its own **flow control (FC)** mechanism to avoid overburdening remote nodes if they cannot keep up with processing incoming messages which increases latency and decreases throughput. The core of DXNet provides **automatic**

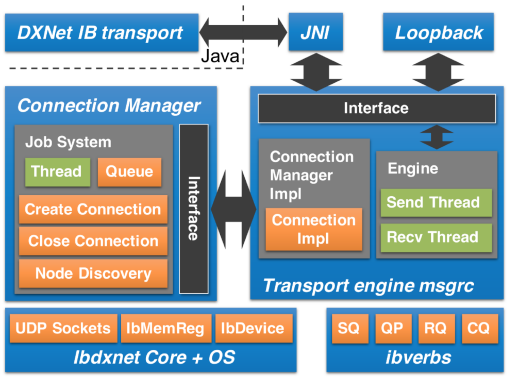

Fig. 1. Simplified architecture of Ibdxnet with the msgrc transport engine

connection and buffer management, serialization of message objects and an interface for implementing different transports. Currently, an Ethernet transport using Java NIO sockets and an InifiniBand transport using *ibverbs* (§IV) is implemented.

IV. Ibdxnet: Native InfiniBand Subsystem with Transport Engine

In this section, we give a brief overview of the design and implementation of the native Ibdxnet subsystem. Figure 1 depicts a simplified view of the architecture including the relevant components of the Idxbnet subsystem.

Java environment and native code. Because existing libraries do not provide optimal performance when bringing InfiniBand support to Java (§II), we decided to develop a custom solution based on the native ibverbs library. However, this requires calling native code from the Java environment which is typically done by using the Java Native Interface (JNI) [26]. Using JNI is often referred to being rather slow but many examples available do not apply best practices. We minimized the overall overhead by applying known best practices [15], [26] like **a careful design of the interface with minimized context switching, caching of method IDs, avoid copying of Java arrays nor accessing Java data structures from native space**. We **avoid relying on Java's garbage collection** when using native code entirely, e.g. no heap allocations in our pipeline's data path and using heap allocated objects in native code, as it increases latency and harms performance here. Furthermore, we evaluated different means of passing data from Java to native and vice versa as well as the function/method call overhead. Due to space constraints, the full results are published in a separate report [33] and we only summerize the most important aspects here. The results show that the average single costs for context switching are neglectable with an average switching time of only up to 0.1 µs. **We exchange data using only primitive function arguments**. Data structures are mapped and accessed as C-structs in the native space. In Java, we access the native C-structs using a helper class which utilizes the Unsafe library [29] as this is the fastest method in both spaces.

Messaging verbs. Many applications using InfiniBand prefer RDMA over messaging verbs for performance reasons. However, we decided to rely on **messaging verbs instead** for two reasons: First, messaging verbs fit better into the existing architecture. **Our messaging based processing pipeline cannot benefit from RDMA verbs** not involving the CPU. Instead, it would require additional synchronization mechanisms to detect processed/incoming buffers increasing complexity of the pipeline. Second, **it is not guaranteed that RDMA verbs always improve the performance over messaging verbs** [40] especially when a complex system is built on top of it. Using messaging verbs, we have implemented the *msgrc* engine based on RC queue pairs (QPs) and the *msgud* engine based on UD QPs. Due to space constraints, we focus only on the design and implementation of the msgrc engine. The msgud engine will be described and evaluated in a separate publication.

Connection management. With dynamic up- and downscaling of the system regarding node counts and highly concurrent environments with hundreds of threads, transparent and efficient connection management is necessary. We elaborate on the various challenges imposed and the design of our connection management addressing them in Section IV-A.

Data processing pipeline. This is the heart of the transport and crucial regarding performance. We introduce the design of a fully **asynchronous pipeline** using a **dedicated thread for sending** (§IV-B) and **receiving** (§IV-C). The pipeline is designed to ensure **scalability with the number of nodes** and **high hardware utilization** on high loads. Otherwise, a very brief stall of the pipeline would impose overall major performance penalties. We use one dedicated thread for sending (§IV-B) and one for receiving (§IV-C) to benefit from the following advantages: a clear separation of responsibilities resulting in a less complex architecture, no scheduling of send/receive jobs when using a single thread and increased concurrency because we can run both threads on different CPU cores. The architecture also allows us to create decoupled pipeline stages using lock-free queues and ring buffers. Thereby, we **avoid complex and slow synchronization** between the two threads and potentially many application threads because they do not have to access shared resources concurrently.

A. Automatic and Scalable Connection Management

Motivation. When using ibverbs and RC QPs, the application has to exchange connection information in order to establish a connection between two QPs using an out-of-band channel. Typically, this is done by using IP-addresses and a TCP/UDP socket over Ethernet. We implement automatic node discovery with IP address to abstract nodeID mapping which is part of the connection creation process. Furthermore, we exchange additional data on connection creation like the nodeID and transport engine implementation dependent data along with the physical QP ID.

Challenges. With many threads concurrently accessing connections to send data to many nodes simultaneously, efficient and low-overhead connection management is important to not impact performance. For example, multiple application threads want to send data to a node but the connection is not yet es-

tablished. Who creates the connection and synchronizes access of other threads? How to avoid synchronization overhead or blocking of threads that want to get an already established connection? How to manage the lifetime of a connection? How to handle open connections if a node is not available anymore? All these challenges are addressed by a **dedicated connection manager in Ibdxnet**. It handles all tasks required to transparently establish and manage connections.

Node identification and connection. A node is identified by **a unique 16-bit integer nodeID (NID)** (configurable size). The NID is assigned to a node during startup of the connection manager and cannot be changed during runtime. A connection consists of the source NID (the current node), the destination NID (the target remote node) and transport specific attributes, e.g. one or multiple ibverbs QPs, buffers and other data necessary to send and receive data using that connection. The connection manager provides a **connection interface for the transport engines** which allows them to implement their own type of connection including required attributes.

Node discovery. The following example describes a connection with a single QP. Before a connection to a remote node can be established, the remote node must be discovered and known as available. We use a dedicated connection-manager thread which executes the node discovery in configurable intervals and manages a list with discovered nodes (NID to IP-address mappings). An initial list with hostnames of available nodes is provided on startup and can be extended during runtime for dynamic scaling. A node must be discovered and have a NID assigned before creating a connection.

Connection creation. Connection creation is also handled asynchronously by the connection-manager thread. If a thread wants to use a connection to send data, it queries the connection manager for the connection. If the connection is not yet established, the manager thread executes these tasks asynchronously by executing a two phase protocol to exchange all necessary connection information, such as physical QP ID, NID and transport engine defined data, with the remote. A remote QP is also created if necessary during that process. The QP is set to ready-to-receive and ready-to-send before completing connection creation. The application thread waits until the connection is created. Concurrent threads can easily detect if connection creation is in progress and wait for completion when trying to acquire the same connection concurrently.

Connection failure and cleanup. Once node failure is discovered, typically by a failed send work request (WR), the connection-manager thread is ordered to cleanup the connection. It sets the connection state to *not available* and frees all allocated resources of it. Now, the connection-manager thread has to re-discover the failed node before a connection can be established, again.

B. Sending of Data Using RC QPs

This section describes the application of different concepts for sending data that are the key to performance in our implementation. This also includes some basic concepts which

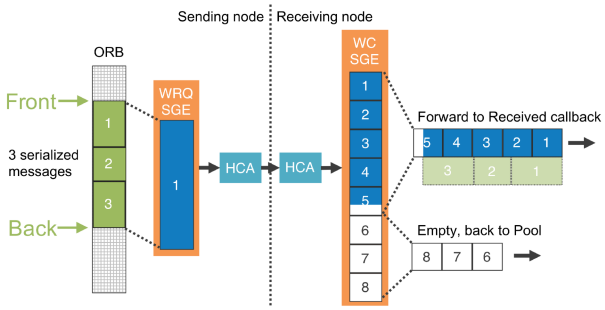

Fig. 2. Example for sending and receiving data using scatter gather elements: Get data (aggregated messages) from ORB, send 1 SGE. Receive data scattered to multiple receive buffers.

are already known but we consider it worthwhile mentioning them because they are important to performance.

Challenges. There are a number of conceptual challenges to consider for the implementation to ensure high performance when sending data: Handling of outgoing data should not introduce additional overhead, e.g. buffer allocation, registering buffers with the protection domain in the data path or copying of data. As the receive buffer size on the remote must be at least the size of the outgoing data, larger data sizes must be considered by either executing a protocol to request a target buffer size or slicing of larger data chunks. Sending of additional out-of-band data is required to identify the incoming data on the remote without having to wrap the data. Ensure high utilization of involved data structures to keep the pipeline busy, e.g. keep the send queue (SQ) and buffers filled whenever possible. Full asynchronous processing is the key to avoid waiting for any sort of completion. Instead, execute the next task and collect the results of previously executed tasks later.

```
workPackage = GetNextDataToSend(
   prevWorkResults, completionList);
Reset(prevWorkResults);
Reset(completionList);
if (workPackage) {
   connection = GetConnection(
      workPackage.nodeId);
   prevWorkResults = SendData(
      connection, workPackage);
   ReturnConnection(connection);
}
completionList = PollCompletions();
```

Listing 1. Send thread main flow (simplified)

Decoupled pipeline stages and synchronization. A dedicated send thread executes the send control flow separately from the application threads. An application thread has to provide outgoing data via a per-connection dedicated ring buffer data structure. This ring buffer is implemented using lock-free synchronization and provides mechanisms to ensure high concurrency for up to hundreds of application threads [12]. This creates two decoupled pipeline stages with a single point of synchronization and avoids expensive synchronization

(excluding ibverbs; these are already thread safe).

Asynchronous and interleaved control flow. Listing 1 depicts a simplified version of the send thread's asynchronous control flow main loop with the relevant aspects. The send thread pulls the data available to send asynchronously from the ring buffer with the call *GetNextDataToSend*. The *workPackage* holds all relevant information of a data package pulled from the ring buffer (pointer positions and target remote). The *prevWorkResults* data describes the amount of data that was successfully posted or not posted because there was no space left in the send queue. The *completionList* is filled, once sending of posted data is confirmed (completion polled). This data is used to move the pointers in the ring buffer. The *prevWorkResults* and *completionList* are cleared at the beginning of the current iteration. If the *workPackage* is valid, i.e. outgoing data is available, the corresponding *connection* is requested according to the target *workPackage.nodeId*. With this call, the connection is automatically established by the connection manager if it does not exist, yet (§IV-A). The *workPackage* is processed in *SendData* by setting the buffer area to send in a WR to the area enclosed by the ORB pointers of the *workPackage*. We avoid allocating a temporary buffer and copying because the ring buffer is allocated and registered with the protection domain on startup and setting arbitrary positions within that pinned area is valid. Once the data is posted, the acquired connection is returned to the manager. Concluding the control flow, the send thread always polls for completions on the shared completion queue (CQ), once. Further details are described in the next paragraphs.

Immediate data field for out-of-band data A WR offers a field for sending immediate data (a 4 byte value) that is not part of the registered memory area of the buffer to the remote. We use this feature to include the NID of the source node sending the data (2 bytes) and FC data if available (1 byte). This avoids adding that data to the payload stored in the buffer which requires a temporary buffer and copying or an additional side channel like another QP which requires additional processing time on posting and polling. This **benefits overall performance, especially with many simultaneous connections**. By including the source NID with every WR posted, we can identify incoming completions on the remote. Otherwise, the only information provided is the sender's unique physical QP ID. In our case, this ID must be mapped to the corresponding NID of the sender. However, this introduces an indirection every time a package arrives which impacts performance.

Scatter gather elements (SGEs) for improved buffer management. For sending data, we use SGEs to enable flexible buffer management for the receiver as well as benefit from sending large messages or many small aggregated messages. Figure 2 depicts an example with three (aggregated) ready to send messages in the ORB. We create a WR for the outgoing data and provide a single **SGE which takes the pointers of the enclosed memory area**. In the example, the total size received on the remote exceeds the size of a single receive buffer. However, as we also post a corresponding receive WR with a SGE list of 8 elements on the remote, all data can be processed with a single WR and is transparently scattered to 5 receive buffers of equal size. The 5 buffers with data are forwarded for processing and the 3 unused ones go back to the buffer pool (§IV-C). This avoids high fragmentation degrees when sending many small messages because empty buffers can be cut off and re-used immediately. The number of SGEs of a WR is set to 0 to send FC data with the immediate data field but without any payload data.

Chaining of work requests. We create and chain multiple WRs within a single call to *ibv_post_send* to reduce call overhead. Multiple WRs are used if the outgoing data exceeds the maximum configurable size (*num_sges* $\times$ *recv_buffer_size*) that can be received by the remote. The outgoing data is split into multiple WRs which are chained and posted to the SQ.

Shared completion queue. All SQs share the same CQ. When data is posted to multiple connections and completions are polled, we avoid having to iterate a per SQ/connection dedicated CQ. The *PollCompletions* call has to poll only a single CQ which avoids overhead and further complexity.

Asynchronous completion polling. The **PollCompletions** function calls *ibv_poll_cq*, **once, to poll for any completions available** on the SCQ. The send thread tracks the number of posted WRs and knows how many WCs are outstanding on the SCQ. If none are being expected, polling is skipped. *ibv_poll_cq* is called only once per PollCompletion call and every call tries to poll WCs in batches to keep the call overhead minimal. Experiments have shown that most calls to *ibv_poll_cq*, even on high loads, will return empty, i.e. no WRs have completed. Thus, "synchronous" polling after posting a WR and until at least one completion is received is the wrong approach. It wastes CPU time that can be spent on other tasks of the pipeline, e.g. further filling up the SQ of the same or another connection increasing overall utilization. Using completion events instead of polling does not solve this and instead increases latency. This performance impact increases with the number of simultaneous connections being served. Furthermore, this increases the chance of SQs running empty because time is wasted on waiting for completions instead of keeping all SQs filled. **Full SQs ensure that the HCA is kept busy which is the key to performance**. However, we have seen that "synchronous" polling is commonly used in many examples as well as larger projects [5], [7].

C. Receiving of Data Using RC QPs

Analogous to Section IV-B, this section describes the concepts for receiving data that are the key to performance. Most concepts overlap with the ones that were already presented in Section IV-B but have a slightly different implementation. Thus, we keep them brief and just mention the relevant differences. Again, we include well known concepts to show their importance regarding performance.

Challenges. Receiving data adds additional challenges to the ones imposed by sending data with some overlapping. Managing buffers used for receiving data: allocations, registering with the protection domain and data copying is harmful

to performance in the data path. To ensure low latency, the CQ must be polled at a high frequency depending on the time required to process any incoming buffers which increases the time the CQ is not polled. Furthermore, fast refilling of the RQ to avoid HCA stalls on receiving new data is important, too. Like on sending data, full asynchronous processing is the key to avoid waiting for completions and waste processing time.

```
workCompletions = PollCompletions();
if (recvQueuePending < ibqSize) {
   Refill();
}
if (workCompletions > 0) {
  ProcessCompletions(workCompletions);
}
if (!RingBufferIsEmpty()) {
   PushReceivedBuffers();
}
```

Listing 2. Receive thread main flow (simplified)

Decoupled pipeline stages and synchronization. Similar to Section IV-B, a dedicated receive thread executes the receive control flow. The received data is passed to application threads in the next stage via a data structure, e.g. a queue. This introduces a single point of synchronization which can be implemented using lock-free techniques for low overhead and avoids additional synchronization in the receive control flow.

Asynchronous and interleaved control flow. Listing 2 depicts a simplified version of the receive control flow executed in a loop by the dedicated receive thread. The loop starts by calling *PollCompletions* to poll the SCQ for WCs. Before processing the WCs returned, the SRQ is refilled with buffers from a pool by calling *Refill*, if the SRQ is currently not completely filled. Next, if any WCs were polled previously, they are evaluated in the *ProcessCompletions* call. Afterwards, the buffers with the data received are pushed to a lock-free ring buffer. Separate handler threads remove and continue processing them, e.g. de-serialization and dispatching. **None of these calls are blocking**, e.g. the thread polls the CQ once instead of busy polling until a completion is available. **Keeping the SRQ filled is important to avoid HCA stalls which impact performance**.

Buffer pool and fast RQ refill. For buffer management, we use a buffer pool of configurable total and buffer size. A single large memory area (total pool size) is allocated and registered with the protection domain on startup. This area is sliced into multiple equally sized buffers of the configured buffer size for the pool. The buffers are used to refill the SRQ. Every **WR consists of a configurable number of SGEs** which make up the maximum receive size. This is also the limiting size the send thread can post with a single WR (sum of sizes of SGE list). Using this method, the receive thread **does not have to take care of any software slicing** of received data because the **HCA scatters one big chunk of send data transparently** to multiple (smaller) receive buffers on the receiver side. At last, *Refill* chains the WRs to a linked list which is posted on a single call to *ibv_post_srq_recv* for minimal overhead.

Shared receive and completion queue. We use a SRQ and SCQ for receiving incoming data from all connections. This avoids the overhead of iterating and polling on multiple queues for incoming data. If the SRQ is not completely filled, we refill it using the buffer pool mentioned previously.

Asynchronous completion polling. The same concept as explained for the send thread in Section IV-B is also applied for receiving data.

V. IB Transport Implementation in DXNet (Java)

This section describes the most important aspects of the transport implementation for DXNet in Java which utilizes the low-level transport engines, e.g. msgrc provided by Ibdxnet (§IV). We describe how the data from Java is forwarded to Ibdxnet for sending as well as incoming data is forwarded to Java from Ibdxnet.

Connection handling. All tasks related to connection handling are implemented in Ibdxnet (see IV-A). The Java transport implementation is just forwarding connection creation requests to the native subsystem. Callbacks received from Ibdxnet (e.g. node discovered, connection created, connection closed) are connected to their Java counterparts.

Asynchronous pulling of data from the ORB. The send thread calls the JNI function *GetNextDataToSend* to switch to the Java space. There, the Write-Interest-Manager (WIM) manages interest tokens using atomic counters and a FIFO queue to keep track of the ORBs of all connections that have ready-to-send (RTS) data available. The send thread periodically enters the Java space to poll the WIM for the next ORB to process. If RTS data is available, it reads the current pointer positions of the ORB storing the data and returns to the native space for sending (see control flow in §IV-B). On the next switch to Java space, the thread reports back about the previously posted data and also about any completions that confirm that posted data was actually sent. The pointers of the ORB of the associated connections are moved according to the amount of data confirmed sent.

Asynchronous processing of received buffers. When the receive thread receives incoming buffers, it switches to the Java space by calling the JNI call *PushReceivedBuffers* (§IV-C), pushes the buffers to the lock-free FIFO-based Incoming Buffer Queue (IBQ) and returns back to the native space. The buffers in the IBQ are processed by dedicated threads in Java which de-serialize and dispatch them to application pre-registered handler methods (§III).

VI. Network Microbenchmarks

For better readability, we refer to DXNet with the transport Ibdxnet and msgrc engine as DXNet from here onwards.

We implemented commonly used network microbenchmarks to compare **DXNet** to two MPI implementations supporting InfiniBand: **MVAPICH2** and **FastMPJ**. We decided to compare to MPI implementations for the following reasons: To the best of our knowledge, there is no other library/system available that offers similar messaging features like DXNet.

Often, big data applications implement a custom network stack to simplify system-specific optimizations and shorten data paths. If not based on MPI, their stack is not available as a separate application/library like DXNet. MPI can be used to partially cover some features of DXNet but not all (§II). We are aware that MPI is mainly targeting the HPC application domain whereas DXNet is targeting big data applications. However, MPI is also used in big data applications [8], [28], [37] and several aspects related to the network stack and the technologies are overlapping in both application domains.

Based on the *osu* benchmarks included with MVAPICH2, we implemented benchmarks to measure uni- and bi-directional throughput as well as uni-directional latency for DXNet and FastMPJ for basic point-to-point communication. Furthermore, we extended the benchmarks to also support all-to-all communication with more nodes and multiple threads.

We used up to 8 servers of our private cluster, each equipped with one Intel Xeon E5-1650 (3.5 GHz) 6 core CPU, 64 GB RAM and Mellanox ConnectX-3 HCA. All nodes are linked up using a single 56 Gbit/s Mellanox FDR switch. The nodes are running Ubuntu 16.04 with kernel version 4.4.0-57. We used FastMPJ 1.0_7 with the device *ibvdev* to run the benchmarks on InfiniBand hardware and MVAPICH2 version 2.3. Optimal configuration values were determined with several test runs.

Due to space constraints, we can only present the most relevant aspects of the extensive evaluation. Further results, e.g. results of all window sizes of the MPI libraries, can be found in our report [33]. Each benchmark was executed three times and the results are displayed using error bars. All throughput benchmarks send 100 million messages and all latency benchmarks 10 million messages. The total number of messages is incrementally halved starting with 4 kb message size to avoid unnecessary long running benchmark runs. In all figures, the payload throughput (GB/s) is displayed as continuous lines and the message throughput in million messages per second as dotted lines. MVAPICH2 and FastMPJ do not support implicit message aggregation like DXNet does. Thus, the benchmarks were executed with *window sizes (WS)* of 1 to 64 with exponential growth to determine the optimal WS to compare to DXNet. Experiments determining latency measure full round-trip-times (RTT). Since FastMPJ does not support multi-threading in a single process, all benchmarks were executed single threaded. DXNet's "single-threaded" results are executed with one application thread and one message handler. A pure single-threaded mode executing both sending and receiving is not possible with DXNet due to its event-driven architecture. To simplify matters, we still refer to this mode as "single threaded". For reference, we used the well known perf tools *ib_send_bw* and *ib_send_lat* to determine a common baseline for each experiment (where possible).

MVAPICH2 supports multi-threading in a single process but the implementation uses a single global lock causing high fluctuations and degrades performance significantly. Figure 3 depicts the results of the bi-directional benchmark with separate send and receive threads. Even with this most basic configuration, the results are not usable for a meaningful comparison to DXNet. Thus, we omitted further multi-threading benchmarks for this evaluation.

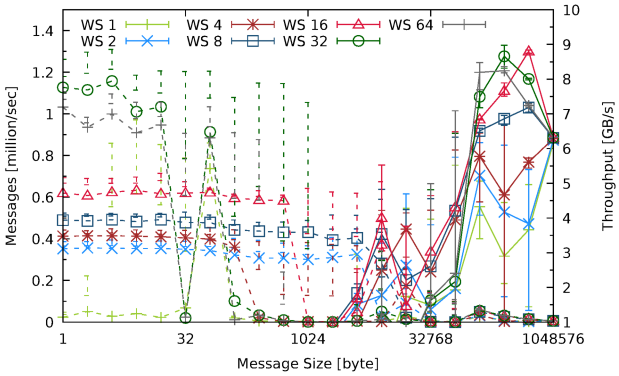

Fig. 3. MVAPICH2: Two nodes, bi-directional throughput and message rate, multi-threaded with one send and one recv thread with increasing message and window size (WS)

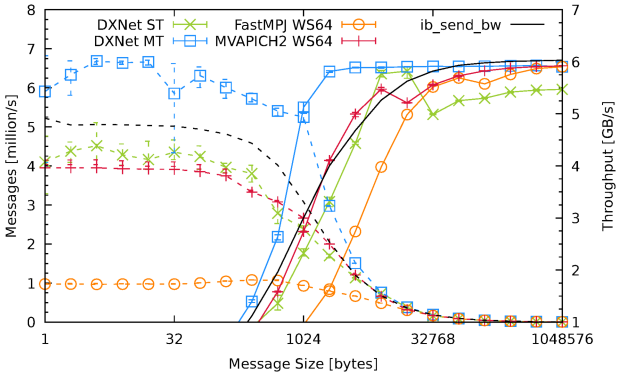

Fig. 4. Two nodes, uni-directional throughput and message rate with increasing message size

Uni-directional. Figure 4 depicts the results of the uni-directional benchmark with increasing message size. DXNet in single-threaded mode achieves a throughput of 4.0 to 4.5 mmps on messages up to 64 byte outperforming FastMPJ with 1.0 mmps and at least being on par with MVAPICH2 with 4.0 mmps. On 512 byte to 4 kb messages, DXNet ST increases its throughput but stays inferior to MVAPICH by up to 0.7 GB/s. For larger messages (32 kb to 1 MB), one message handler is not sufficient to de-serialize and dispatch all incoming messages fast enough. A noticeable drop in performance to 5.4 GB/s at 32 kb message size can be avoided when using at least two message handlers. Using 16 application threads and 4 message handlers (DXNet MT), DXNet can further increase its performance on messages up to 512 byte to up to 6.7 mmps and saturate throughput at 2 kb message size with a stable peak of 5.9 GB/s up to 1 MB messages. DXNet even outperforms the *ib_send_bw* baseline on all message sizes up to 32 kb. Fluctuations on small message sizes can be observed with DXNet on ST benchmark runs and increase on MT benchmark runs. **With**

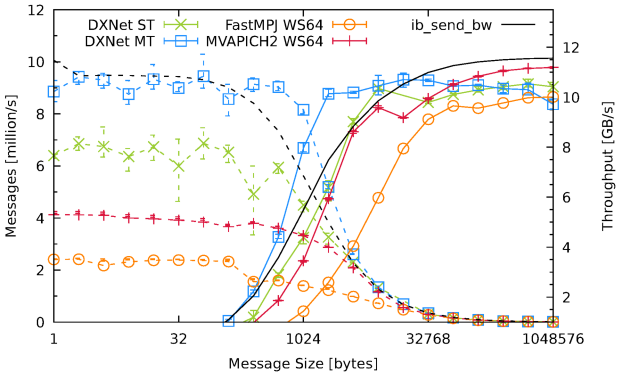

Fig. 5. Two nodes, bi-directional aggregated throughput and message rate with increasing message size

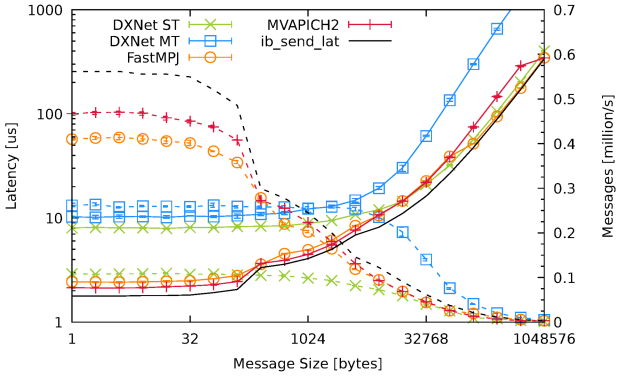

Fig. 6. Two nodes, uni-directional average latency and message rate with increasing message size

multiple application threads sending messages over one connection, DXNet benefits from implicit aggregation by parallel serialization of messages into the ORB (see §III). This results in large chunks of aggregated data to be handled by the underlying transport.

Bi-directional. Figure 5 depicts the results of the bi-directional benchmark with increasing message size. DXNet ST outperforms both FastMPJ and MVAPICH2 on messages up to 512 byte with 6.0 to 6.9 mmps compared to 2.4 mmps and 4.7 mmps. DXNet's ST throughput peaks at 10.4 GB/s. By increasing the number of application threads to 16 and messages handlers to 4, DXNet can further increase performance on all message sizes. For small messages up to 512 byte, DXNet reaches a message rate of 8.6 to 10.2 mmps which is very close to the baseline performance of *ib_send_bw* and at least twice the throughput of MVAPICH2 and 3.5 times the throughput of FastMPJ. Throughput on medium sized messages up to 16 kb also outperforms the *ib_send_bw* baseline by up to a factor of 0.2. However, DXNet's performance decreases from a peak throughput of 11.1 GB/s to 10.4 GB/s for message sizes larger than 32 kb which are not the message sizes DXNet is highly optimized. Nevertheless, this requires further analysis as the cause might impact the performance on next generation hardware.

