Improving Spark Performance with Zero-copy Buffer Management and RDMA

Hu Li*, Tianjia Chen†, Wei Xu‡
Institute for Interdisciplinary Information Sciences
Tsinghua University
Email: *lihu12@mails.tsinghua.edu.cn, †ctj2015@tsinghua.edu.cn, ‡weixu@tsinghua.edu.cn

***Abstract*—With the ever increasing demand on interactive data analytics, latency for big data frameworks becomes more important. We present our preliminary experience designing and implementing NetSpark, an improved Spark [1] framework that is highly optimized for network latency. Combining optimizations on data serialization, network buffer management with hardware-supported Remote Direct Memory Access (RDMA) technology, we show that we can eliminate most of the data copies from end to end, significantly reducing the Spark task running time. Our preliminary experiments show that NetSpark improves GroupBy operation in Spark by about 40% and the PageRank algorithm in GraphX by about 20% on a 10Gbps data center network over the legacy network stack.**

***Keywords*-Distributed Computing; Spark; RDMA; low latency;**

I. Introduction

With the ever increasing demand of data analytics, it is more and more important not only to be able to analyze the data, but to analyze a vast amount of data *interactively*. For example, real time web traffic analysis is becoming the key for the success of e-commerce sites. This interactive requirement is a much more challenging goal than the traditional Hadoop model that is optimized for throughput rather than latency.

In-memory computation frameworks make the interactive data analytics possible. Spark [1] is a widely used data processing system. In addition to all the easy-to-program data processing primitives, a key optimization of Spark is to store intermediate results in memory rather than disks. The in-memory computation improves system performance by removing the expensive overhead from disks and distributed file systems. Thus, Spark is flexible enough to implement different computation models on top, including GraphX [2], Spark SQL [3], and Spark Streaming [4]. These frameworks demand even lower latency for different reasons. GraphX supports graph analytics algorithms that require many iterations, and the running time of an iteration directly affects the job completion time. SparkSQL is designed to handle interactive queries, and shorter latency can greatly improve user experience.

The following main components determine how fast a task runs: scheduling, computation and network communication, The Spark community has proposed many ways to improve the per-task running time as we will review in Section V-C. However, the network latency becomes the major bottleneck preventing us from further reducing the job latency.

While Spark uses network in many ways, one operation uses network the most by far, the shuffle operation. The shuffle operation causes all Executors to exchange information among each other. In the Hadoop system, shuffle requires heavy disk operations and thus the network congestions only happen in rare cases like the incast problem. Spark is different in two ways: first, the small tasks produce lots of small data transfers among many nodes; second, memory cache replaced most disk accesses, making network latency the dominating bottleneck.

This paper presents our preliminary results on improving the network performance of Spark by improving its network stack performance. We also show some preliminary results on how this improvement helps real applications like GraphX.

In the case of Spark with small tasks, single node network stack performance is important. The latency bottleneck on a single node is severely affected by the kernel network stack performance. There is significant overhead to process a packet in the kernel. Sending a packet involves the virtual file system layer, the socket layer, IP layer before it gets to the Network Interface Card (NIC). Many of these processing layers require memory copies. Even worse, the packet handler needs to access many shared kernel data structures, causing lock contentions and context switches, and the situation only gets worse on multi-core machines. Many projects focus on improving data center network stack performance, and we review these approaches in Section V-A.

Remote Direct Memory Access (RDMA) takes even more aggressive on the hardware side. It allows the application to specify an application space buffer so that the network transfer can bypass the operating system kernel. RDMA is widely used in high performance computing interconnects such as the Infiniband fabric. The Infiniband does flow control and congestion control to provide a lossless fabric and thus people do not need a full TCP stack.

Traditionally RDMA is only available on Infiniband, limiting its applicability to commodity cloud computing clusters, the main platform that Spark runs on. Recent development on RDMA over Converged Ethernet (RoCE) enables RDMA to co-exist with Ethernet traffic [5].

However, adopting RDMA is still challenging, especially for a high level language like Java or Scala in which Spark is implemented. In particular, with RDMA, we bypass the kernel network stack, making it necessary for the application to manage many data structures (such as buffers) that the OS

978-1-4673-9955-5/16/$31.00 ©2016 IEEE

would have managed. Even worse, sometimes the automatic memory management in these language runtimes has negative impact on the RDMA performance. Last but not least, as RDMA reduces the network transfer time, any overhead related to buffer management becomes more obvious.

People have build RDMA libraries for Java Virtual Machines (JVMs). For example, we used the jVerbs [6] library from IBM. It is a wrapper of a native RDMA implementation through the Java Native Interface (JNI). As we will show in our evaluation, just porting to RDMA brings some improvement to the Spark performance, but not a significant one. The network buffer management, object serialization, as well as the garbage collection still brings significant overhead to the end-to-end network latency.