Uni-directional latency. Figure 6 depicts the results of the uni-directional latency benchmark. For up to 512 byte messages, FastMPJ reaches an average RTT of 2.4 to 4.5 µs. MVAPICH2's is slightly superior with 2.1 to 3.9 µs. Both systems are close to the baseline of *ib_send_lat* with 1.8 to 3.6 µs. However, DXNet's RTT for up to 512 byte message is 7.8 to 8.3 µs which is up to four times the RTT of MVAPICH2. The 7.8 µs spent on the processing pipeline of DXNet (full breakdown, see [12]) can be broken down into: DXNet core in Java (de-/serialization, message object creation, dispatching) 3.5 µs, hardware 2.0 µs, 2.3 µs for Ibdxnet (native C implementation) including JNI context switching. **DXNet trades some latency for transparent object de/-serialization, asynchronous dispatching for scalable multi-threaded support and event-based dispatching.**

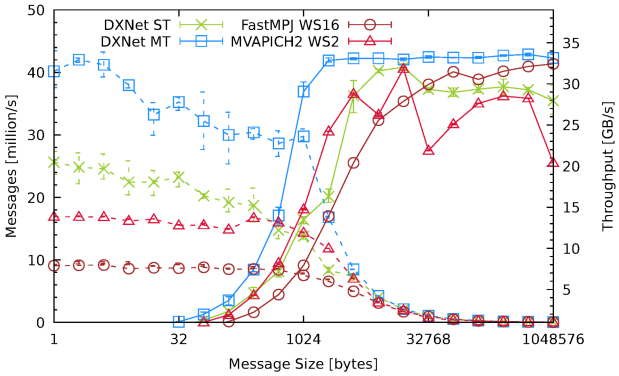

Fig. 7. 8 nodes, all-to-all aggregated send throughput and message rate with increasing message size

All-to-all up to 8 nodes. Figure 7 shows the aggregated throughput of the all-to-all benchmark with up to 8 nodes. For messages up to 64 byte, DXNet ST outperforms both FastMPJ and MVAPICH2 with 18.8 to 25.6 mmps compared to 9.5 mmps and 16.5 to 17.8 mmps. Again, DXNet's fluctuations on small messages can be observed like on the previous benchmarks. On 512 byte to 4 kb messages, DXNet ST is slightly inferior to MVAPICH2 with a drop in performance by approx. 10 GB/s at 2 kb. DXNet ST reaches a peak performance of 32.4 GB/s at 16 kb. Again, for larger messages, one message handler is not sufficient to keep the performance up. But, DXNet MT can further increase performance to 33.2 - 43.4 mmps for 64 byte messages, outperform MVAPICH2 on 512 byte to 4 kb messages by up to two times and establish a stable peak at 33.6 GB/s with only 2 kb message size. **DXNet can handle multiple connections under high load as well as small messages efficiently.**

VII. Yahoo! Cloud Serving Benchmark

We used the Yahoo! Cloud Serving Benchmark (YCSB) [14] to compare the in-memory key-value storage systems **DXRAM** [11] and **RAMCloud** [34]. DXRAM uses **DXNet as its network subsystem and the Ibdxnet transport** with the msgrc engine to enable communication over InfiniBand.

RAMCloud implements a custom network subsystem based on ibverbs with different transport types (*basic*, *tcp*, *infrc*). For the evaluation, we compared to RAMCloud with the *infrc* transport which uses RC QPs.

We used up to 40 servers of our university's cluster, each equipped with two Intel Xeon E5-2697v2 (2.7 GHz) 12 core CPUs, 128 GB RAM and a Mellanox ConnectX-3 HCA. All nodes are connected with 56 Gbit/s Infiniband using a fat-tree hierarchy with two switch levels. The nodes run CentOS 7.4 with the Linux Kernel version 3.10.0-693.

To evaluate the pure network performance of both systems, we designed custom workloads that challenge the network subsystems of the storages. RAMCloud and DXRAM store all objects in-memory and we deactivated any backup/logging on both systems. RAMCloud storage instances are using the same amount of worker threads as CPU cores available. This also applies to DXRAM's message handlers. Object IDs are cached locally after an initial lookup on both DXRAM and RAMCloud. RAMCloud's *read* RPC to get remote objects is very simple and mainly relies on the performance of the network subsystem [34]. The control and data flow of DXRAM's *get* call is similar to RAMCloud's. This ensures a fair comparison because the storage related components do not add any significant overhead on both systems.

On all benchmarks, one half of the nodes are used as storage nodes and the other half as benchmark nodes. All benchmarks are executed with uniform distribution to distribute the benchmark load across all storage nodes resulting in all benchmark nodes communication with all storage nodes. All workloads are 100% read operations to avoid possible overhead introduced by the storage system on update/write operations. Hence, the network of the storage is the dominating factor for performance. Each benchmark node is running 100 application threads emulating interactive users. Every storage node stores 10,000,000 keys and every benchmark node executes 100,000,000 operations. These common attributes apply to the three workloads with the following object sizes:

1) Workload A: 10x 100-byte objects per key, YCSB reference workload (e.g. session store) [14]
2) Workload B: 1x 32 byte object per key, Facebook social graph typical object size [32]

Workload A. The results of Workload A are depicted in Figure 8 on the left. The results for RAMCloud with 20 benchmark nodes and 20 storage nodes are not available due to RAMCloud clients crashing during the benchmark phase with a network error. We could not solve this with several retries and different system/benchmark configurations. DXRAM outperforms RAMCloud with increasing node count and scales well with up to 20 benchmark nodes. With each of the 20 benchmark nodes issuing requests to each of the 20 storage nodes, DXRAM achieves 2.95 mops which is over twice the amount of RAMCloud with 1.34 mops. Here, DXRAM benefits from DXNet's capability of handling many simultaneous connections efficiently.

Workload B. The results of Workload B, depicted in Figure 8 on the right, show that DXRAM performs well on

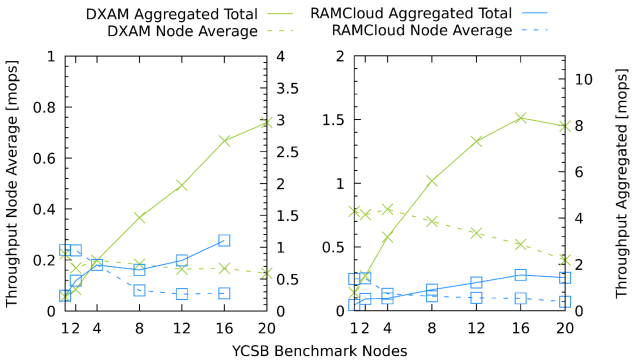

Fig. 8. Workload A on the left and Workload B on the right: Average per node (dotted lines) with y-axis on the left and aggregated throughput (continuous lines) with increasing node count with axis on the right of each plot

read requests with small object sizes which are typical for graph-based applications [13], [32], [42]. DXRAM achieves a throughput of 7.96 mops with 20 benchmark nodes which is more than five times the performance of RAMCloud with 1.42 mops. DXRAM highly benefits from DXNet's careful optimizations regarding small messages. However, both systems experience performance loss with 20 nodes (20 storage and 20 benchmark) compared to 16 nodes which requires further analysis.

VIII. Conclusions

We presented Ibdxnet, a transport for the Java messaging library DXNet which allows multi-threaded Java applications to benefit from low latency and high throughput using InfiniBand hardware. DXNet provides transparent connection management, concurrency handling, message serialization and allows applications to switch from Ethernet to InfiniBand hardware transparently. Ibdxnet's native subsystem provides dynamic, scalable, concurrent and automatic connection management and the msgrc messaging engine implementation. The msgrc engine uses a dedicated send and receive thread to drive RC QPs asynchronously which ensures scalability with many nodes. SGEs are used to simplify buffer handling and increase buffer utilization when sending data provided by the higher-level DXNet core. A carefully crafted architecture minimizes context switching between Java and the native space as well us exchanging data using shared memory buffers. The evaluation shows that DXNet with the Ibdxnet transport can keep up with FastMPJ and MVAPICH2 on single-threaded applications on all message sizes and even outperform them in multi-threaded high load applications. Especially on small message sizes up to 64 byte, DXNet with Ibdxnet outperforms both systems by up to twofold with up to 43.4 mmps on 8 nodes all-to-all communication. Furthermore, we showed that DXRAM, a Java-based key-value storage using DXNet as the network subsystem, scales well with up to 40 nodes on the YCSB benchmark. DXRAM outperforms RAMCloud, implemented in C++, on a standard workload twofold and on a workload targeting graph-based applications even fivefold with 7.96 mops on a total of 40 nodes.

REFERENCES

[1] Infinispan. http://infinispan.org/.

[2] libvma github wiki. https://github.com/Mellanox/libvma/wiki/Architecture.

[3] Mvapich website. http://mvapich.cse.ohio-state.edu/. Accessed: 2018-07-12.

[4] Open mpi. https://www.open-mpi.org/.

[5] Ramcloud source code. https://github.com/PlatformLab/RAMCloud.

[6] rdma_cm - linux man page. https://linux.die.net/man/7/rdma_cm. Accessed: 2018-07-12.

[7] Rdmamojo - blog on rdma technology and programming by dotan barak. https://www.rdmamojo.com.

[8] M. Anderson, S. Smith, N. Sundaram, M. Capotă, Z. Zhao, S. Dulloor, N. Satish, and T. L. Willke. Bridging the gap between hpc and big data frameworks. *Proc. VLDB Endow.*, 10(8):901–912, Apr. 2017.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.

[10] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and X. Li. mpijava: A java interface to mpi. http://www.hpjava.org/mpiJava.html, July 2002.

[11] K. Beineke, S. Nothaas, and M. Schoettner. High throughput log-based replication for many small in-memory objects. In *IEEE 22nd International Conference on Parallel and Distributed Systems*, pages 535–544, 2016.

[12] K. Beineke, S. Nothaas, and M. Schöttner. Efficient messaging for java applications running in data centers. In *International Workshop on Advances in High-Performance Algorithms Middleware and Applications (in proceedings of CCGrid18)*, 2018.

[13] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8:1804–1815, Aug. 2015.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[15] M. Dawson, G. Johnson, and A. Low. Best practices for using the java native interface. https://www.ibm.com/developerworks/library/j-jni/index.html#using. Accessed: 2018-07-08.

[16] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51:107–113, Jan. 2008.

[17] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Incremental page rank computation on evolving graphs. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 1094–1095, 2005.

[18] S. Ekanayake, S. Kamburugamuve, and G. C. Fox. Spidal java: High performance data analytics with java and mpi on large multicore hpc clusters. pages 3:1–3:8, 2016.

[19] B. Evans. An architect's view of hazelcast. https://www.trivadis.com/sites/default/files/downloads/an_architects_view_of_hz.pdf, 2015.

[20] R. R. Exposito, S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. Fastmpj: a scalable and efficient java message-passing library. *Cluster Computing*, 17:1031–1050, Sept. 2014.

[21] R. R. Expósito, G. L. Taboada, J. Touriño, and R. Doallo. Design of scalable java message-passing communications over infiniband. *The Journal of Supercomputing*, pages 141–165, July 2012.

[22] D. Goldenberg, T. Dar, and G. Shainer. Architecture and implementation of sockets direct protocol in windows. *2006 IEEE International Conference on Cluster Computing*, pages 1–9, 2006.

[23] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47:98 – 115, 2015.

[24] V. Kashyap. Ip over infiniband (ipoib) architecture. https://www.ietf.org/rfc/rfc4392.txt, April 2006.

[25] J. Kreps, N. Narkhede, and J. Rao. Kafka: a distributed messaging system for log processing. In *NetDB 2011: 6th Workshop on Networking meetsDatabases*, 2011.

[26] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[27] X. Liu. Entity centric information retrieval. *SIGIR Forum*, 50:92–92, June 2016.

[28] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu. Datampi: Extending mpi to hadoop-like big data computing. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 829–838, May 2014.

[29] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: The java unsafe api in the wild. *SIGPLAN Not.*, 50:695–710, Oct. 2015.

[30] S. Mehta and V. Mehta. Hadoop ecosystem: An introduction. In *International Journal of Science and Research (IJSR)*, volume 5, June 2016.

[31] S. Narravula, A. Mamidala, A. Vishnu, G. Santhanaraman, and D. K. Panda. High performance mpi over iwarp: Early experiences. In *2007 International Conference on Parallel Processing (ICPP 2007)*, pages 46–46, Sept. 2007.

[32] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[33] S. Nothaas, K. Beineke, and M. Schoettner. Ibdxnet: Leveraging infiniband in highly concurrent java applications. https://arxiv.org/abs/1812.01963, 2018.

[34] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[35] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, November 1999. Previous number = SIDL-WP-1999-0120.

[36] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. sson, D. van der Spoel, B. Hess, and E. Lindahl. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29:845–854, 2013.

[37] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita. Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf. *Procedia Computer Science*, 53:121–130, 2015.

[38] A. Shafi, B. Carpenter, and M. Baker. Nested parallelism for multi-core hpc systems using java. In *Journal of Parallel and Distributed Computing*, pages 532–545, 2009.

[39] P. Stuedi, B. Metzler, and A. Trivedi. jverbs: Ultra-low latency for data center applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 10:1–10:14, New York, NY, USA, 2013. ACM.

[40] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 1–15, New York, NY, USA, 2017. ACM.

[41] S. Thirugnanapandi, S. Kodali, N. Richards, T. Ellison, X. Meng, and I. Poddar. Transparent network acceleration for java-based workloads in the cloud. https://www.ibm.com/developerworks/library/j-transparentaccel/, January 2014.

[42] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *CoRR*, abs/1111.4503, 2011.

[43] S. Wang. Graph analytics and machine learning. https://de.slideshare.net/stanleywanguni/graph-analytic-and-machine-learning, 2016.

[44] X. Wu, X. Zhu, G. Q. Wu, and W. Ding. Data mining with big data. *IEEE Transactions on Knowledge and Data Engineering*, 26:97–107, Jan. 2014.

[45] H. Zhang, W. Huang, J. Han, J. He, and L. Zhang. A performance study of java communication stacks over infiniband and giga-bit ethernet. In *in Proceedings of the IFIP International Conference on Network and Parallel Computing Workshops, 2007. (NPC Workshops), 2007*, pages 602–607, 2010.

[46] P. Zhao, Y. Li, H. Xie, Z. Wu, Y. Xu, and J. C. Lui. Measuring and maximizing influence via random walk in social activity networks. pages 323–338, Mar. 2017.

Ibdxnet: Leveraging InfiniBand in Highly Concurrent Java Applications

Stefan Nothaas
stefan.nothaas@hhu.de

Kevin Beineke
kevin.beineke@hhu.de

Michael Schöttner
michael.schoettner@hhu.de

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf
Universitätsstr. 1, 40225 Düsseldorf, Germany

Abstract

Todays big data applications generate hundreds or even thousands of terabytes of data. Commonly, Java based applications are used for further analysis. A single commodity machine, for example in a data center or typical cloud environment, cannot store and process the vast amounts of data making distribution mandatory. Thus, the machines have to use interconnects to exchange data or coordinate data analysis. However, commodity interconnects used in such environments, e.g. Gigabit Ethernet, cannot provide high throughput and low latency compared to alternatives like InfiniBand to speed up data analysis of the target applications. In this report, we describe the design and implementation of Ibdxnet, a low-latency and high-throughput transport providing the benefits of InfiniBand networks to Java applications. Ibdxnet is part of the Java-based DXNet library, a highly concurrent and simple to use messaging stack with transparent serialization of messaging objects and focus on very small messages (< 64 bytes). Ibdxnet implements the transport interface of DXNet in Java and a custom C++ library in native space using JNI. Several optimizations in both spaces minimize context switching overhead between Java and C++ and are not burdening message latency or throughput. Communication is implemented using the messaging verbs of the ibverbs library complemented by an automatic connection management in the native library. We compared DXNet with the Ibdxnet transport to the MPI implementations FastMPJ and MVAPICH2. For small messages up to 64 bytes using multiple threads, DXNet with the Ibdxnet transport achieves a bi-directional message rate of 10 million messages per second and surpasses FastMPJ by a factor of 4 and MVAPICH by a factor of 2. Furthermore, DXNet scales well on a high load all-to-all communication with up to 8 nodes achieving a total aggregated message rate of 43.4 million messages per second for small messages and a throughput saturation of 33.6 GB/s with only 2 kb message size.

1 Introduction

Interactive applications, especially on the web [6, 28], simulations [34] or online data analysis [14, 41, 43] have to process terabytes of data often consisting of small objects. For example, social networks are storing graphs with trillions of edges resulting in a per object size of less than 64 bytes for the majority of objects [10]. Other graph examples are brain simulations with billions of neurons and thousands of connections each [31] or search engines for billions of indexed web pages [20]. To provide high interactivity to the user, low latency is a must in many of these application domains. Furthermore, it is also important in the domain of mobile networks moving state management into the cloud [23].

Big data applications are processing vast amounts of data which require either an expensive supercomputer or distributed platforms, like clusters or cloud environments [21]. High performance interconnects, such as InfiniBand, are playing a key role to keep processing and response times low, especially for highly interactive and always online applications. Today, many cloud providers, e.g. Microsoft, Amazon or Google, offer instances equipped with InfiniBand.

InfiniBand offers messaging verbs and RDMA, both providing one way single digit microsecond latencies. It depends on the application requirements whether messaging verbs or RDMA is the better choice to ensure optimal performance [38].

In this report, we focus on Java-based parallel and distributed applications, especially big data applications, which commonly communicate with remote nodes using asynchronous and synchronous messages [10, 16, 13, 42]. Unfortunately, accessing InfiniBand verbs from Java is not a built-in feature of the commonly used JVMs. There are several external libraries, wrappers or JVMs with built-in support available but all trade performance for transparency or require proprietary environments (§3.1). To use InfiniBand from Java, one can rely on available (Java) MPI implementations. But, these are not providing features such as serialization for messaging objects and no automatic connection

management (§3.2).

We developed the network subsystem DXNet (§2) which provides transparent and simple to use sending and event based receiving of synchronous and asynchronous messages with transparent serialization of messaging objects [8]. It is optimized for high concurrency on all operations by implementing lock-free synchronization. DXNet is implemented in Java and open source and available at Github [1].

In this report, we propose Ibdxnet, a transport for the DXNet network subsystem. The transport uses reliable messaging verbs to implement InfiniBand support for DXNet and provides low latency and high throughput messaging for Java.

Ibdxnet implements scalable and automatic connection and queue pair management, the *msgrc* transport engine, which uses InfiniBand messaging verbs, and a JNI interface. We present best practices applied to ensure scalability across multiple threads and nodes when working with InfiniBand verbs by elaborating on the implementation details of Ibdxnet. We carefully designing an efficient and low latency JNI layer to connect the native Ibdxnet subsystem to the Java based IB transport in DXNet. The IB transport uses the JNI layer to interface with Ibdxnet, extends DXNet's outgoing ring buffer for InfiniBand usage and implements scalable scheduling of outgoing data for many simultaneous connections. We evaluated DXNet with the IB transport and Ibdxnet, and compared then to two MPI implementations supporting InfiniBand: the well known MVAPICH2 and the Java based FastMPJ implementations.

Though, MPI is discussed in related work (§3.2) and two implementations are evaluated and compared to DXNet (§9), DXNet, the IB transport nor Ibdxnet are implementing the MPI standard. The term *messaging* is used by DXNet to simply refer to exchanging data in the form of messages (i.e. additional metadata identifies message on receive). DXNet does not implement any by the standard defined MPI primitives. Various low-level libraries to use InfiniBand in Java are not compared in this report, but in a separate one.

The report is structured in the following way: In Section 2, we present a summary of DXNet and its aspects important to this report. In Section 3, we discuss related work which includes a brief summary of available libraries and middleware for interfacing InfiniBand in Java applications. MPI and selected implementations supporting InfiniBand are presented as available middleware solutions and compared to DXNet. Lastly, we discuss target applications in the field of Big-Data which benefit from InfiniBand usage. Section 4 covers InfiniBand basics which are of concern for this report. Section 5 discusses JNI usage and presents best practices for low latency interfacing with native code from Java using JNI. Section 6 gives a brief overview of DXNet's multi layered stack when using InfiniBand. Implementation details of the native part Ibdxnet are given in Section 7 and the IB transport in Java are presented in Section 8. Section 9 presents and compares the experimental results of MVAPICH2, FastMPJ and DXNet. Conclusions are presented in Section 10.

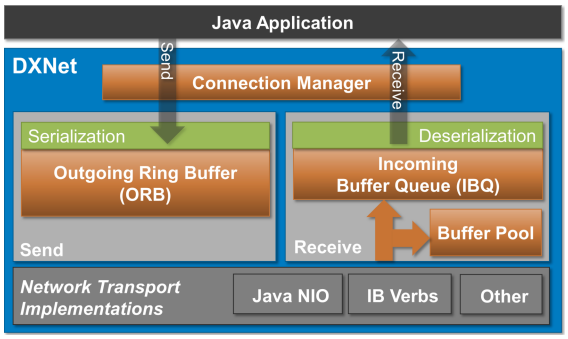

Figure 1: Simplified DXNet Architecture

2 DXNet

DXNet is a network library for Java targeting, but not limited to, highly concurrent big data applications. DXNet implements an **asynchronous event driven messaging** approach with a simple and easy to use application interface. **Messaging** describes **transparent sending and receiving of complex (even nested) data structures** with implicit serialization and de-serialzation. Furthermore, DXNet provides a built in primitive for transparent **request-response communication**.

DXNet is optimized for highly multi-threaded sending and receiving of small messages by using **lock-free data structures, fast concurrent serialization, zero copy and zero allocation**. The core of DXNet provides **automatic connection and buffer management**, serialization of message objects and an interface for implementing different transports. Currently, an Ethernet transport using Java NIO sockets and an InifiniBand transport using *ibverbs* (§7) is implemented.

The following subsections describe the most important aspects of DXNet and its core which are depicted in Figure 1 and relevant for further sections of this report. A more detailed insight is given in a dedicated paper [8]. The source code is available at Github [1].

2.1 Automatic Connection Management

To relieve the programmer from explicit connection creation, handling and cleanup, DXNet implements automatic and transparent connection creation, handling and cleanup. Nodes are addressed using an **abstract and unique 16-bit nodeID**. Address mappings must be registered to allow associating the nodeIDs of each remote node with a corresponding implementation dependent endpoint (e.g. socket, queue pair). To provide scalability with up to hundreds of simultaneous connections, our event driven system does not create one thread per connection. A **new connection is cre-**

ated automatically once the first message is either sent to a destination or received from one. Connections are closed once a configurable connection limit is reached using a recently used strategy. Faulty connections (e.g. remote node not reachable anymore) are handled and cleaned up by the manager. Error handling on connection errors or timeouts is propagated to the application using exceptions.

2.2 Sending of Messages

Messages are serialized Java objects and sent **asynchronously** without waiting for a completion. A message can be targeted towards one or multiple receivers. Using the message type **Request**, it is sent to one receiver, only. When sending a request, the sender waits until **receiving a corresponding response** message (transparently handled by DXNet) or skips waiting and collects the response later.

We expect applications calling DXNet concurrently with **multiple threads** to send messages. Every message is automatically and concurrently serialized into the **Outgoing Ring Buffer (ORB)**, a natively allocated and lock-free ring buffer. **Messages are automatically aggregated** which increases send throughput. The ORB, one per connection, is allocated in native memory to allow **direct and zero-copy access** by the low-level transport. A transport runs a decoupled dedicated thread which removes the serialized and ready to send data from the ORB and forwards it to the hardware.

2.3 Receiving of Messages

The network transport handles incoming data by writing it to **pooled native buffers** to avoid burdening the Java garbage collection. Depending on how a transport writes and reads data, the buffers might contain fully serialized messages or just fragments. Every received buffer is pushed to the ring buffer based **Incoming Buffer Queue (IBQ)**. Both, the buffer pool as well as the IBQ are shared among all connections. **Dedicated handler threads** pull buffers from the IBQ and process them asynchronously by de-serializing them and creating Java message objects. The messages are passed to **pre-registered callback methods** of the application.

2.4 Flow Control

DXNet implements its own **flow control (FC)** mechanism to avoid flooding a remote node with many (very small) messages. This would result in an increased overall latency and lower throughput if the receiving node cannot keep up with processing incoming messages. On sending a message, the per connection dedicated FC checks if a configurable threshold is exceeded. This threshold describes the **number of bytes sent by the current node but not fully processed by the receiving node**. Once the configurable threshold is exceeded, the receiving node slices the number of bytes received into equally sized windows (window size configurable) and sends the number of windows confirmed back to the source node. Once the sender receives this confirmation, the number of bytes sent but not processed is **reduced by the number of received windows multiplied with the configured window size**. If an application send thread was previously blocked due to exceeding this threshold, it can now continue with processing.

2.5 Transport Interface

DXNet provides a transport interface allowing implementations of different transport types. On initialization of DXNet, one of the implemented transports can be selected. Afterwards when using DXNet, the transport is transparent for the application. The following tasks must be handled by every transport implementation:

- Connection: Create, close and cleanup
- Get ready to send data from ORB and send it (ORB triggers callback once data is available)
- Handle received data by pushing it to the IBQ
- Manage flow control when sending/receiving data

Every other task that is not exposed directly by one of the following methods must be handled internally by the transport. The core of DXNet relies on the following methods of abstract Java classes/interfaces which must be implemented by every transport:

- Connection: open, close, dataPosted
- ConnectionManager: createConnection, closeConnection
- FlowControl: sendFlowControlData, getAndResetFlowControlData

We elaborate on further details about the transport interface in Section 8 where we describe the transport implementation for Ibdxnet.

3 Related Work

Related work discusses different topics which are of interest to DXNet with the IB transport and Ibdxnet. First, we present a summary of our evaluation results of existing solutions to use InfiniBand in Java applications (§3.1). These results were important before developing Ibdxnet. Next, we compare DXNet and the MPI standard (§3.2) followed by MPI implementations supporting InfiniBand (§3.3) and UPX (§3.4). To our knowledge, this concludes the list of available middleware offering higher level networking primitives comparable to DXNet's. In the last Subsection 3.5, we discuss big-data systems and applications supporting InfiniBand for target applications of interest to DXNet.

3.1 Java and InfiniBand

Before developing Ibdxnet and the InfiniBand transport for DXNet, we evaluated available (low-level) solutions for leveraging InfiniBand hardware in Java applications. This includes using NIO sockets with **IP over InfiniBand (IPoIB)** [25], **jVerbs** [37], **JSOR** [40], **libvma** [2] and **native c-verbs with ibverbs**. Extensive experiments analyzing throughput and latency of both messaging verbs and RDMA were conducted to determine a suitable candidate for using InfiniBand with Java applications and are published in a separate report.

Summerized, the results show that transparent solutions like IPoIB, libvma or JSOR, which allow existing socket-based applications to send and receive data transparently over InfiniBand hardware, are not able to deliver an overall adequate throughput and latency. For the verbs-based libraries, jVerbs gets close to the native ibverbs performance but, like JSOR, requires a proprietary JVM to run. Overall, none of the analyzed solutions, other than ibverbs, are delivering an adequate performance. Furthermore, we want DXNet to stay independent of the JVM when using InfiniBand hardware. Thus, we decided to use the native ibverbs library with the Java Native Interface to avoid the known performance issues of the evaluated solutions.

3.2 MPI

The message passing interface [19] defines a standard for high level networking primitives to send and receive data between local and remote processes, typically used for HPC applications.

An application can send and receive primitive data types, arrays, derived or vectors of primitive data types, and indexed data types using MPI. The synchronous primitives *MPI_Send* and *MPI_Recv* perform these operations in blocking mode. The asynchronous operations *MPI_Isend* and *MPI_Irecv* allow non blocking communication. A status handle is returned with each started asynchronous operation. This can be used to check the completion of the operation or to actively wait for one or multiple completions using *MPI_Wait* or *MPI_Waitall*. Furthermore, there are various collective primitives which implement more advanced operations such as scatter, gather or reduce.

Sending and receiving of data with MPI requires the application to issue a receive for every send with a target buffer that can hold at least the amount of data sent by the remote. DXNet relieves the application from this responsibility. Application threads can send messages with variable size and-DXNet manages the buffers used for sending and receiving. The application does not have to issue any receive operations and wait for data to arrive actively. Incoming messages are dispatched to pre-registered callback handlers by dedicated handler threads of DXNet.

DXNet supports transparent serialization and de-serialization of complex (even nested) data types (Java objects) for messages. MPI primitives for sending and receiving data require the application to use one of the available data types supported and doesn't offer serialization for more complex datatypes such as objects. However, the MPI implementation can benefit from the lack of serialization by avoiding any copying of data, entirely. Due to the nature of serialization, DXNet has to create a (serialized) "copy" of the message when serializing it into the ORB. Analogously, data is copied when a message is created from incoming data during de-serialization.

Messages in DXNet are sent asynchronously while requests offer active waiting or probing for the corresponding response. These communication patterns can also be applied by applications using MPI. The communication primitives currently provided by DXNet are limited to messages and request-response. Nevertheless, using these two primitives, other MPI primitives, such as scatter, gather or reduce, can be implemented by the application if required.

DXNet does not implement multiple protocols for different buffer sizes like MPI with eager and rendezvous. A transport for DXNet might implement such a protocol but our current implementations for Ethernet and InfiniBand do not. The aggregated data available in the ORB is either sent as a whole or sliced and sent as multiple buffers. The transport on the receiving side passes the stream of buffers to DXNet and puts them into the IBQ. Afterwards, the buffers are reconnected to a stream of data by the MCC before extracting and processing the messages.

An instance using DXNet runs within one process of a Big Data application with one or multiple application threads. Typically, one DXNet instance runs per cluster node. This allows the application to dynamically scale the number of threads up or down within the same DXNet instance as needed. Furthermore, fast communication between multiple threads within the same process is possible, too.

Commonly, an MPI application runs a single thread per process. Multiple processes are spawned according to the number of cores per node with IPC fully based on MPI. MPI does offer different thread modes which includes issuing MPI calls using different threads in a process. Typically, this mode is used in combination with OpenMP [4]. However, it is not supported by all MPI implementations which also offer InfiniBand support (§3.3). Furthermore, DXNet supports dynamic up and down scaling of instances. MPI implementations support up-scaling (for non singletons) but down scaling is considered an issue for many implementations. Processes cannot be removed entirely and might cause other processes to get stuck or crash.

Connection management and identifying remote nodes are similar with DXNet and MPI. However, DXNet does not come with deployment tools such as *mpirun* which assigns the ids/ranks to identify the instances. This intentional design decision allows existing applications to integrate DXNet

without restrictions to the bootstrapping process of the application. Furthermore, DXNet supports dynamically adding and removing instances. With MPI, an application must be created by using the MPI environment. MPI applications must be run using a special coordinator such as *mpirun*. If executed without a communicator, an MPI world is limited to the current process it is created in which doesn't allow communication with any other instances. Separate MPI worlds can be connected but the implementation must support this feature. To our knowledge, there is no implementation (with InfiniBand support) that currently supports this.

3.3 MPI Implementations Supporting InfiniBand

This section only considers MPI implementations supporting InfiniBand directly. Naturally, IPoIB can be used to run any MPI implementation supporting Ethernet networks over InfiniBand. But, as previously discussed (§3.1), the network performance is very limited when using IPoIB.

MVAPICH2 is a MPI library [32] supporting various network interconnects, such as Ethernet, iWARP, Omni-Path, RoCE and InfiniBand. MVAPICH2 includes features like RDMA fast path or RDMA operations for small message transfers and is widely used on many clusters over the world. **Open MPI** [3] is an open source implementation of the MPI standard (currently full 3.1 conformance) supporting a variety of interconnects, such as Ethernet using TCP sockets, RoCE, iWARP and InfiniBand.

mpiJava [7] implements the MPI standard by a collection of wrapper classes that call native MPI implementations, such as MVAPICH2 or OpenMPI, through JNI. The wrapper based approach provides efficient communication relying on native libraries. However, it is not threadsafe and, thus, is not able to take advantage of multi-core systems using multithreading.

FastMPJ [17] uses Java Fast Sockets [39] and ibvdev to provide a MPI implementation for parallel systems using Java. Initially, **ibvdev** [18] was implemented as a low-level communication device for **MPJ Express** [35], a Java MPI implementation of the mpiJava 1.2 API specification. ibvdev implements InfiniBand support using the low-level verbs API and can be integrated into any parallel and distributed Java application. FastMPJ optimizes MPJ Express collective primitives and provides efficient non-blocking communication. Currently, FastMPJ supports issuing MPI calls using a single thread, only.

3.4 Other Middleware

UCX [36] is a network stack designed for next generation systems for applications with an highly multi-threaded environment. It provides three independent layers: UCS is a service layer with different cross platform utilities, such as atomic operations, thread safety, memory management and data structures. The transport layer UCT abstracts different hardware architectures and their low-level APIs, and provides an API to implement communication primitives. UCP implements high level protocols such as MPI or PGAS programming models by using UCT.

UCX aims to be a common computing platform for multi-threaded applications. However, DXNet does not and, thus, does not include its own atomic operations, thread safety or memory management for data structures. Instead, it relies on the multi-threading utilities provided by the Java environment. DXNet does abstract different hardware like UCX but only network interconnects and not GPUs or other co-processors. Furthermore, DXNet is a simple networking library for Java applications and does not implement MPI or PGAS models. Instead, it provides simple asynchronous messaging and synchronous request-response communication, only.

3.5 Target Applications using InfiniBand

Providing high throughput and low latency, InfiniBand is a technology which is widely used in various big-data applications.

Apache Hadoop [22] is a well known Java big-data processing framework for large scale data processing using the MapReduce programming model. It uses the Hadoop Distributed File System for storing and accessing application data which supports InfiniBand interconnects using RDMA. Also implemented in Java, **Apache Spark** is a framework for big-data processing offering the domain-specific-language Spark SQL, a stream processing and machine learning extension and the graph processing framework GraphX. It supports InfiniBand hardware using an additional RDMA plugin [5].

Numerous key-value storages for big-data applications have been proposed that use InfiniBand and RDMA to provide low latency data access for highly interactive applications.

RAMCloud [33] is a distributed key-value storage optimized for low latency data access using InfiniBand with messaging verbs. Multiple transports are implemented for network communication, e.g. using reliable and unreliable connections with InfiniBand and Ethernet with unreliable connections. **FaRM** [15] implements a key-value and graph storage using a shared memory architecture with RDMA. It performs well with a throughput of 167 million key-value lookups and 31 us latency using 20 machines. **Pilaf** [30] also implements a key-value storage using RDMA for get operations and messaging verbs for put operations. **MICA** [27] implements a key-value storage with a focus on NUMA architectures. It maps each CPU core to a partition of data and communicates using a request-response approach using unreliable connections. **HERD** [24] borrows the design of

MICA and implements networking using RDMA writes for the request to the server and messaging verbs for the response back to the client.

4 InfiniBand and ibverbs Basics

This section covers the most important aspects of the InfiniBand hardware and the native ibverbs library which are relevant for this report. Abbreviations introduced here (most of them commonly used in the InfiniBand context) are used throughout the report from this point on.

The **host channel adapter (HCA)** connected to the PCI bus of the host system is the network device for communicating with other nodes. The offloading engine of the HCA processes outgoing and incoming data asynchronously and is connected to other nodes using copper or optical cables via one or multiple switches. The **ibverbs** API provides the interface to communicate with the HCA either by exchanging data using Remote Direct Memory Access (RDMA) or messaging verbs.

A **queue pair (QP)** identifies a physical connection to a remote node when using **reliable connected (RC)** communication. Using non connected **unreliable datagram (UD)** communication, a single QP is sufficient to send data to multiple remotes. A QP consists of one **send queue (SQ)** and one **receive queue (RQ)**. On RC communication, a QP's SQ and RQ are always cross connected with a target's QP, e.g. node 0 SQ connects to node 1 RQ and node 0 RQ to node 1 SQ.

If an application wants to send data, it posts a **work request (WR)** containing a pointer to the buffer to send and the length to the SQ. A corresponding WR must be posted on the RQ of the connected QP on the target node to receive the data. This WR also contains a pointer to a buffer and its size to receive any incoming data to.

Once the data is sent, a **work completion (WC)** is generated and added to a **completion queue (CQ)** associated with the SQ. A WC is also generated for the corresponding WCQ of the remote's RQ receiving the data, once the data arrived. The WC of the send task tells the application that the data was successfully sent to the remote (or provides error information otherwise). On the remote receiving the data, the WC indicates that the buffer attached to the previously posted WR is now filled with the remote's data.

When serving multiple connections, every single SQ and RQ does not need a dedicated CQ. A single CQ can be used as a **shared completion queue (SCQ)** with multiple SQs or RQs. Furthermore, when receiving data from multiple sources, instead of managing many RQs to provide buffers for incoming data, a **shared receive queue (SRQ)** can be used on multiple QPs instead of single RQs.

When attaching a buffer to a WR, it is attached as a **scatter gather element (SGE)** of a **scatter gather list (SGL)**. For sending, the SGL allows the offloading engine to gather the data from many scattered buffers and send it as one WR. For receiving, the received data is scattered to one or multiple buffers by the offloading engine.

5 Low Latency Data Exchange Between Java and C

In this section, we describe our experiences with and best practices for the Java Native Interface (JNI) to avoid performance penalties for latency sensitive applications. These are applied to various implementation aspects of the IB transport which are further explained in their dedicated sections.

Using JNI is mandatory if the Java space has to interface with native code, e.g. for IO operations or when using native libraries. As we decided to use the low-level ibverbs library to benefit from full control, high flexibility and low latency (§3.1), we had to ensure that interfacing with native code from Java does not introduce too much overhead compared to the already existing and evaluated solutions.

The Java Native Interface (JNI) allows Java programmers to call native code from C/C++ libraries. It is a well known method to interface with native libraries that are not available in Java or access IO using system calls or other native libraries. When calling code of a native library, the library has to expose and implement a predefined interface which allows the JVM to connect the native functions to native declared Java methods in a Java class. With every call from Java to the native space and vice versa, a context switch is required to be executed by the JVM environment. This involves tasks related to thread and cache management adding latency to every native call. This increases the duration of such a call and is crucial, especially regarding the low latency of IB.

Exchanging data with a native library without adding considerable overhead is challenging. For single primitive values, passing parameters to functions is convenient and does not add any considerable overhead. However, access to Java classes or arrays from native space requires synchronization with the JVM (and its garbage collector) which is very expensive and must be avoided. Alternatively, one can use ByteBuffers allocated as DirectByteBuffers which allocates memory in native memory. Java can access the memory through the ByteBuffer and the native library can get the native address of the array and the size with the functions `GetDirectBufferAddress` and `GetDirectBufferCapacity`. However, these two calls increase the latency by tenth to even hundreds of microseconds (with high variation).

This problem can be solved by **allocating a native buffer in the native space, passing its address and size** to the Java space and **access it using the Unsafe API** or wrap it as a newly allocated (Direct) ByteBuffer. The latter requires reflection to access the constructor of the DirectByteBuffer and

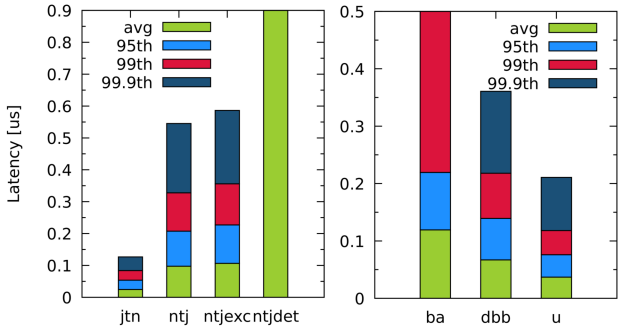

Figure 2: Microbenchmarks to evaluate JNI call overhead and data exchange overhead using different types of memory access

set the address and size fields. We decided to use the Unsafe API because we map native structs and don't require any of the additional features the ByteBuffer provides. The native address is cached which allows fast exchange of data from Java to native and vice versa. To improve convenience when accessing fields of a data structure, a helper class with getter and setter wrapper methods is created to access the fields of the native struct.

We evaluated different means of passing data from Java to native and vice versa as well as the function/method call overhead. Figure 2 shows the results of the microbenchmarks used to evaluate JNI call overhead as well as overhead of different memory access methods. The results displayed are the averages of three runs of each benchmark executing the operation 100,000,000 times. A warm-up of 1,000 operations preceeds each benchmark run. For JNI context switching, we measured the latency introduced of Java to native (jtn), native to Java (ntj), native to Java with exception checking (ntjexc) and native to Java with thread detaching (ntjdet). For exchanging data between Java and native, we measured the latency introduced by accessing a 64 byte buffer in both spaces for a primitive Java byte array (ba), Java DirectByteBuffer (dbb) and Unsafe (u). The benchmarks were executed on a machine with Intel Core i7-5820K CPU and Java 1.8 runtime.

The results show that the average single costs for context switching are neglectable with an average switching time of only up to 0.1 µs. We exchange data using primitive function arguments, only. Data structures are mapped and accessed as C-structs in the native space. In Java, we access the native C-structs using a helper class which utilizes the Unsafe library [29] as this is the fastest method in both spaces.

These results influenced the important design decision to **run native threads, attached once as daemon threads to the JVM**, which call to Java instead of Java threads calling native methods (§7.2.3, §7.2.4). Furthermore, we avoid using any of the JNI provided helper functions where possible [26]. For example: attaching a thread to the JVM involves

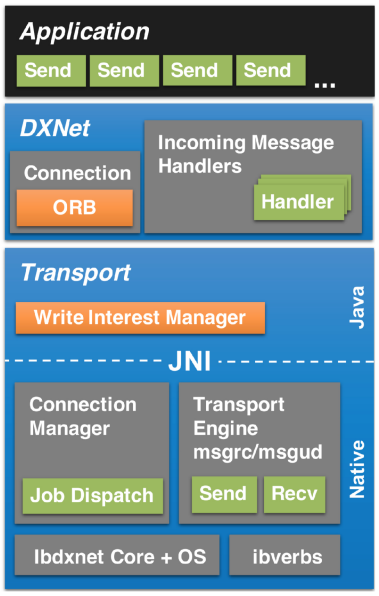

Figure 3: Layered architecture of DXNet with the IB transport and Ibdxnet (using the msgrc engine). Threads are colored green.

expensive operations like creating a new Java thread object and various state changes to the JVM environment. Avoiding them on every context switch is crucial to latency and performance on every call.

Lastly, we minimized the number of calls to the Java space by combining multiple tasks into a single cross-space call instead of yielding multiple calls. For inter space communication, we highly rely on communication via buffers mapped to structs in native space and wrapper classes in Java (see above). This is highly application dependable and not always possible. But if possible and applied, this can improve the overall performance.

We applied this technique of combining multiple tasks into a single cross-space call to sending and receiving of data to minimize latency and context switching overhead. The native send and receive threads implement the most latency critical logic in the native space which is not simply wrapping ibverbs functions to be exposed to Java (§7.2.3 and 7.2.4).. The counterpart to the native logic is implemented in Java (§8). In the end, we are able to reduce sending and receiving of data to a single context switching call.

6 Overview Ibdxnet and Java InfiniBand Transport

This section gives a brief top-down introduction of the full transport implementation. Figure 3 depicts the different components and layers involved when using InfiniBand with DXNet. The **Java InfiniBand transport (IB transport)**

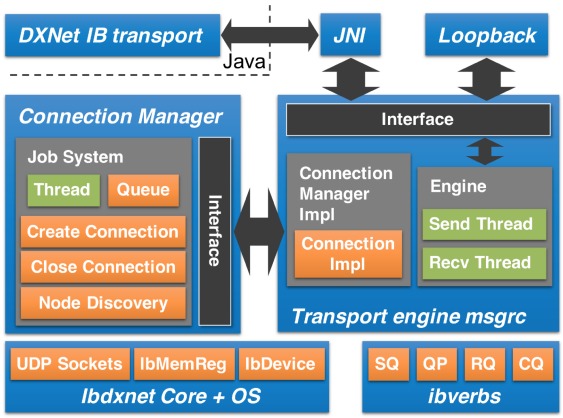

Figure 4: Simplified architecture of Ibdxnet with the msgrc transport engine

(§8) implements DXNet's transport interface (§2.5) and uses JNI to connect to the native counterpart. **Ibdxnet** uses the native ibverbs library to access the hardware and provides a separate subsystem for connection management, sending and receiving data. Furthermore, it implements a set of functions for the Java Native Interface to connect to the Java implementation.

7 Ibdxnet: Native InfiniBand Subsystem with Transport Engine

This section elaborates on the implementation details of our native InfiniBand subsystem **Ibdxnet** which is used by the IB transport implementation in DXNet to utilize InfiniBand hardware. Ibdxnet provides the following key features: a basic foundation with re-usable components for implementations using different means of communication (e.g. messaging verbs, RDMA) or protocols, automatic connection management and transport engines using different communication primitives. Figure 4 shows an outline of the different components involved.

Ibdxnet provides an **automatic connection and QP manager** (§7.1) which can be used by every transport engine. An interface for the connection manager and a connection object allows implementations for different transport engines. The engine **msgrc** (see Figure 4) uses the provided connection management and is based on RC messaging verbs. The engine **msgud** using UD messaging verbs is already implemented and will be discussed and extensively evaluated in a separate publication.

A **transport engine** implements its own protocol to send/receive data and exposes a low-level interface. It creates an abstraction layer to hide direct interaction with the ibverbs library. Through the low-level interface, a transport implementation (§8) provides data-to-send and forwards received data for further processing. For example: the low-level interface of the msgrc engine does not provide concurrency control or serialization mechanisms for messages. It accepts a stream of data in one or multiple buffers for sending and provides buffers creating a stream of data on receive (§7.2). This engine is connected to the Java transport counterpart via JNI and uses the existing infrastructure of DXNet (§8).

Furthermore, we implemented a **loopback** like stand alone transport for debugging and measuring performance of the native engine, only. The loopback transport creates a continuous stream of data for sending to one or multiple nodes and throws away any data received. This ensures that sending and receiving introduce no additional overhead and allows measuring the performance of different low-level aspects of our implementation. This was used to determine the maximum possible throughput with Ibdxnet (§9.2.4).

In the following sections, we explain the implementation details of Ibdxnet's connection manager (§7.1) and the messaging engine msgrc (§7.2). Additionally, we describe best practices for using the ibverbs API and optimizations for optimal hardware utilization. Furthermore, we elaborate on how Ibdxnet connects to the IB transport in Java using JNI and how we implemented low overhead data exchange between Java and native space.

7.1 Dynamic, Scalable and Concurrent Connection Management

Efficient connection management for many nodes is a challenging task. For example, hundreds of application threads want to send data to a node but the connection is not yet established. Who creates the connection and synchronizes access of other threads? How to avoid synchronization overhead or blocking of threads that want to get an already established connection? How to manage the lifetime of a connection?

These challenges are addressed by a **dedicated connection manager in Ibdxnet**. The connection manager handles all tasks required to establish and manage connections and hides them from the higher level application. For our higher level Java transport (§8.1), complexity and latency is reduced for connection setup by avoiding context switching.

First, we explain how nodes are identified, the contents of a connection and how online/offline nodes are discovered and handled. Next, we describe how existing connections are accessed and non-existing connections are created on the fly during application runtime. We explain the details how a connection creation job is handled by the internal job manager, how connection data is exchanged with the remote in order to create a QP. At last, we briefly describe our previous attempt which failed to address the above challenges properly.

A node is identified by **a unique 16-bit integer nodeID (NID)**. The NID is assigned to a node on start of the connection manager and cannot be changed during runtime. A con-

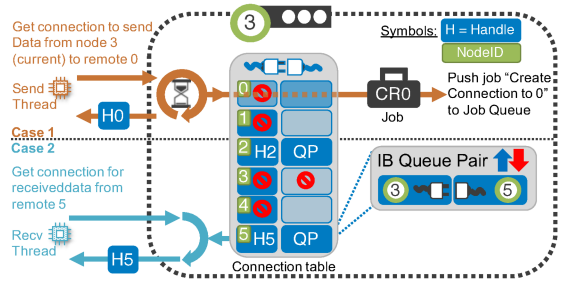

Figure 5: Connection manager: Creating non-existing connections (send thread: node 1 to node 0) and re-using existing connections (recv thread: node 1 to node 5).

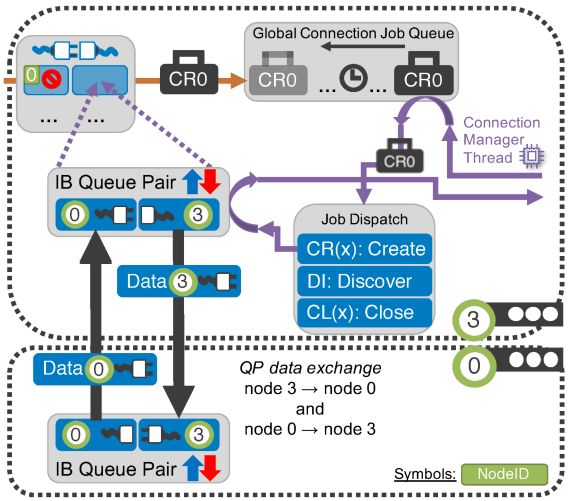

Figure 6: Connection manager: Automatic connection creation with QP data exchange (node 3 to node 0). The job *CR0* is added to the back of the queue to initiate this process. The dedicated thread processes the queue by removing jobs from the front and processing them according to their type.

nection consists of the source NID (the current node) and the destination NID (the target remote node). Depending on the transport implementation, an existing connection holds one or multiple ibverbs QPs, buffers and other data necessary to send and receive data using that connection. The connection manager provides a **connection interface for the transport engines** which allows them to implement their own type of connection. The following example describes a connection with a single QP, only.

Before a connection to a remote node can be established, the remote node must be discovered and known as available. The job type **node discovery** (further details about the job system follow in the next paragraphs) detects online/offline nodes using UDP sockets over Ethernet. On startup, a list of node hostnames is provided to the connection manager. The list can be extended by adding/removing entries during runtime for dynamic scaling. The discovery job tries to contact all non-discovered nodes of that list in regular intervals. When a node was discovered, it is removed from the list and marked as discovered. A connection can only be established with an already discovered node. If a connection to the node was already created and is lost (e.g. node crash), the NID is added back to the list in order to re-discovered the node on the next iteration of the job. Node discovery is mandatory for InfiniBand in order to exchange QP information on connection creation.

Figure 5 shows how existing connections are accessed and new connections are created when two threads, e.g. a send and a receive thread, are accessing the connection manager. The send thread wants to send new data to node 0 and the receive thread has received some data (e.g. from a SRQ). It has to forward it for further processing which requires information stored in each connection (e.g. a queue for the incoming data). If the connection is already established (the receive thread gets the connection to node 5), a connection handle (*H5*) is returned to the calling thread. If no connection has been established so far (the send thread wants to get the connection to node 0), **a job to create the specific connection** (*CR0* = create to node 0) is added to the internal job queue. The calling thread has to wait until the job is dispatched and the connection is created before being able to send the data.

Figure 6 shows how connection creation is handled by the internal job thread. The job *CR0* (yielded by the send thread from the previous example in figure 5) is pushed to the back of the job queue. The job queue might contain jobs which affect different connections, i.e. there is no per connection dedicated queue. **The dedicated connection manager thread** is processing the queue by removing a job from the front and dispatching by type. There are three types of jobs: create a connection to a node with a NID, discover other connection managers, close an existing connection to node.

To create a new connection with a remote node, the current node has to create an ibverbs QP with a SQ and RQ. Both queues are cross-connected to a remote QP (send with recv, recv with send) which requires data exchange using another communication channel (Sockets over Ethernet). For the job *CR0*, the thread creates a new QP on the current node (3) and exchanges its QP data with the remote it wants to connect to (0) using UDP sockets. The remote (0) also creates a QP and uses the received connection information (of 3). It replies with its own QP data (0 to 3) to complete QP creation. The newly established connection is added to the connection table and is now accessible (by the send and receive thread).

At last, we briefly describe our lessons learned from our first attempt for an automatic connection manager. It was relying on active connection creation. The first thread calling the connection manager to acquire a connection creates it on the fly, if it does not exist. The calling thread executes connection exchange, waits for the remote data and finishes connection creation. This requires coordination of all threads

accessing the connection manager either to create a new connection or getting an existing one. It introduced a very complex architecture with high synchronization overhead and latency especially when many threads are concurrently accessing the connection manager. Furthermore, it was error prone and difficult to debug. We encountered severe performance issues when creating connections with one hundred nodes in a very short time range (e.g. all-to-all communication). This resulted in connection creation times of up to half a minute. Even with a small setup of 4 to 8 nodes, creating a connection could take up to a few seconds if multiple threads tried to create the same or different connections simultaneously.

7.2 msgrc: Transport Engine for Messaging using RC QPs

This section describes the **msgrc** transport engine. It uses reliable QPs to implement messaging using a dedicated send and receive thread. The engine's interface allows a transport to provide a stream of data (to send) in form of variable sized buffers and provides a stream of data (received) to a registered callback handler.

This interface is rather low-level and the backend does not implement any means of serialization/deserialization for sending/receiving of complex data structures. In combination with DXNet (§2), the logic for these tasks resides in the Java space with DXNet and is shared with other transports such as the NIO Ethernet transport [9]. However, there are no restrictions to implement these higher level components for the msgrc engine natively, if required. Further details on how the msgrc engine is connected with the Java transport counterpart are given in Section 8.

The following subsections explain the general architecture and interface of the transport, sending and receiving of data using dedicated threads and how various features of InfiniBand were used for optimal hardware utilization.

7.2.1 Architecture

This section explains the basic architecture as well as the low-level interface of the engine. Figure 4 includes the msgrc transport and can be referred to for an abstract representation of the most important components. The engine relies on our dedicated connection manager (§7.1) for connection handling. We decided to use one dedicated thread for sending (§7.2.3) and one for receiving (§7.2.4) to benefit from the following advantages: a clear separation of responsibilities resulting in a less complex architecture, no scheduling of send/receive jobs when using a single thread for both and higher concurrency because we can run both threads on different CPU cores concurrently. The architecture allows us to create decoupled pipeline stages using lock-free queues and ring buffers. Thereby, we avoid complex and slow synchronization between the two threads and with hundreds of threads concurrently accessing shared resources.

7.2.2 Engine interface

```
1  struct NextWorkPackage {
2      uint32_t posBackRel;
3      uint32_t posFrontRel;
4      uint8_t flowControlData;
5      uint16_t nodeId;
6  };
7
8  struct PrevWorkPackageResults {
9      uint16_t nodeId;
10     uint32_t numBytesPosted;
11     uint32_t numBytesNotPosted;
12     uint8_t fcDataPosted;
13     uint8_t fcDataNotPosted;
14 };
15
16 struct CompletedWorkList {
17     uint16_t numNodes;
18     uint32_t
           bytesWritten[NODE_ID_MAX_NUM_NODES];
19     uint8_t
           fcDataWritten[NODE_ID_MAX_NUM_NODES];
20     uint16_t nodeIds[];
21 };
22
23 NextWorkPackage*
       GetNextDataToSend(PrevWorkPackageResults*
       prevResults, CompletedWorkList*
       completionList);
```

Listing 1: Structures and callback of the msgrc engine's send interface

The low-level interface allows fine-grained control for the target transport over the engine. The interface for sending data is depicted in Listing 1 and receiving is depicted in Listing 2. Both interfaces create an abstraction hiding connection and QP management as well as how the hardware is driven with the ibverbs library. For sending data, the interface provides the callback *GetNextDataToSend*. This function is called by the send thread to pull new data to send from the transport (e.g. from the ORB, see 8.2). When called, an instance of each of the two structures *PrevWorkPackageResults* and *CompletedWorkList* are passed to the implementation of the callback as parameters: the first contains information about the previous call to the function and how much data was actually sent. If the SQ is full, no further data can be sent. Instead of introducing an additional callback, we combine getting the next data with returning information about the previous send call to reduce call overhead (important for JNI access). The second parameter contains data about completed work requests, i.e. data sent for the transport. This must be used in the transport to mark data processed (e.g.

moving the pointers of the ORB).

```
struct ReceivedPackage {
    uint32_t count;
    struct Entry {
        uint16_t sourceNodeId;
        uint8_t fcData;
        IbMemReg* data;
        void* dataRaw;
        uint32_t dataLength;
    } m_entries[];
};

struct IncomingRingBuffer {
    uint32_t m_usedEntries;
    uint32_t m_front;
    uint32_t m_back;
    uint32_t m_size;

    struct Entry {
        con::NodeId m_sourceNodeId;
        uint8_t m_fcData;
        uint32_t m_dataLength;
        core::IbMemReg* m_data;
        void* m_dataRaw;
    } m_entries[];
};

uint32_t Received(IncomingRingBuffer*
    ringBuffer);

void ReturnBuffer(IbMemReg* buffer);
```

Listing 2: Structure and callback of the msgrc engine's receive interface

If data is received, the receive thread calls the callback function *Received* with an instance of the *IncomingRingBuffer* structure as its parameter. This parameter holds a list of received buffers with their source NID. The transport can iterate this list and forward the buffers for further processing such as de-serialization. If the transport has to return the number of elements processed and, thus, is able to control the amount of buffers it can process. Once the received buffers are processed by the transport, they must be returned back to the *RecvBufferPool* by calling *ReturnRecvBuffer* to allow re-using them for further receives.

7.2.3 Sending of Data

This section explains the data and control flow of the **dedicated send thread** which **asynchronously** drives the engine for sending data. Listing 3 depicts a simplified version of the contents of its main loop with the relevant aspects for this section. Details of the functions involved in the main flow are explained further below.

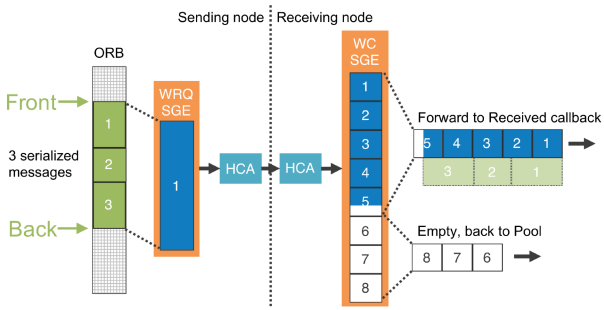

Figure 7: Example for sending and receiving data using scatter gather elements: Get data (aggregated messages) from ORB, send 1 SGE. Receive data scattered to multiple receive buffers.

```
1  workPackage =
       GetNextDataToSend(prevWorkResults,
       completionList);
2  Reset(prevWorkResults);
3  Reset(completionList);
4
5  if (workPackage != NULL) {
6    connection =
         GetConnection(workPackage.nodeId);
7    prevWorkResults = SendData(connection,
         workPackage);
8    ReturnConnection(connection);
9  }
10
11 completionList = PollCompletions();
```

Listing 3: Send thread main flow (simplified)

The loop starts with getting a *workPackage*, the next data to send (line 1), using the engine's low-level interface (§7.2.2). The instance *prevWorkResults* contains information about posted and non-posted data from the previous loop iteration. The instance *completionList* holds data about completed sends. Both instances are reseted/nulled (line 2-3) for re-use in the current iteration.