We design NetSpark, combing the application of RDMA with a number of Spark-specific improvements. Our improvements include application buffer management and serializing an object directly to the RDMA buffer. We show that these improvement is a necessary complements to RDMA, and together with these optimizations, we are able to improve the shuffle read performance by about 40% and the PageRank algorithm in GraphX by about 20%. Further more, NetSpark is fully compatible with off-the-shelf Spark 1.5. The users only need to set a single flag to enable RDMA mode. There is no change to the user program. We believe the compatibility is the key for its practical adoption.

We made two contributions in the work, and this paper presents the preliminary results:

- We propose a combination of memory management optimizations for JVM-based applications to take advantage of RDMA more efficiently and demonstrate these improvements in Spark.
- We build a reliable Spark package that runs on RoCE fabric, improving latency-sensitive task performance, while staying fully compatible with the off-the-shelf Spark.

The remaining of this paper is organized as follows: Section II provides background on Spark network buffer management and RDMA; Section III introduces the design and implementation of the memory management in NetSpark. Section IV provides the preliminary evaluations of NetSpark performance on a cluster of 34 servers. We review related work in Section V, and conclude in Section VI.

II. Background

In this section, we provide a brief introduction of the current Spark network buffer management, identifying its inefficiency. We also provide a brief introduction to the RDMA architecture and programming model for the readers who are not already familiar with topic.

A. Current Spark Network Buffer Management

In each Spark application, a Driver process creates a `SparkContext` data structure that holds the application state information. Each worker node creates Executor processes as the main process for task execution. The Executor creates a thread to execute each task within the same Executor process. Using thread reduces the overhead for task starting/stopping, making it efficient to run small tasks. We modify two components in the Executor: the BlockManager component for data management and the network transfer service for communications among workers.

During a data transfer process, such as in the shuffle operation, an Executor first serializes the object into a buffer, uses the Java network API to copy it from heap to off-heap, and then executes the system call to copy the off-heap buffer to the kernel TCP/IP stack. The buffer needs to be copied several times in the kernel before sent out from the NIC. On the receiver side, the buffer needs to be copied back to the Java application and deserialized into a Java object. Again, the process heavily involves memory copy. Allocating and deallocating the buffers inside the Java process cause frequent garbage collections, making the performance unpredictable.

B. RDMA and RoCE

RDMA has been widely used in high performance computing. RDMA works as follows: when initiating packets sending, the application pins a region of memory, and then let the NIC to register this physical memory region. It also puts the virtual memory to physical memory mapping into the page table. After registration, the NIC can directly access the registered memory without interrupting the CPU. The NIC maintains a complete queue for sending and receiving packet without a full network stack. Since the NIC can copy packets from user space into the NIC directly without CPU context switches, the latency in the RoCE is much lower than TCP [7].

RDMA offers two types of APIs. The first type is *one-side* `read`/`write` operations. In this type, network transfer does not involve the remote host CPU. The second type is *two-side* `send`/`receive` operations that requires CPU processing on the receiver. One-side operations offer smaller latency and more efficient CPU utilization than two-side operations [8] and thus two side operations are usually reserved for control messages only.

RDMA technology is designed for Infiniband network fabric, but recently people have developed RoCE (RDMA over Converged Ethernet) that runs on Ethernet, the most common fabric in data centers. RoCE use the Ethernet priority flow control (PFC) to guarantee that there is no packet loss in the data link layer. We need to enable the PFC and VLAN, and may also need Quality of Service (QoS) in configurations on the servers and switches [9]. To handle very busy network traffic with lots of server under a single top-of-the-rack switch, we need to adjust the switch buffer allocation so that the PFC buffer is large enough.

III. Design and Implementation

The key to take full advantage of RDMA is to optimize the application-specific buffer management. Figure 1 shows an overview of our modifications to the Spark Executor. The BufferManager is the center module in NetSpark, and it manages the data serialization and network communications. A RDMA-based network module that sets up and manages

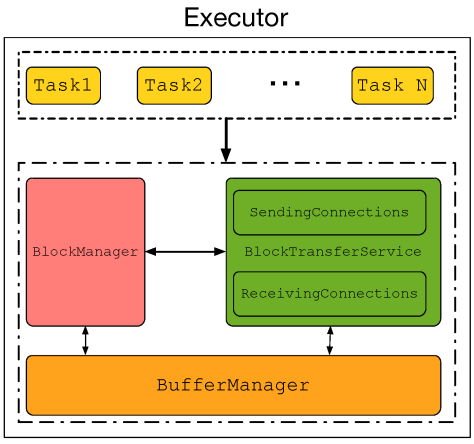

Fig. 1: Modified and new modules in Spark Executor. The BufferManager and RDMA network transfer module is new, and BlockManager module is modified.

the connections, as well as an object serialization library both request buffer from the BufferManager. The BufferManager maintains a set of off-heap, fixed-sized buffer blocks and use the same set of blocks for network sending / receiving and object serialization / deserialization.