If the *workPackage* is valid (line 5), i.e. data to send is available, the *nodeId* from that package is used to get the *connection* to the send target from the connection manager (line 6). The *connection* and *workPackage* are passed to the *SendData* function (line 7). It processes the *workPackage* and returns how much data was processed, i.e. posted to the SQ of the connection, and how much data could not be processed. The latter happens if the SQ is full and must be kept track of to not lose any data. Afterwards, the thread returns the *connection* to the connection manager (line 8).

At the end of a loop iteration, the thread polls the SCQ to remove any available WCs. **We share the completion queue among all SQs/connections to avoid iterating over**

many connections for a task. The loop iteration ends and the thread starts from the beginning by calling *GetNextDataToSend* and provides the work results of our previous iteration. Data about polled WCs from the SCQ are stored in the *completionList* and forwarded via the interface (to the transport).

If no data is available (line 5), lines 6-8 are skipped and the thread executes a completion poll, only. This is important to ensure that any outstanding WCs are processed and passed to the transport (via the *completionList* and calling *GetNextDataToSend*). Otherwise, if no data is sent for a while, the transport will not receive any information about previously processed data. This leads to false assumptions about the available buffer space for sending data, e.g. assuming that data fits into the buffer but actually does not because the processed buffer space is not free'd, yet.

In the following paragraphs, we further explain how the functions *SendData* and *PollCompletions* make optimal use of the ibverbs library and how this cooperates with the interleaved control flow of the main thread loop explained above.

The **SendData** function is responsible for preparing and posting of FC data and normal data (payload). FC data, which determines the number of flow control windows to confirm, is a small number (< 128) and, thus, does not require a lot of space. We post it as part of the **immediate data**, which can hold up to 4 bytes of data, with the WR instead of using a separate side channel, e.g. another QP. This avoids overhead of posting and polling of another QP which **benefits overall performance, especially with many simultaneous connections**. With FC data using 1 byte of the immediate data field, we use further 2 bytes to include the NID of the source node. This allows us to identify the source of the incoming WC on the remote. Otherwise, identifying the source would be very inconvenient. The only information provided with the incoming WC is the sender's unique physical QP id. In our case, this id must be mapped to the corresponding NID of the sender. However, this introduces an indirection every time a package arrives which hurts performance.

For sending normal data (payload), the provided *workPackage* holds two pointers, front and back, which enclose a memory area of data to send. This memory area belongs to a buffer (e.g. the ORB) which was registered with the protection domain on start to allow access by the HCA. Figure 7 depicts an example with three (aggregated) ready to send messages in the ORB. We create a WR for the data to send and provide a single **SGE which takes the pointers of the enclosed memory area**. The HCA will directly read from that area without further copying of the data (zero copy). For buffer wrap arounds, two SGEs are created and attached to one WR: one SGE for the data from the front pointer to the end of the buffer, another SGE for the data from the start of the buffer to the back pointer. If the size of the area to send (sum of all SGEs) exceeds the maximum configurable receive size, the data to send must be sliced into multiple WRs. **Multiple WRs are chained to a link list to minimize call overhead** when posting them to the SQ using *ibv_post_send*. This greatly increases performance compared to posting multiple standalone WRs with single calls.

The number of SGEs of a WR can be 0, if no normal data is available to send but FC data is available. To send FC data only, we write it to the immediate data field of a WR along with our source NID and post it without any SGEs attached which results in a 0 length data WR.

The **PollCompletions** function calls *ibv_poll_cq*, **once, to poll for any completions available** on the SCQ. A SCQ is used instead of per connection CQs to avoid iterating the CQs of all connections which impacts performance. The send thread keeps track of the number of posted WRs and, thus, knows how many WCs are outstanding and expected to arrive on the SCQ. If none are being expected, polling is skipped. *ibv_poll_cq* is called once per PollCompletion call, only, and every call tries to poll WCs in batches to keep the call overhead minimal.

Experiments have shown that most calls to *ibv_poll_cq*, even on high loads, will return empty, i.e. no WRs have completed. Thus, polling the SCQ until at least one completion is received is the wrong approach and greatly impacts overall performance. If the SQ of another connection is not full and there is data available to send, this method wastes CPU resources on busy polling instead of processing further data to send. The performance impact (resulting in low throughput) increases with the number of simultaneous connections being served. Furthermore, this increases the chance of SQs running empty because time is wasted on waiting for completions instead of keeping all SQs filled. **Full SQs ensure that the HCA is kept busy which is the key to optimal performance**.

7.2.4 Receiving of Data

```
1  workCompletions = PollCompletions();
2
3  if (recvQueuePending < ibqSize) {
4      Refill();
5  }
6
7  if (workCompletions > 0) {
8    ProcessCompletions(workCompletions);
9  }
10
11 if (!IncomingRingBufferIsEmpty()) {
12     DispatchReceived();
13 }
```

Listing 4: Receive thread main flow (simplified)

Analogous to Section 7.2.3, this section explains the data and control flow of the **dedicated receive thread** which

asynchronously drives the engine for receiving data. Listing 4 depicts a simplified version of its main loop with the relevant aspects for this section. Details of the functions involved in the main flow are explained further below.

Data is received using a SRQ and SCQ instead of multiple receive and completions queues. This avoids iterating over all open connections and checking for data availability which introduces overhead with increasing number of simultaneous connections. Equally sized buffers for receiving data (configurable size and amount) are pooled and returned for re-use by the transport, once processed (§7.2.2).

The loop starts by calling *PollCompletions* (line 1) to poll the SCQ for WCs. Before processing the WCs returned, the SRQ is refilled by calling *Refill* (line 4), if the SRQ is not filled, yet. Next, if any WCs were polled previously, they are processed by calling **ProcessCompletions** (line 8). This step pushes them to the **Incoming Ring Buffer (IRB)**, a temporary ring buffer, before dispatching them. Finally, if the IRB is not empty (line 11), the thread tries to forward the contents of the IRB by calling *DispatchReceived* via the interface to the transport (§7.2.2).

The following paragraphs are further elaborating on how *PollCompletions*, *Refill*, *ProcessCompletions* and *DispatchReceived* make optimal use of the ibverbs library and how this cooperates with the interleaved control flow of the main thread loop explained above.

The **PollCompletions** function is very similar to the one explained in Section 7.2.3 already. WCs are polled in batches of max. currently available IRB space and buffered before being processed.

The **Refill** function adds new receive WRs to the SRQ, if the SRQ is not completely filled and receive buffers from the receive buffer pool are available. Every WR consists of a configurable number of SGEs which make up the maximum receive size. This is also the limiting size the send thread can post with a single WR (sum of sizes of SGE list). Using this method, the receive thread does not have to take care of any software slicing of received data because the HCA scatters one big chunk of send data transparently to multiple (smaller) receive buffers on the receiver side. At last, *Refill* chains the WRs to a linked list which is posted on a single call to *ibv_post_srq_recv* for minimal overhead.

If WCs are buffered from the previous call to *PollCompletions*, the **ProcessReceived** function iterates this list of WCs. For each WC of the list, it gets the source NID and FC data from the immediate data field. If the recv length of this WC is non zero, the attached SGEs contain the received data scattered to the receive buffers of the SGE list.

As the receive thread does not know or have any means of determining the size of the next incoming data, the challenge is optimal receive buffer usage with minimal internal fragmentation. Here, fragmentation describes the amount of receive buffers provided with a WR as SGEs in relation to the amount of received data written to that block of buffers. The less data written to the buffers, the higher the fragmentation. In the example shown in figure 7, the three aggregated and serialized messages are received in five buffers but the last buffer is not completely used.

This fragmentation cannot be avoided but handled to avoid negative results like empty buffer pools or low per buffer utilization. Receive buffers/SGEs of a WR that do not contain any received data, because the amount of received data is less than the total size of the list of buffers of the SGE list, are pushed back to the buffer pool. All receive buffers of the SGE list that contain valid received data are pushed to the IRB (in the order they were received).

Depending on the target application, the fragmentation degree can be lowered if one configures the receive buffer and pool sizes accordingly. Applications typically sending small messages are performing well with small receive buffer sizes. However, throughput might decrease slightly for applications sending mainly big messages on small receive buffer sizes requiring more WRs per send data send (data sliced into multiple WRs).

If the IRB contains any elements, the **DispatchReceived** function tries to forward them to the transport via the *Received* callback (§7.2.2). The callback returns the number of elements it consumed from the IRB and, thus, is allowed to consume none or up to what's available. The consumed buffers are returned asynchronously to the receive buffer pool by transport, once it finished processing them.

7.2.5 Load Adaptive Thread Parking

The send and receive threads must be kept busy running their loops to send and receive data as fast as possible to ensure low latency. However, pure busy polling without any sleeping or yielding introduces high CPU load and occupying two cores of the CPU permanently. This is unnecessary during periods when the network is not used frequently. We do not want the send and receive threads to waste CPU resources and, therewith, decrease the overall node performance. Experiments have shown that simply adding sleep or yield operations highly impacts network latency and throughput and introduces high fluctuations [8].

To solve this, we used a simple but efficient wait pattern we call *load adaptive thread parking*. After a defined amount of time (e.g. 100 ms) of polling and no data available, the thread enters a yield phase and calls yield on every loop iteration if no data is available. After another timeframe passed (e.g. 1 sec), the thread enters a parking phase calling sleep/park with a minimum value of 1 ns on every loop iteration reducing CPU load significantly. The lowest value possible (1 ns) ensure that the scheduler of the operating system sends the thread sleeping for the shortest period of time possible. Once data is available, the current phase is interrupted and the timer is reset. This ensures busy looping for the next iterations keeping latency for successive messages and on high

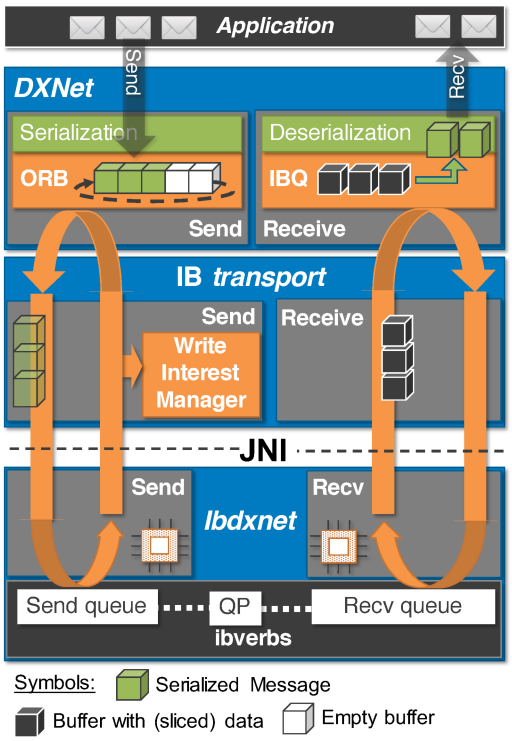

Figure 8: Components of Ibdxnet, IB transport and DXNet involved in data and control flow (simplified).

loads low. For further details including evaluation results refer to our DXNet publication [8].

8 IB Transport Implementation in DXNet (Java)

This section describes the transport implementation for DXNet in Java which utilizes the low-level transport engines, e.g. msgrc (§7.2), provided by Ibdxnet (§7). We describe the native interface which implements the low-level interface exposed by the engine (§7.2.2) and how it is used in the DXNet IB transport for higher level connection management (§8.1), sending serialized data from the ORB (§8.2) and handling incoming receive buffers from remote nodes (§8.3).

Figure 8 depicts the involved components with the main aspects of their data and control flow which are referred to in the following subsection.

If an application wants to send one or multiple messages, it calls DXNet which serializes them into the ORB and signals the WriteInterestManager (WIM) about available data (§2.2). The native send thread checks the WIM for data to send periodically and, if available, gets it from the ORB. Depending on the size, the data to send might be sliced into multiple elements which are posted to the SQ as one or multiple work requests (§7.2.3).

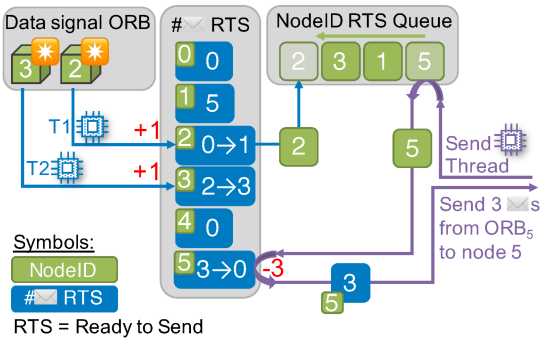

Figure 9: Internals of the Write Interest Manager (WIM).

Received data on the recv queue is written to one or multiple buffers (depending on the amount of data) from a native buffer pool (§7.2.4). Without further processing, the buffers are forwarded to the Java space and pushed to the IncomingBufferQueue (IBQ). DXNet's de-serialization is processing the buffers in order and creates messages (Java objects) which are dispatched to pre-registered callbacks using dedicated message handler threads (§2.3).

8.1 Connection Handling

To implement new transports in DXNet, it provides an interface to create specific connection types for the transport to implement. The DXNet core, which is shared across all transport implementations, manages the connections for the target application by automatically creating new connections on demand or closing connections if a configurable threshold is exceeded (§2.1).

For the IB transport implementation, the derived connection does not have to store further data or implement functionality. This is already stored and handled by the connection manager of Ibdxnet. It reduces overall architectural complexity by avoiding split functionality between Java and native space. Furthermore, it avoids context switching between Java and native code.

Only the NID of either the target node to send to or the source node of the received data is exchanged between the Java and native space and vice versa. Thus, **Connection setup** in the transport implementation in Java is limited to creating the Java connection object for DXNet's connection manager. **Connection close and cleanup** is similar with an additional callback to the native library to signal a connection was closed to Ibdxnet's connection management.

8.2 Dispatch of Ready-to-send Data

The engine msgrc is running dedicated threads for sending data. The send thread pulls new data from the transport via the *GetNextDataToSend* function of the low-level interface

(§7.2.2, §7.2.3). In order to allow this and other callbacks (for connection management and receiving data) to be available to the IB transport, a lightweight JNI binding with the aspects explained in Section 5 was created. The transports implement the *GetNextDataToSend* function exposed by the JNI binding. To get new data to send, the send thread calls the JNI binding which is implemented in the IB transport in Java.

Next, we elaborate on the implementation of *GetNextDataToSend* in the IB transport, how the send thread gets data to send and how the different states for the data (posted, not posted, send completed) are handled in combination with the existing ORB data structure.

Application threads using DXNet and sending messages are concurrently serializing them into the ORB (§2.2). Once serialization completes, the thread signals the transport that there is ready to send (RTS) data in the ORB. For the IB transport, this signal **adds a write interest to the dedicated Write Interest Manager (WIM)**. The WIM manages interest tokens using a lock-free list (based on a ring buffer) and a per connection atomic counter for both, RTS normal data from the ORB and FC data. Each type has a separate atomic counter, but, if not explicitly stated, we refer to them as one for ease of comprehension.

The list contains the nodeIDs of the connections that have RTS data in the order they were added. The atomic counter is used to keep track of the number of interests signalled, i.e. the number of times the callback was triggered for the selected NID.

Figure 9 depicts this situation with two threads (T1 and T2) which finished serializing data to the ORBs of two independent connections (3 and 2). The table with atomic counters keeps track of the number of signaled interests for RTS data/messages per connection. By calling *GetNextDataToSend*, the send thread from Ibdxnet checks a lock-free list which contains nodeIDs of the connections with at least one write interest available. The nodeIDs are added in order to the list but only if it is not already in the list. This is detected by checking if the atomic counter returned 0 after a fetch and add operation. This mechanism ensures that data from many connection is processed in a round robin fashion. Furthermore, avoiding duplicates in the queue sets an upper bound for memory requirement which is *sizeof(nodeID) * maxNumConnections*. Otherwise, the queue can grow depending on the load and number of active connections. If the queue of the WIM is empty, the send thread aborts and returns to the native space.

The send thread uses the NID it removed from the queue to get and reset the number of interests of the corresponding atomic counter. If there are any interests available for FC data, the send thread processes them by getting the FC from the connection and getting, but not yet removing, the stored FC data. For interests concerning normal data, the send thread gets the ORB from the connection and reads the

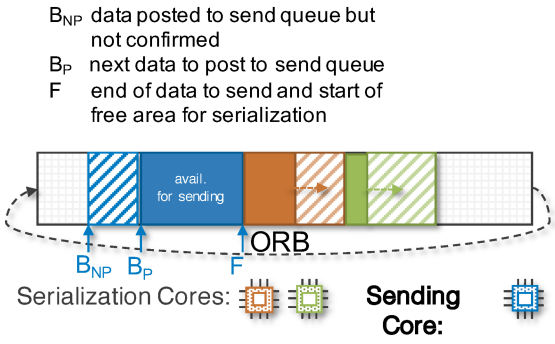

Figure 10: Extended outgoing ring buffer used by IB transport.

current front and back pointers. The pointers of the ORB are not modified, only read (details below). With this data, along with the NID of the connection, the send thread returns to the native space for processing (§7.2.3).

Every time the send thread returns to the Java space to get more data to send, it carries the parameters *prevWorkResults*, which contains data about the previous send operation, and *completionList*, which contains data about completed WRs, i.e. data send confirmations (§7.2.3). For performance reasons, this data resides in native memory as structs and is mapped and accessed using DirectByteBuffers (§5).

The asynchronous workflow used to send and receive data by posting WRs and polling WCs must be adopted by updating the ORB and FC accordingly. Depending on the fill level of the SQ, the send thread might not be able to post all normal data or FC it retrieved in the previous iteration. The *prevWorkResults* parameter contains this information about how much normal and FC data was processed and could not be processed. This information must be preserved for the next send operation to avoid sending data multiple times. For the ORB however, we cannot move the front pointer because this frees up the memory which is not confirmed to be sent, yet.

Thus, we introduce a second front pointer, front posted, which is only known to and modified by the send thread and allows it to keep track of already posted data. Figure 10 depicts the most important aspects of the enhanced ORB which is used for the IB transport. In total, this creates three virtual areas of memory designated to the following states:

- Data posted but not confirmed: front to front posted
- Data RTS and not posted: front posted to back
- Free memory for send threads to serialize to: back to front

Using the parameter *prevWorkResults*, the front posted pointer is moved by the amount of data posted. Any non processed data remains unprocessed (front posted not moved to

cover entire area of RTS data). For data provided with the parameter *completionList*, the front pointer is updated according to the number of bytes now confirmed to be sent. A similar but less complex approach is applied to updating FC.

8.3 Process Incoming Buffers

The dedicated receive thread of msgrc is pushing received data to the low-level interface. Analogous to how RTS data is pulled from the IB transport via the JNI binding, the receive thread uses a received function provided by the binding to push the received buffers to the IB transport into Java space. All received buffers are stored as a batch in the *recvPackage* data structure (§7.2.2) to minimize context switching overhead. For performance reasons, this data resides in native memory as structs and is mapped and accessed using DirectByteBuffers (§5).

The receive thread iterates the package in Java space, dispatches received FC data to each connection and pushes the received buffers (including the connection of the source node) to the IBQ (§2.3). The buffers are handled and processed asynchronously by the MessageCreationCoordinator and one or multiple MessageHandlers of the DXNet core (all of them are Java threads). Once the buffers are processed (de-serializing its contents), the Java threads return them asynchronously to the transport engines receive buffer pool (§7.2.4).

9 Evaluation

For better readability, we refer to DXNet with the IB transport Ibdxnet and msgrc engine as DXNet from here onwards.

We implemented commonly used microbenchmarks to compare DXNet to two MPI implementations supporting InfiniBand: MVAPICH2 and FastMPJ. We decided to compare against two MPI implementations for the following reasons: To the best of our knowledge, there is no other system available that offers all features of DXNet and big data applications implementing their dedicated network stack do not offer it as a separate application/library like DXNet does. MPI can be used to partially cover some features of DXNet but not all (§3). We are aware that MPI is targeting a different application domain, mainly HPC, whereas DXNet is targeting big data. However, MPI was already used in big data applications as well and several aspects related to the network stack and the technologies are overlapping in both application domains.

Bandwidth with two nodes is compared using typical uni- and bi-directional benchmarks. We also compared scalability using an all-to-all benchmark (worst-case scenario) with up to 8 nodes. Latency is compared by measuring the RTT with a request-response communication pattern.

	FastMPJ	MVAPICH2	DXNet
Uni-dir. TP ST	x	x	x
Bi-dir. TP ST	x	x	x
Latency ST	x	x	x
All-to-all TP ST	x	x	x
Uni-dir. TP MT			x
Bi-dir. TP MT		x	x
Latency MT			x
All-to-all MT			x

Table 1: Summary of benchmarks and systems. TP = throughput, ST = single threaded, MT = multi-threaded

These benchmarks are executed single threaded to compare all three systems.

Furthermore, we compared how DXNet and MVAPICH2 perform in a multi-threaded environment which is typical for Big Data but not HPC applications. However, we can only compare it using three benchmarks. Latency multi-threaded is not possible since it would require MVAPICH2 to implement additional infrastructure to store and map requests with responses and dynamic dispatching callbacks to handlers of incoming data to multiple receive threads (similar to DXNet). MVAPICH2 does not provide such a processing pipeline. FastMPJ cannot be compared at all here because it only supports single threaded environments. Table 1 summerizes the systems and benchmarks executed.

All benchmarks were executed on up to 8 nodes of our private cluster, each with a single socket Intel Xeon CPU E5-1650 v3, 6 cores running at 3.50 GHz per core clock speed and 64 GB RAM. The nodes are running Ubuntu 16.04 with kernel version 4.4.0-57. All nodes are equipped with a Mellanox MT27500 HCA, connected with 56 Gbps links to a single Mellanox SX6015 18 port switch. For Java applications, we used the Oracle JVM version 1.8.0_151.

9.1 Benchmarks

The *osu* benchmarks included with MVAPICH2 implement typical micro benchmarks to measure uni- and bi-directional bandwidth and uni-directional latency which reflect basic usage of any network stack for point-to-point communication. *osu_latency* is used as a foundation and extended with recording of all RTTs to determine the 95th, 99th and 99.9th percentile after execution. The latency measured is the full RTT when the source is sending a request to the destination up to when the corresponding response is received by the source. For evaluating throughput, the benchmarks *osu_bw* and *osu_bibw* were combined to a single benchmark and extended to enable all-to-all bi-directional execution with more than two nodes. We consider this a relevant benchmark to show if the system is capable of handling multiple connections under high load. This is a common situation found in

big data applications as well as backend storages [11]. On all-to-all, every node receives from all other nodes and sends messages to all other nodes in a round robin fashion. The bi-directional and all-to-all results presented are the aggregated send throughputs of all participating nodes. We added options to support multi-threaded sending and receiving using a configurable number of send and receive threads. As the per-processor core count increases, the multi-threading aspect becomes more and more important. Furthermore, our target application domain big data relies heavily on multi-threaded environments.

For the evaluation of FastMPJ, we ported the *osu* benchmarks to Java. The benchmarks for evaluating a multi-threaded MPI process were omitted because FastMPJ does not support multi-threaded processes. DXNet comes with its own benchmarks already implemented which are comparable to the *osu* benchmarks.

The *osu* benchmarks use a configurable parameter *window_size* (WS) which denotes the number of messages sent in a single batch. Since MPI does not support implicit message aggregation like DXNet, we executed all MPI experiments with increasing WS to determine bandwidth peaks and saturation under optimal conditions and ensure a fair comparison to DXNet's built in aggregation. No MPI collectives are required for the benchmarks and, thus, aren't evaluated.

All benchmarks are executed three times and their variance is displayed using error bars. Throughputs are specified in GB/s, latencies/RTTs in us and message rates in mmps (million messages per second). All throughput benchmarks send 100 million messages and all latency benchmarks 10 million messages. The total number of messages is incrementally halved starting with 4 kb message size to avoid unnecessary long running benchmark runs. All throughputs measured are based on the total amount of sent payload bytes. This does not include any overhead like message headers or envelopes that are required by the systems for message identification or routing.

Furthermore, we included the results of the ib perf tools *ib_write_bw* and *ib_write_lat* as baselines to all end-to-end type benchmarks. These simple perf tools cannot be compared directly to the complex systems evaluated. But, these baselines show the best possible network performance (without any overhead by the evaluated system) and for rough comparisons of the systems across multiple plots. We chose parameters that reflect the configuration values of DXNet as close as possible (but still allow comparisons to FastMPJ and MVAPICH2 as well): receive queue size 2000 and send queue size 20 for both bandwidth and latency measurements; 100,000,000 messages for bandwidth and 10,000,000 for latency.

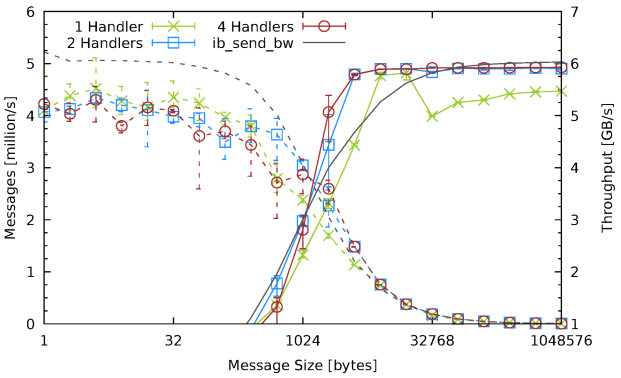

Figure 11: **DXNet**: 2 nodes, uni-directional throughput and message rate with one application send thread, increasing message size and number of message handlers

9.2 DXNet with Ibdxnet Transport

We configured DXNet using the parameters depicted in Table 2. The configuration values were determined with various debugging statistics and experiments, and are currently considered optimal configuration parameters.

For comparing single threaded performance, the number of application threads and message handlers (referred to as MH) is limited to one each to allow comparing it to FastMPJ and MVAPICH2. DXNet's multi-threaded architecture does not allow combining the logic of the application send thread and a message handler into a single thread. Thus, DXNet's "single threaded" benchmarks are always executed with one dedicated send and one dedicated receive thread.

The following subsections present the results of the various benchmarks. First, we present the results of all single threaded benchmarks with one send thread: uni- and bi-directional throughput, uni-directional latency and all-to-all with increasing node count. Afterwards, the results of the same four benchmarks are presented with multiple send threads.

9.2.1 Uni-directional Throughput

The results of the uni-directional benchmark are depicted in figure 11. Considering one MH, DXNet's throughput peaks at 5.9 GB/s at a message size of 16 kb. For larger messages (32 kb to 1 MB), one MH is not sufficient to de-serialize and dispatch all incoming messages fast enough and drops to a peak bandwidth of 5.4 GB/s. However, this can be resolved by simply using two MHs. Now, DXNet's throughput peaks and saturates at 5.9 GB/s with a message size of just 4 kb and stays saturated up to 1 MB. Message sizes smaller than 4 kb also benefit significantly from the shorter receive processing times by utilizing two MHs. Further MHs can still improve performance but only slightly for a few message sizes.

For small messages up to 64 bytes, DXNet achieves peak

IBQ max. capacity buffer count	8192
IBQ max. capacity aggregated data size	128 MB
Message handlers	varying (see experiments)
IB SQ size (per connection)	20
IB SRQ size	2000 (default value for up to 100 connections)
Max. connection limit	100
Recv buffer pool capacity	4 GB
Flow control window	16 MB
Flow control threshold	0.1
Receive buffer size (for small message sizes 1 - 16 kb)	32 kb
SGEs per WR (for small message sizes 1 - 16 kb)	4
Receive buffer size (for medium/large message sizes 32 kb - 1 MB)	1 MB
SGEs per WR (for medium/large message sizes 32 kb - 1 MB)	1

Table 2: DXNet configuration values for experiments

message rates of 4.0-4.5 mmps using one MH. Multiple MHs cannot significantly increase performance for such small messages further. However, with growing message size (512 byte to 16 kb), the message rate can be increased with two message handlers.

Compared to the baseline performance of *ib_send_bw*, DXNet's peak performance is approx. 0.5 to 1.0 mmps less. With increasing message size, this gap closes and DXNet even surpasses the baseline 1 kb to 32 kb message sizes when using multiple threads. DXNet peaks close to the baseline's peak performance of 6.0 GB/s. The results with small message sizes are fluctuating independent of the number of MHs. This can be observed on all other benchmarks with DXNet measuring message/payload throughput as well. It is a common issue which can be observed when running high load throughput benchmarks using the bare ibverbs API as well.

This benchmark shows that DXNet is capable of handling a vast amount of small messages efficiently. The application send thread and, thus, the user does not have to bother with aggregating messages explicitly because DXNet handles this transparently and efficiently. The overall performance benefits from multiple message handlers increasing receive throughput. Large messages do impact performance with one MH because the de-serialization of data consumes most of the processing time during receive. However, simply adding at least another MH solves this issue and further increases performance.

9.2.2 Bi-directional Throughput

Figure 12 depicts the results of the bi-directional benchmark with one send thread. With one MH, the aggregated throughput peaks at approx. 10.4 GB/s for 8 kb. Using two message handlers, the fluctuations starting with 16 kb messages using one MH can be resolved (as already explained in 9.2.1). Further increasing the performance using four MHs is not possible in this benchmark and actually degrades it for 512

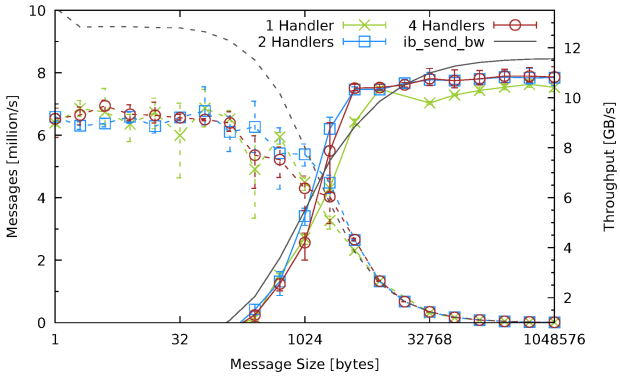

Figure 12: **DXNet**: 2 nodes, bi-directional throughput and message rate with one application send thread, increasing message size and number of message handlers

byte to 2 kb message sizes. DXNet's throughput peaks at approx. 10.4 GB/s and saturates with a message size of 32 kb.

The peak aggregated message rate for small messages up to 64 bytes is varying from approx. 6 to 6.9 mmps with one MH. Using more MHs cannot improve performance significantly for this benchmark. Due to the multi-threaded and highly pipelined architecture of DXNet, these variations cannot be avoided, especially when exclusively handling many small messages.

Compared to the baseline performance of *ib_send_bw*, there is still room for improvement for DXNet's performance on small message sizes (up to 2.5 mmps difference). For medium message sizes, *ib_send_bw* yields slightly higher throughput for up to 1 kb message size. But, DXNet surpasses *ib_send_bw* on 1 kb to 16 kb message size. DXNet's peak performance is approx. 1.1 GB/sec less than *ib_send_bw*'s (11.5 GB/sec).

Overall, this benchmark shows that DXNet can deliver

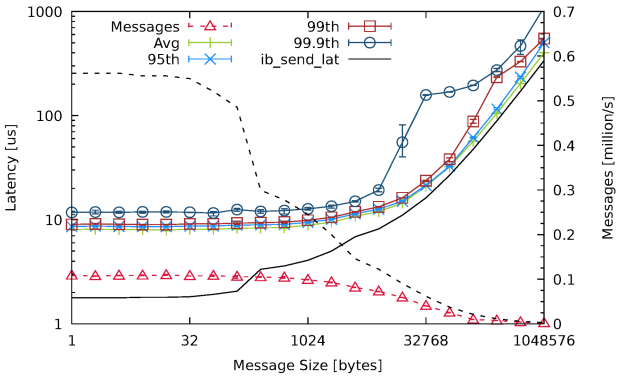

Figure 13: **DXNet**: 2 nodes, uni-directional RTT and message rate with one application send thread, increasing message size

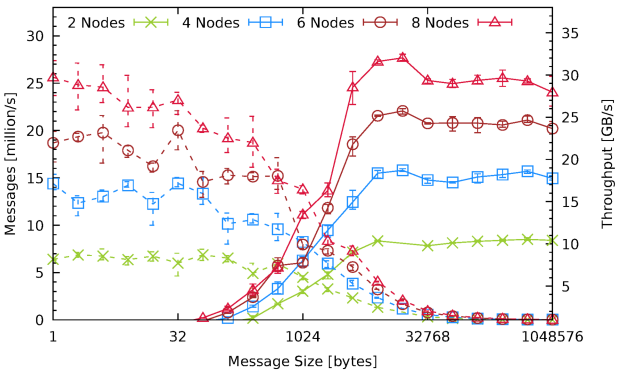

Figure 14: **DXNet**: 2 to 8 nodes, all-to-all aggregated send throughput and message rate with one application send thread, increasing message size and one message handler

great performance especially for small messages similar to the uni-directional benchmark (§9.2.1).

9.2.3 Uni-directional Latency

Figure 13 depicts the average RTTs as well as the 95th, 99th and 99.9th percentile of the uni-directional latency benchmark with one send thread and one MH. For message sizes up to 512 bytes, DXNet achieves an avg. RTT of 7.8 to 8.3 µs, a 95th percentile of 8.5 to 8.9 µs, a 99th percentile of 8.9 to 9.2 and 99.9th percentile of 11.8 to 12.7 µs. This results in a message rate of approx 0.1 mmps. As expected, starting with 1 kb message size, latency increases with increasing message size.

The RTT can be broken down into three parts: DXNet, Ibdxnet and hardware processing. Taking the lowest avg. of 7.8 µs, DXNet requires approx. 3.5 µs of the total RTT (the full breakdown is published in our other publication [8]) and the hardware approx. 2.0 µs (assuming avg. one way latency of 1 µs for the used hardware). Message de- and serialization as well as message object creation and dispatching are part of DXNet. For Ibdxnet, this results in approx. 2.3 µs processing time which includes JNI context switching as well as several pipeline stages explained in the earlier sections.

Compared to the baseline performance of *ib_send_lat*, DXNet's latency is significantly higher. Obviously, additional latency cannot be avoided with such a long and complex processing pipeline. Considering the breakdown mentioned above, the native part Ibdxnet, which calls ibverbs to send and receive data, is to some degree comparable to the minimal perf tool *ib_send_bw*. With a total of 2.3 µs (of the full pipeline's 7.8 µs), the total RTT is just slightly higher than *ib_send_bw*'s 1.8 µs. But, Ibdxnet already includes various data structures for state handling and buffer scheduling (§7.2.3, §7.2.4) which *ib_send_bw* doesn't. Buffers for sending data are re-used instantly and the data received is discarded immediately.

9.2.4 All-to-all Throughput with up to 8 Nodes

Figure 14 shows the aggregated send throughput and message rate of all participating nodes (up to 8) executing the all-to-all benchmark with one send thread and one MH with increasing message size. For small up to 64 byte messages, peak message rates of 7.0 mmps, 14.5 mmps, 20.1 mmps and 25.6 mmps are achieved for 2, 4, 6 and 8 nodes. Throughput increases with increasing node count peaking at 8 kb message size with 10.4 GB/s for 2 nodes. The peaks for 4, 6 and 8 nodes are reached with 16 kb message size at 18.9 GB/s, 26.0 GB/s and 32.4 GB/s. Incrementally adding two nodes, throughput is increased by 8.5 GB/s (for 2 to 4 nodes), by 7.1 GB/s (for 4 to 6 nodes) and 6.4 GB/s (for 6 to 8 nodes). One would expect approx. equally large throughput increments but the gain is noticeably lowered with every two nodes added.

We tried different configuration parameters for DXNet and ibverbs like different MTU sizes, SGE counts, receive buffer sizes, WRs per SQ/SRQ or CQ size. No combination of settings allowed us to improve this situation.

We assume that the all-to-all communication pattern puts high stress on the HCA which, at some point, cannot keep up with processing outstanding requests. To rule out any software issues with DXNet first, we implemented a low-level "loopback" like test which uses the native part of Ibdxnet, only. The loopback test does not involve any dynamic message posting when sending data or data processing when receiving. Instead, a buffer equally to the size of the ORB is processed by Ibdxnet's send thread on every iteration and posted to every participating SQ. This ensures that all SQs are filled and are quickly refilled once at least one WR was processed. When receiving data on the SRQ, all buffers received are directly put back into the pool without processing

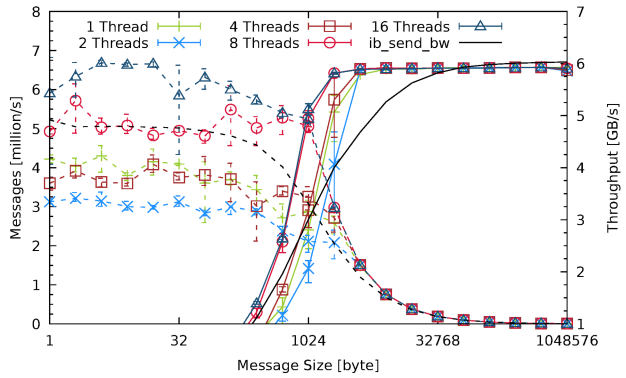

Figure 15: **DXNet**: 2 nodes, uni-directional throughput and message rate with multiple application send threads, increasing message size and 4 message handlers

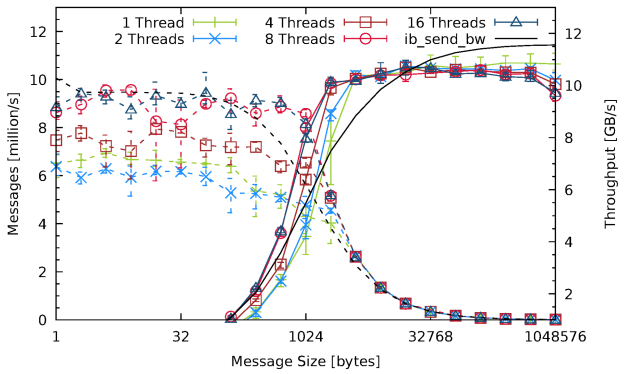

Figure 16: **DXNet**: 2 nodes, bi-directional throughput and message rate with multiple application send threads, increasing message size and 4 message handlers

and the SRQ is refilled. This ensures that no additional processing overhead is added for sending and receiving data. Thus, Ibdxnet's loopback test comes close to a perftool like benchmark. We executed the benchmark with 2, 4, 6 and 8 nodes which yielded aggregated throughputs of 11.7 GB/s, 21.7 GB/s, 28.3 GB/s and 34.0 GB/s.

These results are very close to the performance of the full DXNet stack but don't rule out all software related issues, yet. The overall aggregated bandwidth could still somehow be limited by Ibdxnet. Thus, we executed another benchmark which, first, executes all-to-all communication with up to 8 nodes, then, once bandwidth is saturated, switching to a ring formation for communication without restarting the benchmark (every node sends to its successor determined by NID, only).

Once the nodes switch the communication pattern during execution, the per node aggregated bandwidth increases very quickly and reaches a maximum aggregated bandwidth of approx. $(11.7/2 \times num\_nodes)$ GB/s independent of the number of nodes used. This rules out total bandwidth limitations for software and hardware. Furthermore, we can now rule out any performance issues in DXNet or even ibverbs with connection management (e.g. too many QPs allocated).

This leads to the assumption that the HCA cannot keep up with processing outstanding WRQs when SQs are under high load (always filled with WRQs). With more than 3 SQs per node, the total bandwidth drops noticably. Similar results with other systems further support this assumption (§9.3.4 and 9.4.4).

9.2.5 Uni-directional Throughput Multi-threaded

Figure 15 shows the uni-directional benchmark executed with 4 MHs and 1 to 16 send threads. For 1 to 4 send threads throughput saturates at 5.9 GB/s at either 4 kb or 8 kb messages. For 256 byte to 8 kb, using one thread yields better throughput than two or sometimes four threads. However, running the benchmark with 8 and 16 send threads increases overall throughput for all messages greater 32 byte significantly with saturation starting at 2 kb message size. DXNet's pipeline benefits from the many threads posting messages to the ORB concurrently. This results in greater aggregation of multiple messages and allows higher buffer utilization for the underlaying transport.

DXNet also increases message throughput on small message sizes up to 512 byte. from approx. 4.0 mmps up to 6.7 mmps for 16 send threads. Again, performance is slightly worse with two and four compared to a single thread.

Furthermore, DXNet even surpasses the baseline performance of *ib_send_bw* when using multiple send threads. However, the peak performance cannot be improved further which shows the current limit of DXNet for this benchmark and the hardware used.

9.2.6 Bi-directional Throughput Multi-threaded

Figure 16 shows the bi-directional benchmark executed with 4 MHs and 1 to 16 send threads. With more than one send thread, the aggregated throughput peaks at approx. 10.4 and 10.7 GB/s with messages sizes of 2 and 4 kb. DXNet delivers higher throughputs for all medium and small messages with increasing send thread count. The baseline performance of *ib_send_bw* is reached on small message sizes and even surpassed with medium sized messages up to 16 kb. The peak throughput is not reached showing DXNet's current limit with the used hardware.

The overall performance with 8 and 16 send threads don't differ noticeably which indicates saturation of DXNet's processing pipeline. For small messages (less than 512 byte), the message rates also increase with increasing send thread count. Again, saturation starts with 8 send threads with a message rate of approx. 8.6 to 10.2 mmps.

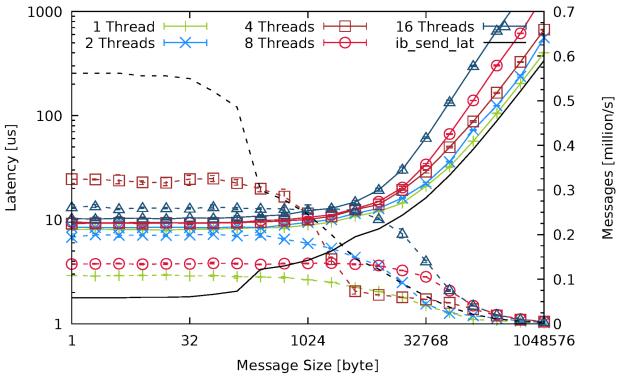

Figure 17: **DXNet**: 2 nodes, uni-directional avg. RTT and message rate with multiple application send threads, increasing message size and 4 message handlers

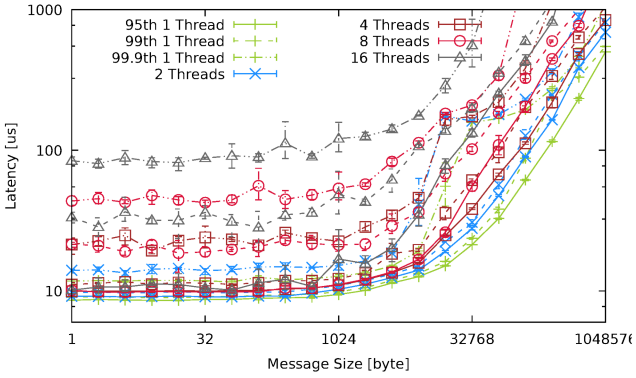

Figure 18: **DXNet**: 2 nodes, uni-directional 95th, 99th and 99.9th percentile RTT and message rate with multiple application send threads, increasing message size and 4 message handlers

DXNet is capable of handling a multi-threaded environment under high load with CPU over-provisioning and still delivers high throughput. Especially for small messages, DXNet's pipeline even benefits from the highly concurrent activity by aggregating many messages. This results in higher buffer utilization and, for the user, higher overall throughput.

9.2.7 Uni-directional Latency Multi-threaded

Figure 17 depicts the avg. RTT and message rate of the uni-directional latency benchmark with up to 16 send threads and 4 MHs. The 95th, 99th and 99.9th percentiles are depicated in figure 18. DXNet keeps a very stable avg. RTT of 8.1 µs for message sizes of 1 to 512 bytes with one send thread. Using two send threads, this value just slightly increases. With four or more send threads the avg. RTT increases to approx 9.3 µs. When the total number of threads, which includes

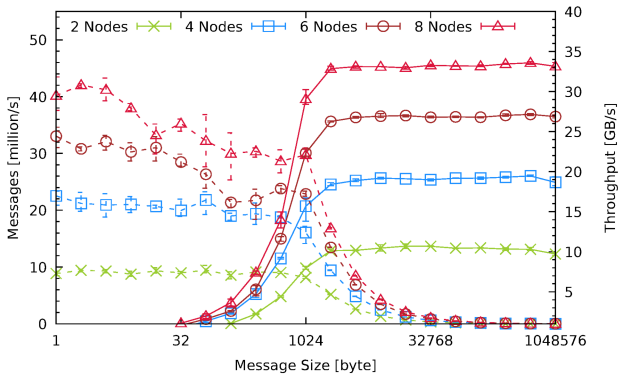

Figure 19: **DXNet**: 2 to 8 nodes, all-to-all aggregated send throughput and message rate with 16 application send threads, increasing message size and 4 message handlers

DXNet's internal threads, MH and send threads, exceed the core count of the CPU, DXNet switches to different parking strategies for the different thread types which slightly increase latency but greatly reduce overall CPU load (§7.2.5).

The message rate can be increased up to 0.33 mmps with up to 4 send threads as, practically, every send thread can use a free MH out of the 4 available. With 8 and 16 send threads, the MHs on the remote must be shared and DXNet's over-provisioning is active which reduces the overall throughput. The percentiles shown in figure 18 reflect this sitution very well and increase noticeably.

With a single thread, as already discussed in 9.2.3, the difference of the avg. (7.8 to 8.3 µs) and 99.9th percentile (11.8 to 12.7 µs) RTT for message sizes less than 1 kb is approx. 4 to 5 µs. When doubling the send thread count, the 99.9th percentiles roughly double as well. When over-provisioning the CPU, we cannot avoid the higher than usual RTT caused by the increasing amount of messages getting posted.

9.2.8 All-to-all Throughput with up to 8 Nodes Multi-threaded

Figure 19 shows the results of the all-to-all benchmark with up to 8 nodes, 16 application send threads and 4 message handlers. Compared to the single threaded results (§9.2.4), DXNet achieves slightly higher throughputs for all node counts: for two nodes, throughput saturates at 4 kb message size and peaks at 10.7 GB/s; for 4 nodes, throughput saturates at 4 kb message size and peaks at 19.5 GB/s; for 6 nodes, throughput saturates at 2 kb message size and peaks at 27.0 GB/s; for 8 nodes, throughput saturates at 2 kb message size and peaks at 33.6 GB/s. However, the message rate is improved significantly for small messages up to 64 byte with 8.4 to 10.3 mmps, 18.9 to 21.1 mmps, 27.6 to 31.4 mmps and 33.2 to 43.4 mmps for 2 to 8 nodes.

These results show that DXNet delivers high throughputs

and message rates under high loads with increasing node and thread count. Small messages profit significantly through better aggregation and buffer utilization.

9.2.9 Summary Results

This section briefly summerizes the most important results and numbers of the previous benchmarks. All values are considered "up to" and show the possible peak performance in the given benchmark.

Single-threaded:

- **Uni-directional throughput** One MH: saturation with 16 kb messages, peak throughput at 5.9 GB/s; Two MHs: saturation with 4 kb messages, peak throughput at 5.9 GB/s; Peak message rate of 4.0 to 4.5 mmps for small messages up to 64 bytes
- **Bi-directional throughput** Saturation with 8 kb messages at 10.4 GB/s with one MH; Peak message rate of 6.0 to 6.9 mmps for small messages up to 64 bytes
- **Uni-directional latency** up to 512 byte messages: avg. 7.8 to 8.3 µs, 95th percentile of 8.5 to 8.9 µs, 99th percentile of 8.9 to 9.2, 99.9th percentile of 11.8 to 12.7 µs; Peak message rate of 0.1 mmps.
- **All-to-all nodes** With 8 nodes: Total aggregated peak throughput of 32.4 GB/s, saturation with 16 kb message size; Peak message rate of 25.6 mmps for small messages up to 64 bytes.

Multi-threaded: Overall, DXNet benefits from higher message aggregation through multiple outstanding messages in the ORB posted concurrently by many threads.

- **Uni-directional throughput** Saturation at 5.9 GB/s at 4 kb message size
- **Bi-directional throughput** Overall improved throughput for many message sizes, saturation at 10.7 GB/s with 4 kb message size, message rate of 8.6 to 10.2 mmps for small messages up to 64 bytes
- **Uni-directional latency** Slightly higher latencies than single threaded as long as enough MHs serve available send threads. Message rate can be increased with additional send threads but at the cost of increasing avg. latency. The 99.9th percentiles roughly double when doubling the number of send threads.
- **All-to-all nodes**: With up to 8 nodes, 33.6 GB/s peak throughput, saturation at 2 kb message size, 33.2 to 43.4 mmps for up to 64 byte messages

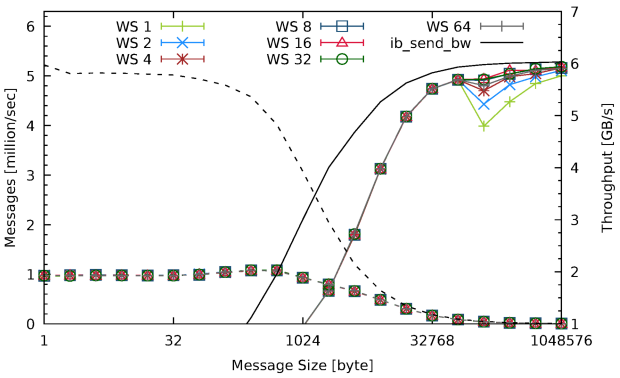

Figure 20: **FastMPJ**: 2 nodes, uni-directional throughput and message rate with increasing message and window size

9.3 FastMPJ

This section describes the results of the benchmarks executed with FastMPJ and compares them to the results of DXNet presented in the previous sections. We used FastMPJ 1.0_7 with the device *ibvdev* to run the benchmarks on InfiniBand hardware. The *osu* benchmarks of MVAPICH2 were ported to Java (§9.1) and used for all following experiments. Since FastMPJ does not support multithreading in a single process, all benchmarks were executed single threaded and compared to the single threaded results of DXNet, only.

9.3.1 Uni-directional Throughput

Figure 20 shows the results of executing the uni-directional benchmark with two nodes with increasing message size. Furthermore, the benchmark was executed with increasing WS to ensure bandwidth saturation. As expected, throughput increases with increasing message size and bandwidth saturation starts at a medium message size of 64k with approx. 5.7 GB/s. The actual peak throughput is reached with large 512k message for a WS of 64 with 5.9 GB/s.

For small message sizes up to 512 byte and independent of the WS, FastMPJ achieves a message rate of approx. 1.0 mmps. Furthermore, the results show that the WS doesn't matter for message sizes up to 64 KB. For 128 KB to 1 MB, FastMPJ profits from explicit aggregation with increasing WS. This indicates that ibvdev might include some message aggregation mechanism.

Compared to the baseline performance of *ib_send_bw*, FastMPJ's performance is always inferior to it with a peak performance of 5.9 GB/s close to *ib_send_bw*'s with 6.0 GB/s.

Compared to the results of DXNet (§9.2.1), DXNet's throughput saturates and peaks earlier at a message size of 16 kb with 5.9 GB/s. However, if using one MH, throughput drops for larger messages down to 5.4 GB/s due to increased message processing time (de-serialization). However, such a

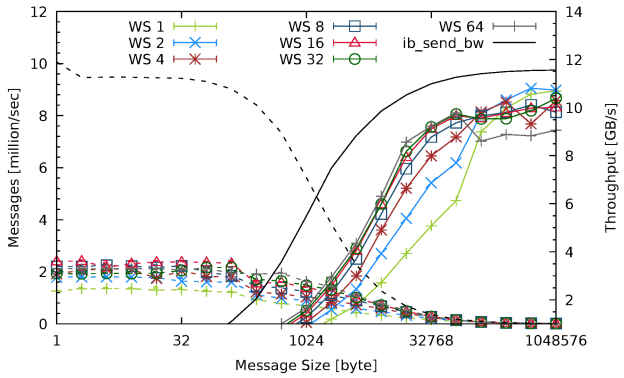

Figure 21: **FastMPJ**: 2 nodes, bi-directional (aggregated) throughput and message rate with increasing message and window size

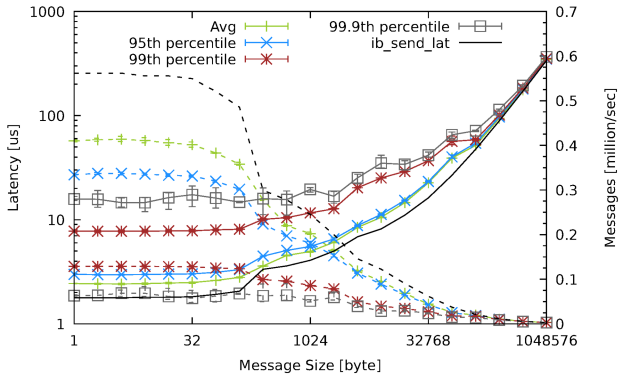

Figure 22: **FastMPJ**: 2 nodes, uni-directional latency and message rate with increasing message and window size

mechanism is absent from FastMPJ and DXNet can further improve performance by using two MHs. With two MHs, DXNet's throughput peaks even earlier at 5.9 GB/s with 4 kb message size. For small messages of up to 64 bytes, DXNet achieves 4.0 to 4.5 mmps compared to FastMPJ with 1.0 mmps.

9.3.2 Bi-directional Throughput

The results of the bi-directional benchmark are depicted in figure 21. Again, throughput increases with increasing message size peaking at 10.8 GB/s with WS 2 and large 512 kb messages. However, when handling messages of 128 kb and greater, throughput peaks at approx 10.2 GB/s for the WSs 4 to 32 and saturation varies depending on the WS. For WSs 4 to 32, throughput is saturated with 64 kb messages, for WSs 1 and 2 at 512 kb. Starting at 128 kb message size, WSs of 1 and 2 achieve slightly better results than the greater WSs. Especially WS 64 drops significantly with message sizes of 128 kb and greater. However, for message sizes of 64 kb to 512 kb, FastMPJ profits from explicit aggregation.

Compared to the uni-directional results (§9.3.1), FastMPJ does profit to some degree from explicit aggregation for small messages with 1 to 128 bytes. WS 1 to 16 allow higher message throughputs with WS 16 as an optimal value peaking at approx. 2.4 mmps for 1 to 128 byte messages. Greater WSs degrade message throughput significantly. However, this does not apply to message sizes of 256 bytes where greater explcit aggregation does always increase message throughput.

Compared to the baseline performance of *ib_send_bw*, FastMPJ's performance is again always inferior to it with a difference in peak performance of 0.7 GB/sec (10.8 GB/s to 11.5 GB/s).

When comparing to DXNet's results (§9.2.2), the throughputs are nearly equal with 10.7 GB/s also at 512 kb message size. However, DXNet outperforms FastMPJ for medium sized messages by reaching a peak throughput of 10.4 GB/s for just 8 kb messages. Even with a WS of 64, FastMPJ can only achieve 6.3 GB/s aggregated throughput here. For small messages of up to 64 bytes, DXNet clearly outperforms FastMPJ with 6 to 7.2 mmops compared to 1.9 to 2.1 mmops with a WS of 16.

9.3.3 Uni-directional Latency

The results of the latency benchmark are depicted in figure 22. FastMPJ achieves a very low average RTT of 2.4 µs for up to 16 byte messages. This just slightly increases to 2.8 µs for up to 128 byte messages and to 4.5 µs for up to 512 byte messages. 3 µs RTT is achieved by the 95th percentile for up to 64 byte which slightly increases to 5 µs for up to 512 byte messages. Message sizes up to 16 bytes achieve a 7.7 µs RTT for the 99th percentile with up to 10 µs for up to 512 byte messages. For the 99.9th percentile, messages sizes up to 16 byte fluctuate slightly with a RTT of 14.5 to 15.5 µs. This continues for 32 byte to 2 kb with a low of 16.3 µs and a high of 19.5 µs. The average message rate peaks at approx. 0.41 mmps for up to 16 byte messages.

Compared to the baseline performance of *ib_send_lat*, FastMPJ's average RTT comes close to its 1.8 µs and closes that gap slightly further starting with 256 byte message size.

Comparing the avg. RTT and 95th percentile to DXNet's results (§9.2.3), FastMPJ outperforms DXNet by a up to four times lower RTT. This is also reflected by the message rate of 0.41 mmps for FastMPJ and 0.1 mmps for DXNet. The breakdown given Section 9.2.3 explains the rather high RTTs and the amount of processing time spent by DXNet on major sections of the pipeline. However, even DXNet's avg. RTT for message sizes up to 512 byte is higher than FastMPJ's, DXNet achieves lower 99th (8.9 to 9.2 µs) and 99.9th percentile (11.8 to 12.7 µs) than FastMPJ.

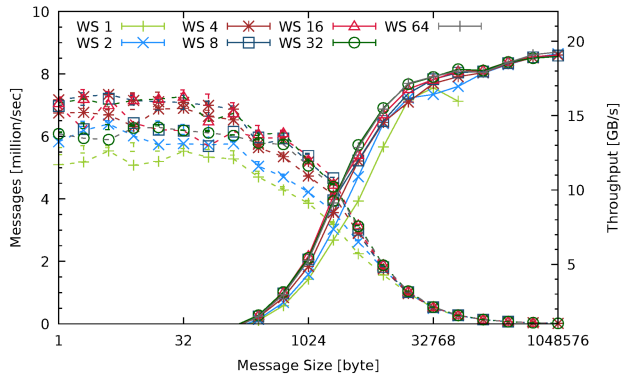

Figure 23: **FastMPJ**: 4 nodes, all-to-all (aggregated) throughput and message rate with increasing message and window size

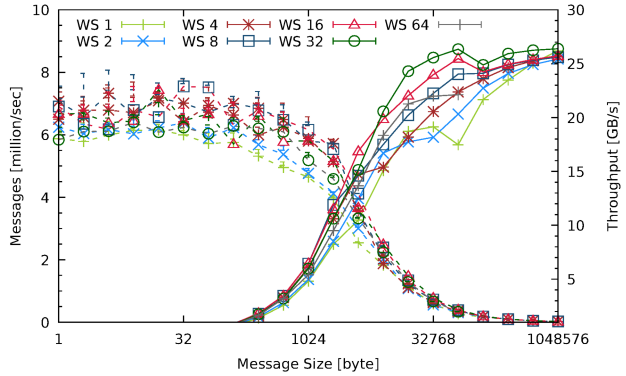

Figure 24: **FastMPJ**: 6 nodes, all-to-all (aggregated) throughput and message rate with increasing message and window size

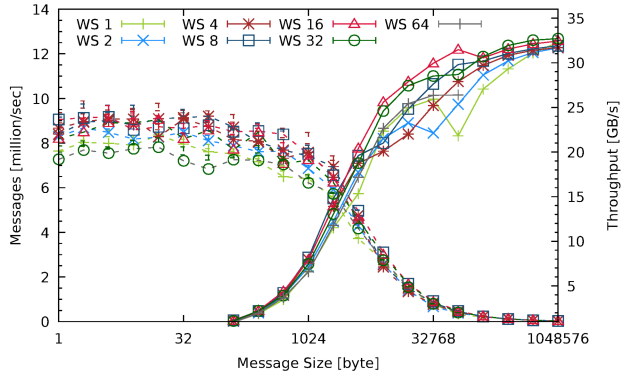

Figure 25: **FastMPJ**: 8 nodes, all-to-all (aggregated) throughput and message rate with increasing message and window size

9.3.4 All-to-all with Increasing Node Count

Figures 23, 24 and 25 show the aggregated send throughputs and message rates of the all-to-all benchmark running on 4, 6 and 8 nodes. The results for 2 nodes were already discussed in 9.3.2 and are depicted in figure 21. The results for WS 64 and messages greater than 64 kb are absent because FastMPJ hangs (no error output) on message sizes greater than 64 kb with WS 64. We couldn't resolve this by re-running the benchmark several times and with different configuration parameters like increasing buffer sizes.

FastMPJ scales well with increasing node count on all-to-all communication with the following peak throughputs: 10.8 GB/s with WS 2 and 512 kb messages on 2 nodes, 19.2 GB/s with WS 64 and 1 MB messages on 4 nodes, 26.3 GB/s with WS 32 and 1 MB messages on 6 nodes, 32.7 GB/s with WS 32 and 1 MB messages on 8 nodes. This results in per node send throughputs of 5.1 GB/s, 4.8 GB/s, 4.38 GB/s and 4.08 GB/s. The gradually decreasing per node throughput seems to be a non software related issue as explained in Section 9.2.4. For small messages up to 64 bytes, FastMPJ achieves the following peak message rates: 2.4 mmps for WS 16 on 2 nodes, 7.2 mmps for WS 16 on 4 nodes, 7.6 mmps for WS 16 on 6 nodes and 9.5 mmps for WS 8 on 8 nodes.

DXNet also reaches peak throughputs close to FastMPJ's (§9.2.4) on all node counts. However, DXNet saturates bandwidth very early with just 8 kb and 16 kb message sizes. Furthermore, DXNet outperforms FastMPJ's message rates for small messages on all node counts by up to three times (7.0 mmps, 15.0 mmps, 21.1 mmps and 27.3 mmps).

9.3.5 Summary Results

This section briefly summerizes the most important results and key numbers of the previous benchmarks. All values are considered "up to" and show the possible peak performance in the given benchmark and are single-threaded, only. All results benefit from explicit aggregation using the WS.

- **Uni-directional throughput** Saturation at 64 kb message size with 5.7 GB/s; Peak throughput at 512 kb message size with 5.9 GB/s; 1.0 mmps for message sizes up to 64 byte
- **Bi-directional throughput** Saturation at 64 kb message size with 10.8 GB/s; 2.4 mmps for message sizes up to 128 byte
- **Uni-directional latency** For up to 512 byte messages: avg. RTT of 2.4 to 4.5 µs, 95th percentile of 3 to 5 µs; 99th percentile of 7.7 to 10 µs; 99.9th percentile of 16.3 to 19.5 µs
- **All-to-all nodes** With 8 nodes: Total aggregated peak throughput of 32.7 GB/s, saturation with 1 mb message size; Peak message rate of 9.5 mmps for small messages up to 64 byte

Compared to DXNet's single threaded results, it outperforms FastMPJ on small messages with a up to 4 times

higher message rate on both un- und bi-directional benchmarks. However, FastMPJ achieves a lower average and 95th percentile latency on the uni-directional latency benchmark. But, even with a more complicated and dynamic pipeline, DXNet achieves lower 99th and 99.9th percentile than FastMPJ demonstrating high stability. On all-to-all communication with up to 8 nodes, DXNet reaches similar throughputs to FastMPJ's for large messages but outperforms FastMPJ's message rate by up to three times for small messages. **DXNet is always better for small messages**.

9.4 MVAPICH2

This section describes the results of the benchmarks executed with MVAPICH2 and compares them to the results of DXNet. All *osu* benchmarks (§9.1) were executed with MVAPICH2-2.3. Since MVAPICH2 supports MPI calls with multiple threads of the same process, some benchmarks were executed single and multi-threaded. We set the following environmental variables for optimal performance and comparability:

- MV2_DEFAULT_MAX_SEND_WQE=128
- MV2_DEFAULT_MAX_RECV_WQE=128
- MV2_SRQ_SIZE=1024
- MV2_USE_SRQ=1
- MV2_ENABLE_AFFINITY=1

Additionally for the multi-threaded benchmarks, the following environmental variables were set:

- MV2_CPU_BINDING_POLICY=hybrid
- MV2_THREADS_PER_PROCESS=X (where X equals the number of threads we used when executing the benchmark)
- MV2_HYBRID_BINDING_POLICY=linear

9.4.1 Uni-directional Throughput

The results of the uni-directional single threaded benchmark are depicted in figure 26. The overall throughput increases with increasing message size, peaking at 5.9 GB/s with multiple WS on large messages: 512 kb with 64 WS, 256 kb with 32 WS, 512 KB with 8 WS, 512 kb with 4 WS and 1 MB with 2 WS. Bandwidth saturation starts at approx. 64 kb to 128 kb for WSs of 16 or greater. This also applies to smaller messages with up to 64 bytes. Reaching a peak of 4.0 mmps is only possible with WS 64. If send calls are not batched explicitly, message rates are rather low (0.45 mmps for WS 1).

Compared to the baseline performance of *ib_send_bw*, MVAPICH2's peak performance is approx. 1.0 mmps less

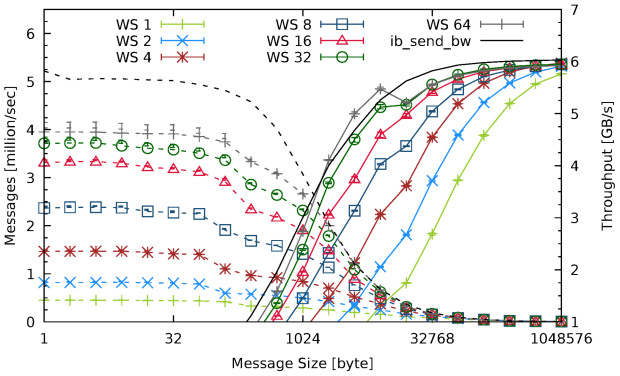

Figure 26: **MVAPICH2**: 2 nodes, uni-directional throughput and message rate, single threaded with increasing message and window size

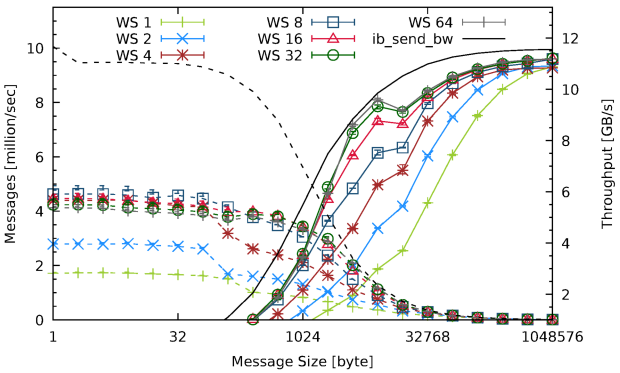

Figure 27: **MVAPICH2**: 2 nodes, bi-directional throughput and message rate, single threaded with increasing message and window size

for small messages. With increasing message size, on a WS of 64, the performance comes close to the baseline and even exceeds it for 2 kb to 8 kb messages. MVAPICH2 peaks very close to the baseline's peak performance of 6.0 GB/s.

DXNet achieves very similar results (§9.2.1) compared to MVAPICH2 but without relying on explicit aggregation. DXNet's throughput saturates and peaks earlier at a message size of 16 kb with 5.9 GB/s. However, if using one MH, throughput drops for larger messages down to 5.4 GB/s due to increased message processing time (de-serialization). As already explained in Section 9.3.1, this can be resolved by using two MHs. For small messages of up to 64 bytes, DXNet achieves an equal to slightly higher message rate of 4.0 to 4.5 mmps.

9.4.2 Bi-directional Throughput

The results of the bi-directional single threaded benchmark are depicted in figure 27. Overall throughput increases with

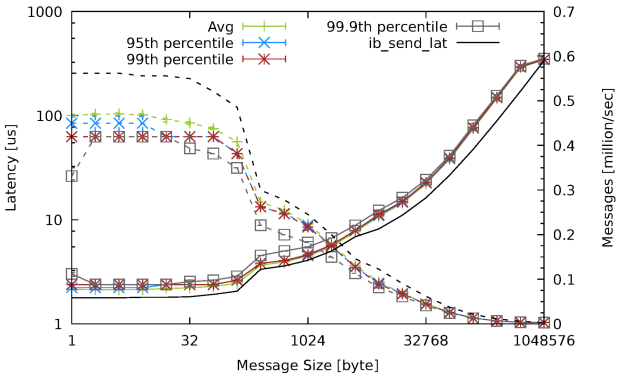

Figure 28: **MVAPICH2**: 2 nodes, uni-directional latency and message rate, single threaded with increasing message size

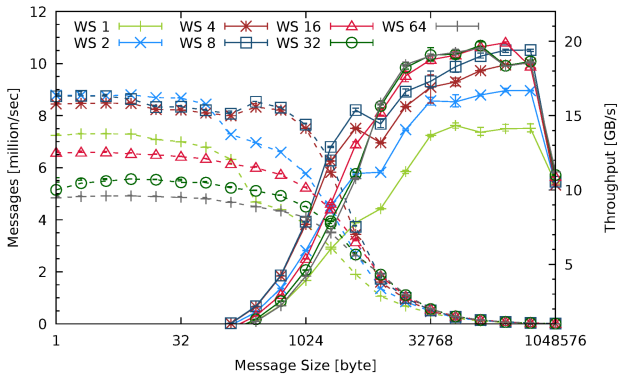

Figure 29: **MVAPICH2**: 4 nodes, aggregated send throughputs and message rates, single threaded with increasing message size

message size and, like on the uni-directional benchmark, benefits a lot from greater WSs. The aggregated throughput peaks at 11.1 GB/s with 512 kb messages on multiple WSs. Throughputs for 128 byte to 512 kb message sizes benefit from explicit aggregation. The message rate for small messages up to 64 bytes do not always profit from explicit aggregation. Message rate increases with WS 1 to 8 and peaks at 4.7 mmps with WS 8. However, greater WS degrade the message rate slightly compared to the optimal case.

Compared to the baseline performance of *ib_send_bw*, MVAPICH2's peak performance for small messages is approx. half of *ib_send_bw*'s 9.5 mmps. With increasing message size, the throughput of MVAPICH2 comes close *ib_send_bw*'s with WS 64 and 32 for 4 and 8 kb messages, only. Peak throughput for large messages comes close to *ib_send_bw*'s 11.5 GB/s.

Compared to DXNet's results (§9.2.2), the aggregated throughput is slightly higher than DXNet's (10.7 GB/s). However, DXNet outperforms MVAPICH2 for medium sized messages by reaching a peak throughput of 10.4 GB/s compared to 9.5 GB/s (on WS 64) for just 8 kb messages. Furthermore, DXNet offers a higher message rate of 6 to 7.2 mmps on small messages up to 64 bytes. DXNet achieves overall higher performance without relying on explicit message aggregation.

9.4.3 Uni-directional Latency

Figure 28 shows the results of the uni-directional single threaded latency benchmark. MVAPICH2 achieves a very low average RTT of 2.1 to 2.4 µs for up to 64 byte messages and up to 3.9 µs for up to 512 byte messages. The 95th, 99th and 99.9th percentile are just slightly higher than the average RTT with 2.2 to 4.0 µs for the 95th, 2.4 to 4.0 µs for the 99th and 2.4 to 5.0 µs for the 99.9th (for up to 512 byte message size). This results in an average message rate of 0.43 to 0.47 mmps for up to 64 byte messages and 0.25 to 0.40 for 128 to 512 byte messages.

Compared to the baseline performance of *ib_send_lat*, MVAPICH2's average, 95h, 99th, and 99.9th percentile RTT are very close to the baseline. With a minimum of 2.1 µs for the average latency and maximum of 5.0 µs for the 99.9th percentile on small messages, MVAPICH2 shows that its overall overhead is very low.

Compared to DXNet's results (§13), MVAPICH2 achieves an overall lower latency. DXNet's average with 7.8 to 8.3 µs is nearly four times higher. The 95h (8.5 to 8.9 µs), 99th (8.9 to 9.2 µs) and 99.9th percentile (11.8 to 12.7 µs) are also at least two to three times higher. MVAPICH2 implements a very thin layer of abstraction, only. Application threads issuing MPI calls, are pinned to cores and are directly calling ibverbs functions after passing through these few layers of abstraction. DXNet however implements multiple pipeline stages with de/-serialization and multiple (JNI) context/thread switches. Naturally, data passing through such a long pipeline takes longer to process which impacts overall latency. However, DXNet traded latency for multithreading support and performance as well as efficient handling of small messages.

9.4.4 All-to-all Throughput with up to 8 Nodes

Figures 29, 30 and 31 show the results of executing the all-to-all benchmark with 4, 6 and 8 nodes. The results for 2 nodes are depicted in figure 27.

MVAPICH2 achieves a peak throughput of 19.5 GB/s with 128 kb messages on WSs 16, 32 and 64 and starts at approx 32 kb message size. WS 8 gets close to the peak throughput as well but the remaining WSs peak lower for messages greater than 32 kb. Minor fluctuations appear for WS 1 to 16 for 1 kb to 16 kb messages. For small messages of up to 512 byte, the smaller the WS the better the performance.

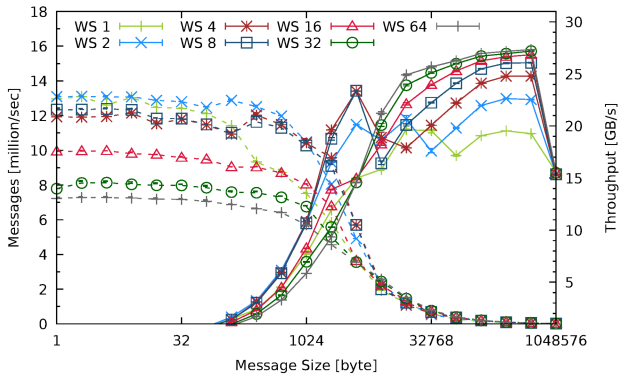

Figure 30: **MVAPICH2**: 6 nodes, aggregated send throughputs and message rates, single threaded with increasing message size

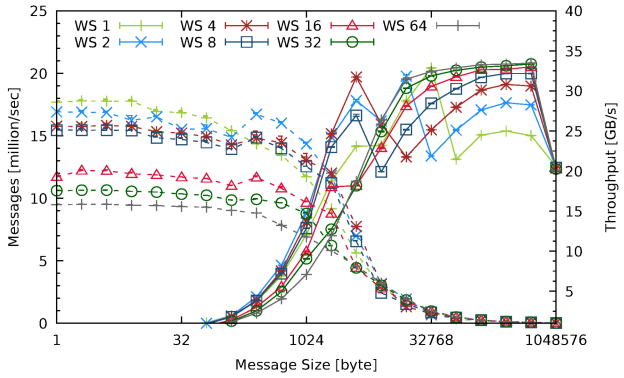

Figure 31: **MVAPICH2**: 8 nodes, aggregated send throughputs and message rates, single threaded with increasing message size

With WS 2, a message rate 8.4 to 8.8 mmps for up to 64 byte messages is achieved and 6.6 to 8.8 mmps for up to 512 byte.

Running the benchmark with 6 nodes, MVAPICH2 hits a peak throughput of 27.3 GB/s with 512 kb messages on WSs 16, 32 and 64. Saturation starts with a message size of approx. 64 to 128 kb depending on the WS. For 1 kb to 32 kb messages, the fluctuations increased compared to executing the benchmark with 4 nodes. Again, message rate is degraded when using large WS for small messages. An optimal message rate of 11.9 to 13.1 is achieved with WS 2 for up to 64 byte messages.

With 8 nodes, the benchmark peaks at 33.3 GB/s with 64 kb messages on a WS of 64. Again, WS does matter for large messages as well with WS 16, 32 and 64 reaching the peak throughput and starting saturation at approx. 128 kb message size. The remaining WSs peak significantly lower. The fluctuations for mid range messages sizes of 1 kb to 64 kb increased further compared to 6 nodes. Most notable, the performance with 4 kb messages and WS 4 is nearly 10 GB/s

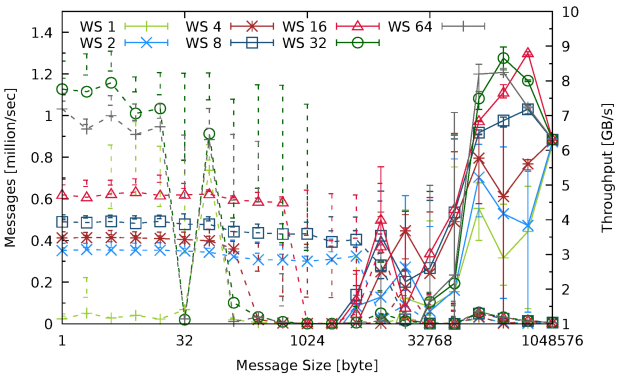

Figure 32: **MVAPICH2**: 2 nodes, bi-directional throughput and message rate, multi-threaded with one send and one recv thread with increasing message and window size

better than 4 kb with WS 64. With up to 64 byte messages, a message rate of 16.5 to 17.8 mmps is achieved. For up to 512 byte messages, the message rate varies with 13.5 to 17.8 mmps. As with the previous node counts, a smaller WS increases the message rate significantly while larger WSs degrade performance by a factor of two.

MVAPICH2 has the same "scalability issues" as DXNet (§9.2.4) and FastMPJ (§9.3.4). The maximum achievable bandwidth matches what was determined with the other systems. With the same results on three different systems, it's very unlikely that this is some kind of software issue like a bug or bad implementation but most likely a hardware limitation. So far, we haven't seen this issue discussed in any other publication and think it is noteworthy to know what the hardware is currently capable of.

Compared to DXNet (§9.2.4), MVAPICH2 reaches slightly higher peak throughputs for large messages. However, this peak as well as saturation is reached later at 32 to 512 kb messages compared to DXNet with approx. 16 kb. The fluctuations for mid range size messages cannot be compared as DXNet does not rely on explicit aggregation. For small messages up to 64 byte, DXNet achieves significantly higher message rates, with peaks at 7.0 mmps, 15.0 mmps, 21.1 mmps and 27.3 mmps for 2 to 8 nodes, compared to MVAPICH2.

9.4.5 Bi-directional Throughput Multi-threaded

Figure 32 shows the results of the bi-directional multi-threaded benchmark with two threads (on each node): a separate thread for sending and receiving each. In our case, this is the simplest multi-threading configuration to utilize more than one thread for MPI calls. The plot shows highly fluctuating results of the three runs executed as well as overall low throughput compared to the single threaded results (§9.4.2). Throughput peaks at 8.8 GB/s with a message size of 512 kb

for WS 16. A message rate of 0.78 to 1.19 mmps is reached for for up to 64 byte messages for WS 32.

We tried varying the configuration values (e.g. queue sizes, buffer sizes, buffer counts) but could not find configuration parameters that yielded significantly better, especially less fluctuating, results. Furthermore, the benchmarks could not be finished with sending 100,000,000 messages. When using *MPI_THREAD_MULTIPLE*, the memory consumption increases continuously and exhausts the total memory available on our machine (64 GB). We reduced the number of messages to 1,000,000 which still consumes approx. 20% of the total main memory but at least executes and finishes within a reasonable time. This does not happen with the widely used *MPI_THREAD_SINGLE* mode.

MVAPICH2 implements multi-threading support using a single global lock for various MPI calls which includes *MPI_Isend* and *MPI_Irecv* used in the benchmark. This fulfils the requirements described in the MPI standard and avoids a complex architecture with lock-free data structures. However, a single global lock reduces concurrency significantly and does not scale well with increasing thread count [12]. This effect impacts performance less on applications with short bursts and low thread count. However, for multi-threaded applications under high load, a single-threaded approach with one dedicated thread driving the network decoupled from the application threads, might be a better solution. Data between application threads and the network thread can be exchanged using data structures such as buffers, queues or pools like provided by DXNet.

MVAPICH2's implementation of multi-threading does not allow to improve performance by increasing the send or receive thread counts. Thus, further multi-threaded experiments using MVAPICH2 are not reasonable.

9.4.6 Summary Results

This section briefly summerizes the most important results and numbers of the previous benchmarks. All values are considered "up to" and show the possible peak performance in the given benchmark. **Single-threaded**:

- **Uni-directional throughput** Saturation with 64 kb to 128 kb message size, peak at 5.9 GB/s; Message rate of 4.0 mmps for up 64 byte messages
- **Bi-directional throughput** Saturation at 512 kb message size, peak at 11.1 GB/s; Message rate of 4.7 mmps for up to 64 byte messages
- **Uni-directional latency** For up to 64 byte message size: 2.1 to 2.4 µs average latency and 2.4 to 5.0 µs for 99.9th percentile; 0.43 to 0.47 mmps message rate
- **All-to-all nodes** For 8 nodes: peak at 33.3 GB/s with 64 kb message size on WS 64, WS matters for large messages; Message rate of 16.5 to 17.8 mmps for up to 64 byte messages
- **Bi-directional throughput multi-threaded**: High fluctuations with low throughputs caused by global locking, 8.8 GB/s peak throughput at 512 kb message size; Message rate of 0.78 to 1.19 mmps for up to 64 byte messages

Compared to DXNet, the uni-directional results are similar but DXNet does not require explicit message aggregation to deliver high throughput. On bi-directional communication, MVAPICH2 achieves a slightly higher aggregated peak throughput than DXNet but DXNet performs better by approx 0.9 GB/s on medium sized messages. DXNet outperforms MVAPICH2 on small messages with a up to 1.8 times higher message rate. But, MVAPICH2 clearly outperforms DXNet on the uni-directional latency benchmark with an overall lower average, 95th, 99th and 99.9th percentile latency. On all-to-all communication with up to 8 nodes, MVAPICH2 reaches slightly higher peak throughputs for large messages but DXNet reaches its saturation earlier and performs significantly better on small message sizes up to 64 bytes.

The low multi-threading performance of MVAPICH2 cannot be compared to DXNet's due to the following reasons: First, MVAPICH2 implements synchronization using a global lock which is the most simplest but very often least performant method to ensure thread safety. Second, MVAPICH2, like many other MPI implementations, typically create multiple processes (one process per core) to enable concurrency on a single processor socket. However, as already discussed in related work (§3), this programming model is not suitable for all application domains, especially in big data applications.

DXNet is better for small messages and multi-threaded access like required in big-data applications.

10 Conclusions

We presented Ibdxnet, a transport for the Java messaging library DXNet which allows multi-threaded Java applications to benefit from low latency and high-throughput using InfiniBand hardware. DXnet provides transparent connection management, concurrency handling, message serialization and hides the transport which allows the application to switch from Ethernet to InfiniBand hardware transparently, if the hardware is available. Ibdxnet's native subsystem provides dynamic, scalable, concurrent and automatic connection management and the msgrc messaging engine implementation. The msgrc engine uses a dedicated send and receive thread and to drive RC QPs asynchronously which ensures scalability with many nodes. Load adaptive parking avoids high loads on idle but ensures low latency when

busy. SGEs are used to simplify buffer handling and increase buffer utilization when sending data provided by the higher level DXNet core. A carefully crafted architecture minimizes context switching between Java and the native space as well us exchanging data using shared memory buffers. The evaluation shows that DXNet with the Ibdxnet transport can keep up with FastMPJ and MVAPICH2 on single threaded applications and even exceed them in multi-threaded applications on high load applications. DXNet with Ibdxnet is capable of handling concurrent connections and data streams with up to 8 nodes. Furthermore, multi-threaded applications benefit significantly from the multi-threaded aware architecture.

The following topics are of interest for future research with DXnet and Ibdxnet:

- Experiments with more than 100 nodes on our university's cluster
- Evaluate DXNet with the key-value store DXRAM using the YCSB and compare it to RAMCloud
- Implementation and evaluation of a UD QP based transport engine
- Hybrid mode for DXNet: Analyze if applications benefit from using Ethernet and InfiniBand if both are available
- RDMA path: Boost performance for applications like key-value storages

References

[1] Github dxnet. `https://github.com/hhu-bsinfo/dxnet`.

[2] libvma github wiki. `https://github.com/Mellanox/libvma/wiki/Architecture`.

[3] Open mpi. `https://www.open-mpi.org/`.

[4] Openmp. `http://www.openmp.org/`, 1997.

[5] Apache spark rdma plugin. `https://community.mellanox.com/docs/DOC-3012`, October 2017.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, 2012.

[7] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and X. Li. mpijava: A java interface to mpi. `http://www.hpjava.org/mpiJava.html`, July 2002.

[8] K. Beineke, S. Nothaas, and M. Schöttner. Efficient messaging for java applications running in data centers. In *International Workshop on Advances in High-Performance Algorithms Middleware and Applications (in proceedings of CCGrid18)*, 2018.

[9] K. Beineke, S. Nothaas, and M. Schöttner. Scalable messaging for java-based cloud applications. In *14th International Conference on Networking and Services (ICNS)*, 2018.

[10] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8:1804–1815, Aug. 2015.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[12] H. V. Dang, S. Seo, A. Amer, and P. Balaji. Advanced thread synchronization for multithreaded mpi implementations. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 314–324, May 2017.

[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51:107–113, Jan. 2008.

[14] P. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Incremental page rank computation on evolving graphs. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 1094–1095, 2005.

[15] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, 2014. USENIX Association.

[16] S. Ekanayake, S. Kamburugamuve, and G. C. Fox. Spidal java: High performance data analytics with java and mpi on large multicore hpc clusters. pages 3:1–3:8, 2016.

[17] R. R. Exposito, S. Ramos, G. L. Taboada, J. Touriño, and R. Doallo. Fastmpj: a scalable and efficient java message-passing library. *Cluster Computing*, 17:1031–1050, Sept. 2014.

[18] R. R. Expósito, G. L. Taboada, J. Touriño, and R. Doallo. Design of scalable java message-passing communications over infiniband. *The Journal of Supercomputing*, pages 141–165, July 2012.

[19] M. P. Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[20] A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web*, WWW '05, pages 902–903, 2005.

[21] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47:98 – 115, 2015.

[22] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High performance rdma-based design of hdfs over infiniband. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 35:1–35:35, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[23] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless network functions. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '15, pages 49–54, New York, NY, USA, 2015. ACM.

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

[25] V. Kashyap. Ip over infiniband (ipoib) architecture. `https://www.ietf.org/rfc/rfc4392.txt`, April 2006.

[26] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[27] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, 2014. USENIX Association.

[28] X. Liu. Entity centric information retrieval. *SIGIR Forum*, 50:92–92, June 2016.

[29] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: The java unsafe api in the wild. *SIGPLAN Not.*, 50:695–710, Oct. 2015.

[30] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, number 12 in USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association.

[31] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500, May 2010.

[32] S. Narravula, A. Mamidala, A. Vishnu, G. Santhanaraman, and D. K. Panda. High performance mpi over iwarp: Early experiences. In *2007 International Conference on Parallel Processing (ICPP 2007)*, pages 46–46, Sept. 2007.

[33] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[34] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. sson, D. van der Spoel, B. Hess, and E. Lindahl. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29:845–854, 2013.

[35] A. Shafi, B. Carpenter, and M. Baker. Nested parallelism for multi-core hpc systems using java. In *Journal of Parallel and Distributed Computing*, pages 532–545, 2009.

[36] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller. Ucx: An open source framework for hpc network apis and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43, Aug 2015.

[37] P. Stuedi, B. Metzler, and A. Trivedi. jverbs: Ultra-low latency for data center applications. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 10:1–10:14, New York, NY, USA, 2013. ACM.

[38] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu. Rfp: When rpc is faster than server-bypass with rdma. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 1–15, New York, NY, USA, 2017. ACM.

[39] G. L. Taboada, J. Touriño, and R. Doallo. Java fast sockets: Enabling high-speed java communications

on high performance clusters. *Comput. Commun.*, 31:4049–4059, Nov. 2008.

[40] S. Thirugnanapandi, S. Kodali, N. Richards, T. Ellison, X. Meng, and I. Poddar. Transparent network acceleration for java-based workloads in the cloud. `https://www.ibm.com/developerworks/library/j-transparentaccel/`, January 2014.

[41] X. Wu, X. Zhu, G. Q. Wu, and W. Ding. Data mining with big data. *IEEE Transactions on Knowledge and Data Engineering*, 26:97–107, Jan. 2014.

[42] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59:56–65, Oct. 2016.

[43] P. Zhao, Y. Li, H. Xie, Z. Wu, Y. Xu, and J. C. Lui. Measuring and maximizing influence via random walk in social activity networks. pages 323–338, Mar. 2017.

Chapter 6

Further Benchmarks and Applications

This chapter depicts further contributions that are not the main focus of this thesis or part of DXRAM itself but the application and development environment.

6.1 Cluster Deployment Tool

With the development of a distributed system as well as evaluation of other (distributed) systems, fast and straightforward deployment becomes an important matter. However, many applications do not provide tools for automated deployment to aid in debugging and evaluation. Such a situation is further complicated when the environment to deploy to is not always the same, e.g., hardware requirements change for an evaluation. Typically, this is approached by creating small Bash or Python scripts to start and coordinate the desired distributed application on multiple servers. Often, however, these scripts are not reusable for other applications let alone different cluster environments (e.g., private cluster, HPC cluster with a job system, cloud environment). This results in very repetitive and time-consuming tasks of creating similar scripts with changes to adapt to the different application or environment. Thus, the author of this thesis put considerable effort into designing and implementing a solution to streamline these tedious and time-consuming deployment tasks.

The cluster deployment tool cdepl solves these issues by simplifying deployment of distributed applications while considering different types of cluster environments. cdepl is written entirely in Bash and in its core only uses commonly available and often by default installed Linux utilities making it easily extensible and portable to different cluster installments. cdepl is open source and available at GitHub [20]. Figure 6.1 depicts an overview of cdepl's architecture.

A cluster abstraction layer allows implementing support for different cluster types, currently supporting localhost, "simple" clusters (SSH with public-key auth), Microsoft Azure, the private cluster of the Operating Systems workgroup at the University of Düsseldorf and the HILBERT HPC environment of the "Centre for Information and Media Technology" (ZIM) at the University of Düsseldorf. This layer abstracts not only implementations sending simple SSH commands to remote nodes (e.g., simple cluster) but also cluster job management systems (e.g., PBS of HILBERT) to allow transparent interaction with the target cluster environment (e.g., server/environment setup and allocation).

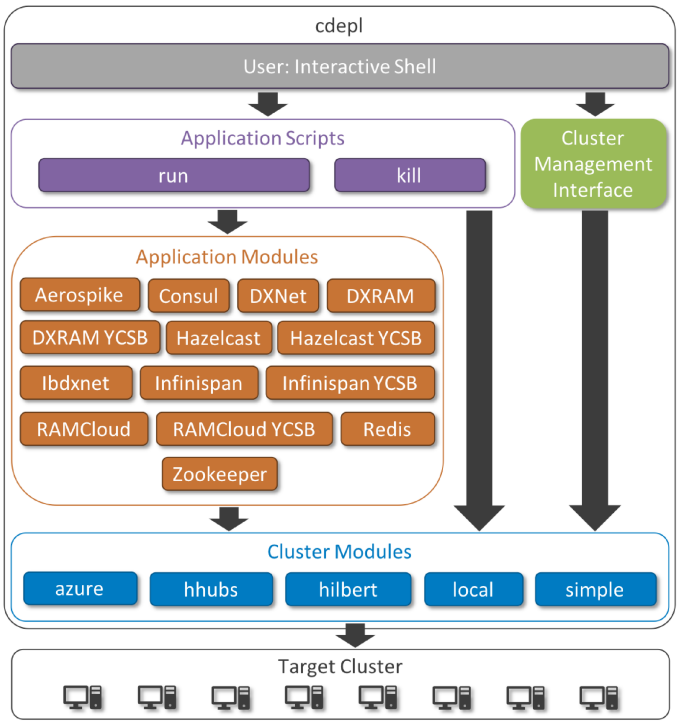

Figure 6.1: Overview of cdepl's layered architecture.

An application abstraction layer allows implementing different application modules with access to commonly used features from the cluster abstraction layer, e.g., running applications on remote servers or executing file system related commands. At the time of writing this thesis, modules for the following applications are available in cdepl: Aerospike, Consul, DXNet, DXRAM, DXRAM YCSB, Hazelcast, Hazelcast YCSB, Ibdxnet, Infinispan, Infinispan YCSB, RAMCloud, RAMCloud YCSB, Redis, Zookeeper.

The deployment process is further abstracted and simplified by a thin cluster management interface that allows cluster and node configuration, node allocation and cleanup. cdepl abstracts application configuration, start as well as coordination by an application script layer providing "run" and "kill" scripts with application dependent interfaces to quickly start and cleanup even large deployments. An interactive shell allows the user to use the described interfaces to execute the three simple steps: cluster resource allocation, application deployment, and cleanup. This environment was explicitly designed to allow fast re-deployments for testing and debugging sessions.

cdepl was designed and developed with the majority of contributions by Stefan Nothaas. Dr. Kevin Beineke implemented the cluster module to support deployment to Microsoft's Azure cloud and helped improve cdepl with various bugfixes. Fabian Ruhland implemented the application modules for Aerospike and Redis as well as helped to fix various bugs. Filip Krakowski implemented the Consul application module and also provided various bugfixes. Burak Akguel and Christian Gesse contributed further bugfixes and minor features.

6.2 Yahoo! Cloud Serving Benchmark Client

The Yahoo! Cloud Serving Benchmark [25] is a framework to evaluate the performance of different key-value storages/databases/caches. The benchmark provides a flexible and configurable workload interface which allows specifying workloads with different application focuses, e.g., read-heavy workloads or workloads with small objects. Clients have to provide implementations of the basic CRUD operations and an optional scan-operation.

The author put considerable effort into optimizing the DXRAM client implementation because this benchmark is widely used by the industry and academia for evaluating the various key-value storages supported. The DXRAM client implements a custom and optimized *YCSBObject* data structure based on DXRAM's chunk model to allow fast de-/serialization of the YCSB data to/from the DXRAM backend. The client uses a thread local object pool of *YCSBObject*s to avoid burdening the garbage collection. When benchmarking with many threads, concurrency is handled transparently by the DXRAM backend. The DXRAM YCSB client implementation uses the DXRAM client to connect to a DXRAM cluster to access the key-value back-end storage.

Dr. Kevin Beineke implemented the initial client, and Stefan Nothaas refactored and optimized it multiple times. Significant optimizations of the YCSB DXRAM client implementation, the DXRAM storage backend and API were necessary to allow exploiting the performance when running over InfiniBand hardware by using Ibdxnet. The client was published as a contribution to the official repository by Stefan Nothaas [44].

6.3 DXRAM Build System and Pipeline

While working on this thesis, the DXRAM project grew in size and complexity. Several parts of DXRAM were off-loaded into separate external and re-usable libraries. This separation increased the overhead of managing the project with its dependencies and exceeded the limits of the initially proposed ant-based build system. The latter was created by Dr. Florian Klein, maintained and extended by Dr. Kevin Beineke and Stefan Nothaas as development on DXRAM continued. Filip Krakowski replaced this with a custom Gradle-based build pipeline that lowered overall build times as well as maintenance efforts, improved modularity, and extensibility. Stefan Nothaas helped in designing the new build pipeline as well as fixing various bugs during the refactoring phase.

As testing and maintaining a distributed system is a very time-consuming task, unit tests are testing several smaller submodules. However, testing the fundamental functionality of the entire system is not possible with the standard unit testing framework. Thus, Stefan Nothaas developed a dedicated testing framework that allows bootstrapping and running multiple DXRAM instances locally using a custom JUnit runner with test code running on one or multiple instances.

6.4 DXRAM API and DXApplications

The API of DXRAM underwent significant changes in the course of this thesis. Dr. Florian Klein proposed the initial API and implemented CRUD operations as well as an operation to migrate chunks. With the extensive refactoring of DXRAM (see Chapter 3) and major components of it (see Chapters 4 and 5), the API was adapted and altered often.

At the time of writing this thesis, DXRAM implements two interfaces to access and use the now extensive DXRAM API: by using DXRAM as a client library (e.g., see Section 6.2) or writing a DXApp which implements the application interface. The latter is compiled as a separate jar-package and deployed along with DXRAM instances. DXRAM manages the installed applications and runs them either once when the DXRAM instance is started successfully or when the user used the DXTerm command line tool to trigger a manual start. DXApps can access various exposed core services of the DXRAM instance (see Figure 2.3) not only limited to the CRUD operations of the back-end storage but also the network subsystem, migration manager, monitoring and statistics information, task service and job service.

This containerized approach is very flexible allowing different types of applications to implement concurrent and distributed computations or algorithms to run on DXRAM storage servers.

The initial design of the DXApp API was developed and implemented by Stefan Nothaas with various contributions and ongoing development by Filip Krakowski.

Chapter 7

Conclusions and Perspectives

With more and more networked digital devices generating data today and in the future, the global data sphere is continuing to grow further. Storing and processing the resulting large data volumes is already a challenging task for companies and researchers today and becomes even more challenging in the future. With new technology arising and existing technology evolving towards this trend, new concepts must be developed to create new systems or enhance existing ones to prepare them for these highly demanding tasks.

In this thesis, the author addressed three primary research questions regarding low-latency data access in a Java-based distributed in-memory key-value storage. With the first question, the author addressed the concerns of distributed computations on an in-memory distributed key-value storage with a focus on graph-data processing in Java. The second major question discussed a memory management suitable for efficiently storing many small objects with low-latency data access for highly concurrent Java applications. Lastly, remote latency concerns were addressed by the author by using low-latency InfiniBand hardware in Java applications.

In the following sections of this concluding chapter, the author presents the achievements of this thesis (see Section 7.1), the lessons learned from his work on the thesis and its projects (see Section 7.2) and provides a perspective for future work and research (see Section 7.3).

7.1 Achievements

This section summarizes the achievements related to the three primary research challenges addressing different and essential fields in the big data application domain. The fourth challenge applies to the first three by addressing them in the Java environment which is the typical environment in the field of big data.

7.1.1 An In-Memory Key-Value Store as a Compute Platform for Parallel Java Applications

Chapter 3 elaborated on the first challenge and discussed the question if an in-memory key-value storage can be used as a scalable compute platform especially for graph data-sets with concurrent and distributed algorithms. We used the DXRAM storage system to build a flexible and scalable compute platform on top of a Java-based in-memory key-value storage to enable running distributed and concurrent computations with low-latency data access. Two additional services, the JobService and MasterSlaveService, allow developing distributed applications by providing tools for distributing and coordinating thread and inter-server concurrency. Our graph framework DXGraph is built on top of the computation layer and provides graph data structures as well as tasks for loading and processing graph data-sets.

An implementation of the BFS algorithm serves as a benchmark to compare our platform to Grappa and GraphLab, two graph processing frameworks implemented in C++. The evaluation shows that DXRAM is capable of storing graph data without introducing considerable overhead compared to a five times increase in memory consumption by GraphLab. Our BFS implementation with DXRAM and the compute framework is at least 2.5 times faster than Grappa's and GraphLab's and can be even up to five times faster when using the direction-optimized mode.

These results show that our proposed concepts and implementation in DXRAM create a robust platform for Java-based distributed and highly concurrent computations which can even outperform state-of-the-art C++-based systems.

7.1.2 Concurrent Low-Latency Data Access for Parallel Java Applications

In Chapter 4, we discussed the second question, if the local storage can provide low-latency data access and scalability on highly concurrent local computations benefitting from data locality. We presented the revised design of DXRAM's memory manager, now named DXMem, which is optimized for storing many small objects, found in graph data-sets, with a very low memory overhead as well as handling over one hundred concurrent threads efficiently. The latter is achieved by an efficient low-overhead and scalable per chunk lock implementation as well as careful optimizations to the overall memory management regarding target application workloads. The CRUD core operation-set was extended to support a variety of locking combinations. This extension allows the application to either utilize highly optimized fine-grained synchronization on a per-object basis or build more coarse-grained concurrency control mechanisms for more complex data structures. DXMem was separated from DXRAM and is now distributed as a standalone library. Using DXMem, other Java applications can benefit from its low-overhead allocator as well as low-latency memory management.

Our evaluation and comparison to the two Java-based key-value caches Hazelcast (commercially supported) and InfiniSpan show that DXMem can provide an at least eleven-times lower overhead on an average object size of 32 bytes. When comparing the local performance of the memory management, DXMem achieves single-digit microseconds latencies on read-heavy

benchmarks with 128 threads outperforming Hazelcast and InfiniSpan up to 28-fold with up to 78 mops. In a distributed setup, DXRAM using DXMem outperforms the two systems on the YCSB with an aggregated throughput of 4.6 mops on a high load read-heavy benchmark with 16 storage servers and 16 benchmark clients running over Ethernet network.

These results show that our proposed concepts applied to DXMem, a local storage, optimized for many small objects of graph data-sets, can provide low-latency data access and scalability for highly concurrent local computations in a Java environment.

7.1.3 Leveraging High-Speed and Low-Latency Networks in Java Applications

In Chapter 5, we discussed the third question, if the network can support graph-based applications efficiently regarding overall latency and handling of many small messages caused by highly concurrent random remote access. We presented several achievements regarding low-latency remote communication in Java applications.

The first is the JIB benchmark suite to evaluate existing InfiniBand solutions for Java applications providing benchmarks for comparing the verbs-based libraries jVerbs and C-verbs as well as socket-based solutions IPoIB, libvma, and JSOR. All benchmarks implemented compare the uni-directional and bi-directional throughput as well as one-sided latency and latency of a ping-pong communication pattern. Furthermore, the verbs-based implementations implement variants for using RDMA write, RDMA read and messaging operations. We used the benchmark to determine which solution is optimal for our use-case. However, other developers can also use the benchmark suite to analyze the solutions on their hardware of choice regarding their target application. Our results concluded that only a custom network subsystem that is optimized for the verbs API is capable of leveraging the potential of InfiniBand hardware in Java.

These essential conclusions were influencing our second primary achievement which is DXnet, a standalone network subsystem for highly concurrent Java applications. DXNet implements event-based messaging with high-level asynchronous and synchronous communication primitives using Java objects. DXNet's core is optimized for sending and receiving messages using a fast and efficient serialization, lock-free and zero-copy data structures. A transport interface was created to allow implementing different network interconnects. The core was designed with great attention to low overhead and low latency. Our evaluation shows that DXNet is capable of handling over one hundred threads sending messages concurrently, delivering high throughputs and low overhead to benefit from high-speed network interconnects such as InfiniBand.

The third and last primary achievement is Ibdxnet, a custom InfiniBand transport implementation for DXNet. Ibdxnet's architecture implements a scalable pipeline for low-latency and high-throughput sending and receiving of data using a ring buffer data structure. With the entire pipeline stretching from Java to native space and vice versa, a careful design that considers context switching as well as low-overhead sharing of data between spaces is crucial to optimal performance. With Ibdxnet as a transport back-end, any Java application can use DXNet over InfiniBand hardware transparently.

We evaluated DXNet with the Ibdxnet transport and compared it to the Java-based MPI implementation FastMPJ as well as the C-based MPI implementation MVAPICH2, both supporting InfiniBand. DXNet's throughput with Ibdxnet is on par with FastMPJ's and MVAPICH2's on middle and large-sized messages (up to 1 MB). On small messages (up to 512 bytes), DXNet outperforms both systems especially in a multi-threaded environment at least two-fold with an aggregated message throughput of 8.6 to 10.2 mmps. In an all-to-all benchmark with 8 servers, DXNet shows overall robust scalability on a worst-case situation. DXNet outperforms FastMPJ and MVAPICH2 by up to two-fold with approx. 33.2 to 43.4 mmps for up to 64-byte messages. Naturally, with DXNet's core providing concurrency control, transparent message object serialization and event-based dispatching, the overall latency with the InfiniBand transport (best-case RTT of 8 µs) is higher compared to the more bare-metal MPI implementation MVAPICH2 (base-case RTT of 2 to 4 µs).

Another comparison of DXRAM using DXNet with the Ibdxnet transport to the InfiniBand-based RAMCloud storage system shows that DXRAM can leverage the performance of InfiniBand hardware using DXNet. We used the YCSB benchmark with 20 storage and 20 benchmark servers to evaluate the two systems. DXRAM outperforms RAMCloud on a reference workload two-fold and a graph-based application workload even five-fold.

These results show that our proposed concepts and implementation in DXNet and Ibdxnet can provide high performance messaging for concurrent Java applications which even outperforms MVAPICH2, a well established and mature C-based MPI implementation.

7.1.4 Java as a Suitable Language for High Performance and Low-Latency Applications

In the previous Sections 7.1.1, 7.1.2 and 7.1.3, we summarized the achievements of the three primary research questions of this thesis. All three were discussed in the context of the DXRAM storage system implemented in Java. Naturally, some parts had to be implemented as native modules in C or C++ when having to rely on low-level OS functionality (optimized access to SSDs for DXRAM's logging) or libraries (C-verbs with Ibdxnet for DXNet) that were not available in the Java environment and also performance critical. The evaluations and comparisons show that DXRAM or subsystems of it can outperform systems implemented in C and C++. Type safety, garbage collection, and object orientation can be used and applied to benefit from enhanced code safety and maintainability without sacrificing performance. This thesis and the DXRAM project shows that the Java language can be used for high performance and big data applications.

7.2 Lessons Learned

7.2.1 The DXRAM Project

With two generations of Ph.D. students have completed their research and work on DXRAM and the third generation already working on new and challenging research questions, the code base has grown a lot since 2012. More than 1,000 source files contain over 100,000 lines of code (the majority in Java with portions in C and C++) and 50,000 lines of comments and documentation. A maintainable project structure was implemented from the beginning that ensured a good evolution and re-usability of the system over the last years and for future research and contributions.

Many parts of DXRAM are optimized for low overhead and low latency typically regarding memory (also including the Java garbage collection) and CPU resources. A lot of time and effort was spent on extensive profiling and debugging the system. These tasks became increasingly difficult with performance issues or bugs that cannot be approached with commonly used techniques, such as debuggers or print-debugging. These methods introduce additional latency causing disadvantageous timing shifting. In such situations, especially in a (large) distributed setup, bugs could not be debugged because either different errors occurred due to the shift in timing or the bug did not appear anymore. Such a problematic situation required clever and context-sensitive tricks by using simple counters, flags and in-memory logging of data, only. These tricks ensured a correct reproduction of the bug to solve and allowed getting additional information from the system for solving it.

In the course of this thesis, the author was also challenged by various issues that did not originate from the software he developed. Though this thesis has shown that Java is a suitable language for high-performance and low-latency applications, the Java environment was repeatedly responsible for unexpected performance issues. For example, the Java garbage collection was often halting the JVM due to previously unknown and unexpected allocations caused by used packages of the standard Java library.

Another unexpected issue arose with the Linux kernel's dedicated worker threads (kworker) for dispatching kernel related tasks like the handling of interrupts, timers or I/O. We encountered situations, even on recent kernel versions, that these threads start running on high load and not stopping anymore after the server was running for an extended period. Often, this was not detected immediately and lead to a lowered performance or even inexplicable results on benchmarks or when debugging DXRAM. Even this issue has been known for years [67], we could not find a solution that fixed this for us other than rebooting the machine. Thus, we always restarted all servers before a more complex debugging session or evaluation to ensure this issue is not present.

7.2.2 InfiniBand

Working with low-latency hardware was an exciting but also challenging experience. With single-digit microseconds latency, there is no room for subpar solutions or code. Small mistakes either increase latency ten- or hundred-fold, or even render the system unusable. One of the most frustrating experiences of the author of this thesis was researching a performance issue related to sending small messages using messaging verbs over a reliable connection. On high loads, often, the throughput was as expected but could randomly drop down to sending less than ten messages per second. It was not possible to reproduce this issue consistently. Researching and debugging of this required over one month as it was unclear for a very long time if this was a software or hardware issue, or even both. During this period, the author significantly extended his knowledge and understanding of InfiniBand down to the hardware protocol level. As a result, the author of this thesis found out that the RC protocol randomly introduced inappropriate long sleep times (up to 500 ms) when the receiver was not ready to receive the data. In general, this mechanism is implemented to avoid polling on the protocol level when the remote cannot provide resources for receiving data at that moment. However, determining the wait time to slow down the sender was buggy and randomly created a practically blocked pipeline instead. This issue was reported to and discussed with Mellanox engineers who also confirmed it on older versions of the OFED software package. Since OFED version 4.2, this issue is resolved, and DXNet can always deliver high throughput using InfiniBand.

While issues with software are typically more common, hardware issues must always be considered as well. The author used the HPC cluster of the ZIM of the University of Düsseldorf in several experiments during this thesis. In experiments with InfiniBand, the author detected that the fundamental latency of their Ivybridge-based cluster with 56 Gbit/s InfiniBand is significantly higher (about 7 µs) compared to their newer Skylake-based cluster with 100 Gbit/s. A comparison to the private cluster of the operation systems workgroup yielded results identical to the Skylake-based cluster. All tests to determine this issue were limited to a single chassis for a single hop on the HPC cluster. The Ivybridge-based cluster uses chassis by Bull with 18 blades each and a built-in 56 Gbit/s InfiniBand switch. The cluster of the operation systems workgroup connects its "pizza box" sized servers to a typical Mellanox 56 Gbit/s switch. The results of the experiments with different hardware configurations concluded that this fundamentally higher latency on the Ivybridge-based cluster is caused by the 56 Gbit/s switch built into the chassis. We also ruled out the possibility of slow CPUs by evaluating the switch built into the Bull chassis and the standalone 56 Gbit/s Mellanox switch with the same CPU. For some inexplicable reason, InfiniBand packages using the RC protocol were more often NACK'd on the switch built into the Bull chassis resulting in more re-transmissions increasing overall latency.

7.3 Future Work and Perspectives

This last section presents open questions and opportunities regarding big data processing in Java as well as the DXRAM system that were revealed by this thesis. Many good results were obtained through extensive evaluation and serve as a foundation for future research regarding the general perspective of a Java-based compute platform for graph applications, efficient and highly concurrent memory management and exploiting low-latency networks in Java.

7.3.1 Memory Management

Parallel create operations. The initial design by Dr. Florian Klein favored the less frequently used create operations over more typically used get/put operations when considering common big data, and especially graph processing workloads. This design decision resulted in significant performance issues in especially highly concurrent benchmarks leading to the revised design proposed in Chapter 4. However, as we started experimenting with huge graph data sets (up to 8 TB), these had to be loaded from disk before executing the benchmark or computation. Even when distributed to multiple servers, loading times for the data were significantly higher compared to the old design. Naturally, the loading phase consisted of create-operations only, and the memory management could not exploit local parallelism well. However, when proposing and implementing the revised design, we could not come up with a solution to alter the arena manager and segmentation to work with the new design. Now, with the new design and many optimizations implemented, the memory management achieves sub-microseconds overhead outperforming other state-of-the-art systems many times over. These results would allow bringing back the initial arena management by introducing another lock level and segmentation to allow concurrent create operations. With the new lock design providing very low overhead, this should not impact the latency of get and put-operations significantly anymore.

Defragmentation. Offline graph analytics typically load the data once, process it, output the results, and the task ends. The analysis does not generate a lot of new data or even none at all but reads and writes the loaded data mostly. Long-running applications also issue create and remove operations over time, e.g., when a user posts text, uploads or deletes assets. Thus, the memory management cannot avoid introducing external fragmentation over time. Our new design allows running a concurrent defragmentation thread that can work on a per chunk level by using the new locking mechanism implemented. This change allows higher concurrency for the application because the blocking overhead is reduced compared to a more coarse-grained approach. However, intensive research by Florian Hucke, one of our students, has shown that designing and implementing an appropriate defragmentation strategy is a complex topic in itself. Due to time constraints, this field could not be researched any further in the course of this thesis.

Evaluation on high core count CPUs. With multi-threaded programming and applications utilizing many-core CPUs common today, the per CPU socket core count has further increased over the years. This trend also spawned CPUs with up to 48 cores (e.g., the Cavium ThunderX with 2x 48 core ARMv8 CPUs) which allow high concurrency on the hardware level. It would be interesting to see how the low-overhead memory management performs when exposed to an even higher level of software and hardware concurrency.

7.3.2 Network

Scalability with one hundred servers and more. We have already experimented with DXNet and the InfiniBand transport on the HILBERT cluster with one hundred nodes, but the results were not showing optimal scalability with fluctuations. Naturally, debugging and optimizing at that scalability level is a very challenging task and allocating the required resources is not always possible on a shared cluster. We could determine that remote scalability was

limited by improper thread management suppressing the higher priority back-end threads driving the InfiniBand transport (see also the last paragraph of Section 7.3.4). This situation is further worsened by more active connections on an increasing scale of a worst-case all-to-all communication pattern. We suggest that this thread management issue has to be resolved first before we can adequately re-evaluate the scalability of DXNet with InfiniBand.

Scalability on next-generation InfiniBand hardware. The core design of DXNet is already well prepared for handling many small messages on a large scale. We have also shown, that its overall latency overhead is low and independent of the transport used, and it can provide throughputs exceeding the capabilities of our currently used hardware (56 Gbit/s). Thus, the core is already well prepared at least for the next generation of hardware (100 Gbit/s). Further experiments should be conducted to verify this as well as evaluate the scalability regarding increasing server count.

UD transport and scalability. UD QPs are typically used on scalability concerns regarding an increasing count of remote servers. Fabian Ruhland already implemented a UD-based backend for Ibdxnet to analyze this and compare the performance to Ibdxnet's RC-based implementation. We reckon that our future evaluation might unveil insights on possible scalability issues that allows us to improve Ibdxnet's current design further for increased scalability.

Leveraging RDMA operations for even faster remote access. Utilizing RDMA operations for low-latency remote data access is already a popular field of research when the storage is developed in native languages such as C or C++ (see Section 2.2). Naturally, it is more challenging to bring this technology to the Java space and allow Java applications, in particular, in-memory storage systems, to leverage the performance of the hardware using RDMA operations. However, this performance cannot be exploited if the application is not suitable. When the application and its algorithms rely on messaging based communication, typically used for implicit remote synchronization in an algorithm, RDMA operations accessing remote data without involving the CPU might not improve the performance or cannot even be implemented at all. In general, it is not guaranteed that RDMA operations should always be favored over messaging verbs [122]. Such a design decision requires further time and effort but is an exciting research topic especially in the context Java big data processing, and not just limited to in-memory key value storages.

Experiments with 40/100 Gbit/s Ethernet hardware. With 10 Gbit/s already common today and 40/100 Gbit/s Ethernet becoming available, especially in the field of HPC [120], an evaluation of the NIO transport of DXNet with such modern Ethernet hardware and comparison to the InfiniBand transport might yield interesting results regarding the overall performance of DXNet and the performance of the transports. As standard Ethernet-based HCAs do not implement a full offload engine like InfiniBand HCAs, CPU might limit scalability. This limitation could be further evaluated and compared to additional TCP offload engine hardware to analyze the various benefits and drawbacks of the solutions in the context of highly concurrent Java applications with DXNet.

7.3.3 Fast and Scalable Deployment for Development of Distributed Applications

In the course of this thesis, the author spent much time with deploying applications to multiple servers on a remote cluster either for development or evaluation. Thus, the author spent additional effort on reducing the time and work involved in writing and maintaining scripts by creating cdepl (see Section 6.1). However, cdepl and its design reached its limits regarding scalability when deploying to more than one hundred servers. The scripting language Bash has noticeable deficits in processing speed on such a larger scale. The cdepl project grew more extensive than expected which makes it more difficult to maintain in a scripting language. A revised design of cdepl, called *jdepl*, uses the Java environment instead and implements a scalable deployment approach. Daemons running on the servers to deploy to are forming a tree-based topology that is controlled by a master instance. Thus, deployment to many servers can be routed using the tree-based structure reducing deployment time and increasing scalability compared to the SSH-based approach with cdepl. Due to time constraints, the author could not complete the implementation of jdepl which was started in the course of this thesis. The project is handed to Filip Krakowski who continues with development.

7.3.4 DXRAM as a Compute Platform

DXRAM started as an in-memory key-value storage with minimal API and evolved over the years to a Java-based compute platform for low-latency and highly concurrent and distributed applications. This approach fits the general trend of convergence of big data and HPC which describes a "new type of distributed service platform" combining "computing communication, and buffer/storage resources in a data processing network" [134].

Distributed data structures. With DXRAM having evolved into an already stable system, various application driven projects have been implemented and tested with the system. These projects provided valuable insights on the future research and development of DXRAM. Currently, different implementations of distributed data structures (e.g., list, tree, hash map) are being worked on and evaluated to create a basic re-usable toolbox for distributed applications.

Thread mangement. With the optimizations and re-design of DXMem and DXNet, DXRAM's back-end is optimized well for high concurrency applications. However, evaluations with high-load workloads and the InfiniBand transport have shown a significant increase of the overall remote operation latency when utilizing hundreds of threads overprovisioning commodity cluster hardware. This latency increase is caused by improper thread management suppressing important and higher priority back-end threads (e.g., network) required for driving important core processing pipelines. This issue could be addressed by a custom thread management that allows prioritizing these back-end threads over application threads. Core-aware thread management implemented by Arachne [106] has shown to improve the overall performance of systems significantly and should be considered when developing a solution for this issue.

Acronyms

API Application Programming Interface

BASE Basically Available, Soft state, Eventual consistenty

BFS Breadth-First Search

CID Chunk ID

CPU Central Processing Unit

CRUD Create, Read, Update, Delete

CUB Catch-Up Buffer

HCA Host Channel Adapter

HDD Hard Disk Drive

HPC High Performance Computing

I/O Input/Output

IPoIB IP over InfiniBand

JNI Java Native Interface

JVM Java Virtual Machine

LID Local ID

mmps million messages per second

NID Node ID

ORB Outgoing Ring Buffer

QP Queue Pair

RAM Random Access Memory

RC Reliable Connected

RDBMS Relational DataBase Management System

RDMA Remote Direct Memory Access

SDP Sockets Direct Protocol

SIMD Single Instruction, Multiple Data

SLA Service-Level-Agreement

SQL Structured Query Language

SSD Solid State Disk

SSH Secure SHell

UD Unreliable Datagram

YCSB Yahoo! Cloud Serving Benchmark

Bibliography

[1] *7 Big Data Examples – Application of Big Data in Real Life*. https://intellipaat.com/blog/7-big-data-examples-application-of-big-data-in-real-life/. Accessed: 2019-02-24 (Page: 1).

[2] *Amazon Web Services*. https://aws.amazon.com/. Accessed: 2019-02-24 (Page: 2).

[3] *Apache Crail (Incubating)*. https://crail.incubator.apache.org/. Accessed: 2019-02-24 (Pages: 14, 15).

[4] *Apache Giraph*. https://giraph.apache.org/. Accessed: 2019-02-24 (Page: 14).

[5] *Apache Spark RDMA plugin*. https://community.mellanox.com/docs/DOC-3012. Accessed: 2019-02-24 (Page: 15).

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. "Workload Analysis of a Large-scale Key-value Store". In: *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '12. 2012, pp. 53–64 (Page: 3).

[7] *Azure Cloud Services*. https://azure.microsoft.com/en-us/. Accessed: 2019-02-24 (Page: 2).

[8] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. "Megastore: Providing scalable, highly available storage for interactive services". In: *5th Biennial Conference on Innovative Data Systems Research*. CIDR'11. Jan. 2011, pp. 223–234 (Page: 14).

[9] Sumita Barahmand and Shahram Ghandeharizadeh. "BG: A Benchmark to Evaluate Interactive Social Networking Actions". In: *CIDR*. 2013 (Page: 19).

[10] Scott Beamer, Krste Asanovic, and David Patterson. "Direction-optimizing Breadth-first Search". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. IEEE Computer Society Press, 2012, 12:1–12:10 (Page: 3).

[11] Kevin Beineke. "Schnelle parallele Fehlererholung in verteilten In-Memory Key-Value Systemen". PhD thesis. Universitaetsstrasse 1, 40225 Düsseldorf, Germany: Institute for Computer Science, Heinrich-Heine University Düsseldorf, June 2018 (Pages: 17, 63).

[12] Kevin Beineke, Stefan Nothaas, and Michael Schoettner. "Efficient Messaging for Java Applications Running in Data Centers". In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. May 2018, pp. 589–598 (Pages: 18, 63).

[13] Kevin Beineke, Stefan Nothaas, and Michael Schoettner. "Fast Parallel Recovery of Many Small In-Memory Objects". In: *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. Dec. 2017, pp. 248–257 (Page: 18).

[14] Kevin Beineke, Stefan Nothaas, and Michael Schoettner. "High Throughput Log-Based Replication for Many Small In-Memory Objects". In: *IEEE 22nd International Conference on Parallel and Distributed Systems*. Dec. 2016, pp. 535–544 (Page: 18).

[15] Kevin Beineke, Stefan Nothaas, and Michael Schoettner. "High Throughput Log-Based Replication for Many Small In-Memory Objects". In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2016, pp. 160–161 (Page: 18).

[16] Kevin Beineke, Stefan Nothaas, and Michael Schoettner. "Parallelized Recovery of Hundreds of Millions Small Data Objects". In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2017, pp. 621–622 (Page: 18).

[17] Kevin Beineke, Stefan Nothaas, and Michael Schoettner. "Scalable Messaging for Java-based Cloud Applications". In: *ICNS 2018, The Fourteenth International Conference on Network and Services* 14 (May 2018), pp. 32–41 (Pages: 18, 63).

[18] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC'13. 2013, pp. 49–60 (Page: 3).

[19] Josiah L. Carlson. *Redis in Action*. Greenwich, CT, USA: Manning Publications Co., 2013 (Pages: 12, 13).

[20] *cdepl Project on Github*. `https://github.com/hhu-bsinfo/cdepl`. Accessed: 2019-02-24 (Pages: 10, 118).

[21] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. "FASTER: A Concurrent Key-Value Store with In-Place Updates". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. 2018, pp. 275–290. ISBN: 978-1-4503-4703-7 (Pages: 12, 14).

[22] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Trans. Comput. Syst.* 26 (June 2008), 4:1–4:26 (Pages: 2, 3, 13).

[23] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. "One Trillion Edges: Graph Processing at Facebook-scale". In: *Proc. VLDB Endow.* 8 (Aug. 2015), pp. 1804–1815 (Page: 3).

[24] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Commun. ACM* 13.6 (June 1970), pp. 377–387 (Page: 13).

[25] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking cloud serving systems with YCSB". In: *Proc. of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154 (Pages: 19, 120).

[26] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113 (Page: 2).

[27] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. "Efficient Big Data Processing in Hadoop MapReduce". In: *Proc. VLDB Endow.* 5 (Aug. 2012), pp. 2014–2015 (Pages: 2, 12, 14).

[28] N.A. Doekemeijer and A.L. Varbanescu. *A Survey of Parallel Graph Processing Frameworks*. Technical Report: PDS-2014-003. Delft University of Technology, 2014 (Pages: 3, 14).

[29] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. "FaRM: Fast Remote Memory". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Apr. 2014, pp. 401–414 (Pages: 14, 15).

[30] *DXMem Project on Github*. `https://github.com/hhu-bsinfo/dxmem`. Accessed: 2019-02-24 (Pages: 10, 19).

[31] *DXNet Project on Github*. `https://github.com/hhu-bsinfo/dxnet`. Accessed: 2019-02-24 (Pages: 9, 19).

[32] *DXRAM Project on Github*. `https://github.com/hhu-bsinfo/dxram`. Accessed: 2019-02-24 (Pages: 9, 17).

[33] *Ehcache Homepage*. `http://www.ehcache.org` (Page: 13).

[34] Marc Ewert. "A thread-pool-based network interface for a distributed in-memory storage". Master's Thesis. Universitaetsstrasse 1, 40225 Düsseldorf, Germany: Institute for Computer Science, Heinrich-Heine University Düsseldorf, Feb. 2015 (Page: 50).

[35] Roberto R. Exposito, Sabela Ramos, Guillermo L. Taboada, Juan Touriño, and Ramón Doallo. "FastMPJ: a scalable and efficient Java message-passing library". In: *Cluster Computing* 17 (Sept. 2014), pp. 1031–1050 (Page: 15).

[36] *Facebook Reports Third Quarter 2018 Results*. `https://investor.fb.com/investor-news/press-release-details/2018/Facebook-Reports-Third-Quarter-2018-Results/default.aspx`. Accessed: 2019-02-24 (Page: 2).

[37] Brad Fitzpatrick. "Distributed caching with memcached". In: *Linux journal* 2004 (2004) (Page: 13).

[38] M. R. Garey, D. S. Johnson, and L. Stockmeyer. "Some Simplified NP-complete Problems". In: *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*. STOC '74. 1974, pp. 47–63 (Page: 4).

[39] *GemFire - In-Memory Data Grid powered by Apache Geode*. `https://pivotal.io/pivotal-gemfire`. Accessed: 2019-02-24 (Pages: 12, 14).

[40] *Github: DXDdl*. `https://github.com/hhu-bsinfo/dxddl`. Accessed: 2019-02-24 (Page: 17).

[41] *Github: DXGraph*. `https://github.com/hhu-bsinfo/dxgraph`. Accessed: 2019-02-24 (Page: 10).

[42] *Github: Ibdxnet*. `https://github.com/hhu-bsinfo/ibdxnet`. Accessed: 2019-02-24 (Page: 9).

[43] *Github: Java InfiniBand Benchmarks*. `https://github.com/hhu-bsinfo/jib-benchmarks/`. Accessed: 2019-02-24 (Page: 52).

[44] *Github: Yahoo! Cloud Serving Benchmark*. `https://github.com/brianfrankcooper/YCSB`. Accessed: 2019-02-24 (Page: 120).

[45] Dror Goldenberg, Tzachi Dar, and Gilad Shainer. "Architecture and Implementation of Sockets Direct Protocol in Windows". In: *2006 IEEE International Conference on Cluster Computing* (2006), pp. 1–9 (Page: 15).

[46] *Google Cloud*. `https://cloud.google.com/compute/`. Accessed: 2019-02-24 (Page: 2).

[47] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. "The rise of "big data" on cloud computing: Review and open research issues". In: *Information Systems* 47 (2015), pp. 98–115 (Page: 2).

[48] *Hazelcast - An in-memory Data Grid*. https://hazelcast.com. Accessed: 2019-02-24 (Pages: 12, 13).

[49] *High-Performance Big Data (HiBD)*. http://hibd.cse.ohio-state.edu/. Accessed: 2019-02-24 (Page: 15).

[50] Joel Hruska. *Intel, Micron reveal Xpoint, a new memory architecture that could outclass DDR4 and NAND*. https://www.extremetech.com/extreme/211087-intel-micron-reveal-xpoint-a-new-memory-architecture-that-claims-to-outclass-both-ddr4-and-nand. Accessed: 2019-02-24. 2015 (Page: 4).

[51] Florian Hucke. "Optimierung paralleler Zugriffe auf eine Speicherverwaltung für viele kleine Objekte". Heinrich-Heine-Universität Düsseldorf, 2018 (Page: 36).

[52] *InfiniBand Enables the Most Powerful Cloud: Windows Azure*. https://www.mellanox.com/blog/2014/02/infiniband-enables-the-most-powerful-cloud-windows-azure/. Accessed: 2019-02-24 (Pages: 4, 15).

[53] *Infinispan*. http://infinispan.org/. Accessed: 2019-02-24 (Pages: 12, 13).

[54] *InfiniteGraph*. https://www.objectivity.com/products/infinitegraph/. Accessed: 2019-02-24 (Page: 14).

[55] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. "LDBC Graphalytics: A Benchmark for Large-scale Graph Analysis on Parallel and Distributed Platforms". In: *Proc. VLDB Endow.* 9.13 (Sept. 2016), pp. 1317–1328 (Pages: 3, 19, 21).

[56] *IP over InfiniBand (IPoIB) Architecture*. https://www.ietf.org/rfc/rfc4392.txt. Accessed: 2019-02-24 (Page: 15).

[57] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md. Wasi-ur-Rahman, Nusrat S. Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, and Dhabaleswar K. Panda. "Memcached Design on High Performance RDMA Capable Interconnects". In: *Proceedings of the 2011 International Conference on Parallel Processing*. ICPP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 743–752. ISBN: 978-0-7695-4510-3. DOI: 10.1109/ICPP.2011.37. URL: http://dx.doi.org/10.1109/ICPP.2011.37 (Page: 15).

[58] E. Jovanov. "Wireless Technology and System Integration in Body Area Networks for m-Health Applications". In: *2005 IEEE Engineering in Medicine and Biology 27th Annual Conference*. Jan. 2005, pp. 7158–7160 (Page: 2).

[59] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Using RDMA Efficiently for Key-value Services". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. ACM, 2014, pp. 295–306 (Pages: 14, 15).

[60] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *SIAM J. Sci. Comput.* 20.1 (Aug. 1998), pp. 359–392 (Page: 3).

[61] Florian Klein. "Metadata-Management in a distributed In-Memory Storage". PhD thesis. Universitaetsstrasse 1, 40225 Düsseldorf, Germany: Institute for Computer Science, Heinrich-Heine University Düsseldorf, Nov. 2015 (Pages: 5, 17, 18, 22, 34).

[62] Florian Klein, Kevin Beineke, and Michael Schoettner. "Distributed Range-Based Meta-Data Management for an In-Memory Storage". In: *LNCS Europar Workshop Proceedings, 4th Big Workshop on Big Data Managements in Clouds*. Sept. 2015 (Page: 17).

[63] Florian Klein, Kevin Beineke, and Michael Schoettner. "Memory Management for Billions of Small Objects in a Distributed In-Memory Storage". In: *IEEE Cluster 2014*. Sept. 2014 (Pages: 18, 34).

[64] Florian Klein and Michael Schoettner. "Dxram: A persistent in-memory storage for billions of small objects". In: *Proceedings of the 14th International Conference on Parallel and Distributed Computing, Applications and Technologies*. PDCAT 13. 2013 (Page: 17).

[65] Filip Krakowski. "Nebenläufige Migration großer In-Memory-Datenmengen". Master's Thesis. Heinrich-Heine-Universität Düsseldorf, 2018 (Page: 18).

[66] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. "What is Twitter, a Social Network or a News Media?" In: *Proceedings of the 19th International Conference on World Wide Web*. WWW '10. 2010, pp. 591–600 (Page: 2).

[67] *Kworker, what is it and why is it hogging so much CPU?* `https://askubuntu.com/questions/33640/kworker-what-is-it-and-why-is-it-hogging-so-much-cpu`. Accessed: 2019-02-24 (Page: 126).

[68] Neal Leavitt. "Will NoSQL Databases Live Up to Their Promise?" In: *Computer* 43.2 (Feb. 2010), pp. 12–14 (Page: 13).

[69] Jure Leskovec and Christos Faloutsos. "Sampling from Large Graphs". In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '06. 2006, pp. 631–636 (Page: 3).

[70] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. "Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations". In: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. KDD '05. 2005, pp. 177–187 (Page: 3).

[71] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks". In: SOCC '14. 2014 (Page: 13).

[72] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999 (Page: 12).

[73] *libvma Github Wiki*. `https://github.com/Mellanox/libvma/wiki/Architecture`. Accessed: 2019-02-24 (Page: 15).

[74] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 429–444 (Pages: 14, 15).

[75] *List of NoSQL Databases*. `http://nosql-database.org/`. Accessed: 2019-02-24 (Page: 13).

[76] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proc. VLDB Endow.* 5 (Apr. 2012), pp. 716–727 (Pages: 12, 14).

[77] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. 2010, pp. 135–146 (Page: 14).

[78] W. Glynn Mangold and David J. Faulds. “Social media: The new hybrid element of the promotion mix”. In: *Business Horizons* 52 (2009), pp. 357–365 (Page: 2).

[79] James Martin. *Managing the Data Base Environment*. 1st. Prentice Hall PTR, 1983 (Page: 13).

[80] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. “Use at Your Own Risk: The Java Unsafe API in the Wild”. In: *SIGPLAN Not.* 50 (Oct. 2015), pp. 695–710 (Page: 12).

[81] *Mellanox Website*. `https://www.mellanox.com/`. Accessed: 2019-02-24 (Page: 4).

[82] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. “Graph Structure in the Web — Revisited: A Trick of the Heavy Tail”. In: *Proceedings of the 23rd International Conference on World Wide Web*. WWW ’14 Companion. 2014, pp. 427–432 (Page: 3).

[83] *mpiJava: A Java Interface to MPI*. `http://www.hpjava.org/mpiJava.html`. Accessed: 2019-02-24 (Page: 15).

[84] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. “Introducing the graph 500”. In: (2010) (Pages: 3, 4, 19, 21, 23).

[85] *MVAPICH Website*. `http://mvapich.cse.ohio-state.edu/`. Accessed: 2019-02-24 (Page: 15).

[86] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. *Grappa: A latency-tolerant runtime for large-scale irregular applications*. Technical Report. 2014 (Pages: 12, 14).

[87] *Neo4j*. `https://neo4j.com/`. Accessed: 2019-02-24 (Page: 14).

[88] Kai Stefan Neyenhuys. “Entwicklung und Evaluierung eines vielseitigen Indexservice innerhalb eines verteilten In-Memory Key-Value Stores”. Master’s Thesis. Heinrich-Heine-Universität Düsseldorf, 2018 (Page: 17).

[89] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski and Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. “Scaling Memcache at Facebook”. In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 2013, pp. 385–398 (Pages: 2–4).

[90] Stefan Nothaas, Kevin Beineke, and Michael Schoettner. *Ibdxnet: Leveraging InfiniBand in Highly Concurrent Java Applications*. `https://arxiv.org/abs/1812.01963`. 2018. eprint: `arXiv:1812.01963` (Pages: 18, 75).

[91] Stefan Nothaas, Kevin Beineke, and Michael Schöttner. “Distributed Multithreaded Breadth-first Search on Large Graphs Using DXGraph”. In: *Proceedings of the First International Workshop on High Performance Graph Data Management and Processing*. HPGDMP ’16. 2016, pp. 1–8 (Page: 19).

[92] Stefan Nothaas, Kevin Beineke, and Michael Schöttner. “Leveraging InfiniBand for Highly Concurrent Messaging in Java Applications”. Unpublished (Pages: 18, 75).

[93] Stefan Nothaas, Kevin Beineke, and Michael Schöttner. “Optimized Memory Management for a Java-Based Distributed In-Memory System”. Submitted to the 4th International Workshop on Scalable Compution For Real-Time Big Data Applications (SCRAMBL), 2019 (Pages: 18, 36).

[94] Stefan Nothaas, Fabian Ruhland, and Michael Schöttner. "A Benchmark Suite to Evaluate InfiniBand Solutions for Java Applications". Submitted to the 28th International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC), 2019 (Pages: 18, 52).

[95] *Open MPI*. `https://www.open-mpi.org/`. Accessed: 2019-02-24 (Page: 15).

[96] *Open source memory-centric distributed database, caching and processing platform - Apache Ignite*. `https://ignite.apache.org/index.html`. Accessed: 2019-02-24 (Pages: 12, 13).

[97] *OpenFabrics Alliance*. `https://www.openfabrics.org/`. Accessed: 2019-02-24 (Page: 15).

[98] Oracle. *Java I/O, NIO, and NIO.2*. `https://docs.oracle.com/javase/8/docs/technotes/guides/io/index.html`. Accessed: 2018-01-21 (Page: 12).

[99] Mohd Fauzi Othman and Khairunnisa Shazali. "Wireless Sensor Network Applications: A Study in Environment Monitoring System". In: *Procedia Engineering* 41 (2012), pp. 1204–1210 (Page: 2).

[100] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. "The RAMCloud Storage System". In: *ACM Trans. Comput. Syst.* 33 (Aug. 2015), 7:1–7:55 (Pages: 12, 14, 15).

[101] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Nov. 1999 (Page: 3).

[102] *Performance of Java versus C++*. `http://scribblethink.org/Computer/javaCbenchmark.html`. Accessed: 2019-02-24 (Page: 12).

[103] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. "Transactional consistency and automatic management in an application data cache". In: *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. OSDI'10. 2010, pp. 1–15 (Page: 13).

[104] Dan Pritchett. "BASE: An Acid Alternative". In: *Queue* 6 (May 2008), pp. 48–55 (Page: 13).

[105] Sander Pronk, Szilárd Páll, Roland Schulz, Per Larsson, Pär Bjelkmar, Rossen Apostolov, Michael R. Shirts, Jeremy C. Smith, Peter M. sson, David van der Spoel, Berk Hess, and Erik Lindahl. "GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit". In: *Bioinformatics* 29 (2013), pp. 845–854 (Page: 2).

[106] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. "Arachne: Core-Aware Thread Management". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 145–160 (Pages: 5, 130).

[107] David Reinsel, John Gantz, and John Rydning. *The Digitization of the World - From Edge to Core*. `https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf`. Accessed: 2019-02-24. 2018 (Page: 1).

[108] Fabian Ruhland. "Entwicklung eines Benchmarks für InfiniBand-Kommunikation in Java-Anwendungen". Master's Thesis. Universitaetsstrasse 1, 40225 Düsseldorf, Germany: Institute for Computer Science, Heinrich-Heine University Düsseldorf, Sept. 2018 (Pages: 18, 52).

[109] Scott Sallinen, Keita Iwabuchi, Suraj Poudel, Maya Gokhale, Matei Ripeanu, and Roger Pearce. "Graph Colouring As a Challenge Problem for Dynamic Graph Processing on Distributed Systems". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '16. 2016, 30:1–30:12 (Page: 3).

[110] Pratik Satapathy, Jash Dave, Priyanka Naik, and Mythili Vutukuru. "Performance Comparison of State Synchronization Techniques in a Distributed LTE EPC". In: *IEEE Conf. on Network Function Virtualization and Software Defined Networks*. 2017 (Pages: 2, 3).

[111] Michael Schlapa. "Auswertung verschiedener InfiniBand-Implementierung für Java". Master's Thesis. Universitaetsstrasse 1, 40225 Düsseldorf, Germany: Institute for Computer Science, Heinrich-Heine University Düsseldorf, Mar. 2016 (Pages: 52, 75).

[112] Michael Schroeck, Rebecca Shockley, Janet Smart, Dolores Romero-Morales, and Peter Tufano. *IBM Institute for Business Value, IBM Institute for Business Value-Executive Report*. 2012 (Page: 1).

[113] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller. "UCX: An Open Source Framework for HPC Network APIs and Beyond". In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. Aug. 2015, pp. 40–43 (Page: 15).

[114] Bin Shao, Haixun Wang, and Yatao Li. "Trinity: A Distributed Graph Engine on a Memory Cloud". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. 2013, pp. 505–516 (Page: 14).

[115] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. "Efficient transaction processing in SAP HANA database: the end of a column store myth". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. 2012, pp. 731–742 (Pages: 2, 12, 13).

[116] *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. Accessed: 2019-02-24 (Pages: 2, 3, 21).

[117] Chris Snijders, Uwe Matzat, and Ulf-Dietrich Reips. ""Big Data" : Big Gaps of Knowledge in the Field of Internet Science". In: *International Journal of Internet Science* 7 (Jan. 2012), pp. 1–5 (Page: 1).

[118] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. "Aerospike: Architecture of a Real-time Operational DBMS". In: *Proc. VLDB Endow.* 9 (Sept. 2016), pp. 1389–1400 (Pages: 12, 14).

[119] *Stackoverflow: Is Java really slow?* `https://stackoverflow.com/questions/2163411/is-java-really-slow`. Accessed: 2019-02-24 (Page: 12).

[120] *Statistics | TOP500 Supercomputer Sites*. `https://www.top500.org/statistics/`. Accessed: 2019-02-24 (Pages: 2, 4, 15, 129).

[121] Patrick Stuedi, Bernard Metzler, and Animesh Trivedi. "jVerbs: Ultra-low Latency for Data Center Applications". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. New York, NY, USA: ACM, 2013, 10:1–10:14 (Page: 15).

[122] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. "RFP: When RPC is Faster Than Server-Bypass with RDMA". In: *Proc. of the 12th European Conf. on Computer Systems*. 2017, pp. 1–15 (Page: 129).

[123] Wenhui Tang, Yutong Lu, Nong Xiao, Fang Liu, and Zhiguang Chen. "Accelerating Redis with RDMA Over InfiniBand". In: *Data Mining and Big Data*. Ed. by Ying Tan, Hideyuki Takagi, and Yuhui Shi. Cham: Springer International Publishing, 2017, pp. 472–483 (Page: 15).

[124] *The Human Connectome Project*. `http://www.humanconnectomeproject.org/`. Accessed: 2019-02-24 (Pages: 2, 3).

[125] *The size of the World Wide Web (The Internet)*. `https://www.worldwidewebsize.com/`. Accessed: 2019-02-24 (Page: 2).

[126] Sivasakthi Thirugnanapandi, Sreedhar Kodali, Neil Richards, Tim Ellison, Xiaoqiao Meng, and Indrajit Poddar. *Transparent network acceleration for Java-based workloads in the cloud*. `https://www.ibm.com/developerworks/library/j-transparentaccel/`. 2014 (Page: 15).

[127] *TITAN - Distributed Graph Database*. `http://titan.thinkaurelius.com/`. Accessed: 2019-02-24 (Page: 14).

[128] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. "FENNEL: Streaming Graph Partitioning for Massive Scale Graphs". In: *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*. WSDM '14. ACM, 2014, pp. 333–342 (Page: 3).

[129] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. "The Anatomy of the Facebook Social Graph". In: *CoRR* abs/1111.4503 (2011) (Pages: 2, 3, 13).

[130] Mehul Nalin Vora. "Hadoop-HBase for large-scale data". In: *Proceedings of 2011 International Conference on Computer Science and Network Technology*. Vol. 1. Dec. 2011, pp. 601–605 (Page: 13).

[131] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. "Building a Replicated Logging System with Apache Kafka". In: *Proc. VLDB Endow.* 8 (Aug. 2015), pp. 1654–1655 (Page: 12).

[132] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. "GraphX: A Resilient Distributed Graph System on Spark". In: *First International Workshop on Graph Data Management Experiences and Systems*. GRADES '13. 2013, 2:1–2:6 (Pages: 12, 14).

[133] Eiko Yoneki, Amitabha Roy, and Derek Murray. *Systems and Algorithms for Large-scale Graph Analytics*. Technical Report: Report from Dagstuhl Seminar 14462, Nov. 2014 (Page: 3).

[134] Andre G. Antoniu M. Asch R. Badia Sala M. Beck P. Beckma Zacharov. "The BDEC " Pathways to Convergence " Report Toward a Shaping Strategy for a Future Software and Data Ecosystem for Scientific Inquiry". In: 2017 (Page: 130).

[135] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. 2010, pp. 10–10 (Pages: 12, 14).

[136] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. "In-Memory Big Data Management and Processing: A Survey". In: *IEEE Transactions on Knowledge and Data Engineering* 27.7 (July 2015), pp. 1920–1948 (Page: 13).

[137] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. "Tapestry: A Fault-tolerant Wide-area Application Infrastructure". In: *SIGCOMM Comput. Commun. Rev.* 32 (Jan. 2002), pp. 81–81 (Page: 2).

[138] Yu Zheng, Quannan Li, Yukun Chen, Xing Xie, and Wei-Ying Ma. "Understanding Mobility Based on GPS Data". In: *Proceedings of the 10th International Conference on Ubiquitous Computing*. UbiComp '08. 2008, pp. 312–321 (Page: 2).

Eidesstattliche Erklärung
laut §5 der Promotionsordnung vom 06.12.2013

Ich versichere an Eides Statt, dass die Dissertation von mir selbständig und ohne unzulässige fremde Hilfe unter Beachtung der „Grundsätze zur Sicherung guter wissenschaftlicher Praxis an der Heinrich-Heine-Universität Düsseldorf“ erstellt worden ist.

Ort, Datum	Stefan Nothaas