In this section, we introduce the design and implementation of the buffer management that allows us to achieve zero copy from object serialization to send out a message, significantly reducing the network transfer time.

A. Network Buffer Management for RDMA

As we mentioned in Section II-B, we need to pin memory pages of the RDMA buffer in the main memory, preventing the operating system from paging them out. In a native process without automatic memory management, we can do so using the `mlock` system call. In a Java runtime, we need to allocate this buffer on the off-heap region. We can pin the off-heap memory pages. However, off-heap pages is relatively slow to allocate and release, and may increase latency if we perform lots of small allocations.

To eliminate this overhead, we have to pre-allocate some memory space as the RDMA buffer. The challenge here is how to determine the size of the pre-allocated buffer. Smaller buffer size means more management overhead, while a size too large means wasted memory spaces. The size of transfers in Spark, especially for the shuffle operations, vary significantly, making the size choice difficult.

One naive approach is to allocate a small number of large-enough fixed-size buffers on the off-heap space. After receiving a RDMA transfer (and hence the size is known), the program can dynamically allocate a buffer on the heap space with the right size, and copy the data from the off-heap RDMA buffer to heap space. The off-heap buffer is used for the transfer, acting like the kernel network buffer in TCP/IP model. Once the data is copied to the heap buffer, another transfer can reuse the off-heap buffer. After the application finish using the data, the on-heap buffer is subject to garbage collection. While this approach is simple, it requires one memory allocation and one memory copy, introducing extra garbage collection and heap management overhead. Those overhead increase when jobs get smaller, and when the network performance gets higher.

Another approach to avoid copying the data between heap and off-heap is to allocate a number of off-heap buffers with variable sizes. Some RDMA-based key value storage and RPC systems adopt this choice. For example, FaRM uses SLAB [10] to manage variable sized buffer allocations. The choice of SLAB is based on the assumption that the transfer size follows a specific distribution that can be estimated beforehand, without too many large messages. For example, in a key-value storage system, the message sizes are typically small with little variations in size [11]. However, the assumption is no longer true for shuffle traffic in Spark as the message sizes show significant variations.

Due to these considerations, we use a BufferManager to pre-allocate a number of fixed sized buffer blocks in the off-heap space, and pin all of them. These buffer blocks are used both for sending and receiving messages and they are shared across all connections. We choose the buffer block size to be 2MB(size of Huge page). 2MB is big enough to saturate 10Gb Ethernet, can be reclaimed in 2ms, and can reduce 40% of register time compared with normal memory page [12].

For each connection, we use two-side operation to exchange memory address and notify(we also optimised with inline and selective signaling[8]), and use the one-side read for data transfer. The BufferManager allocates all buffer blocks for the RDMA read operation. As the two-side operation is only used for control information, it is small and do not add much latency. The one-side operation for the actual transfers makes the overall data transfer efficient.

B. Object serilization/deserialization

Object serialization/deserialization may lead to extra memory copy / allocation overhead. In the traditional network stack, the network send/receive buffer is different from the buffer that is accessible by the application. During the deserialization process, one needs to make another copy of the buffer content to memory space separate from the network buffer. Doing so allows the system to reuse the network buffer space for new transfers, at the cost of an extra memory copy.

Thanks to the flexibility of our network buffer management, and combine network buffer with serilization/deserialization buffer, we can reuse the off-heap space for serialization/deserialization. We implemented the functionality by providing custom `InputStream` and `OutputStream` classes. The internal buffer in `InputStream` and `OutputStream` is a list of buffers allocated by the BufferManager. The `OutputStream` automatically segment the data into multiple buffer blocks, so there no RDMA buffer size tuning in our design, only some messages may waste some memory(at most 2MB). Using this interface, we do not need to change the other parts of the Executor to move the buffer off-heap too.

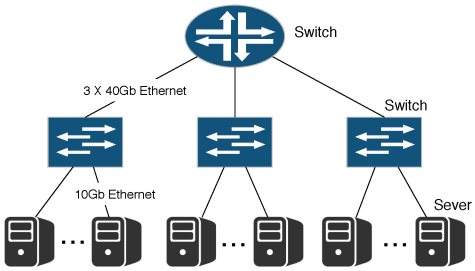

Fig. 2: Network topology of our testbed.

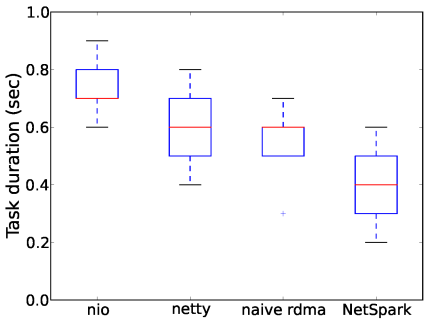

Fig. 3: Time to complete the reduce stage of a *GroupBy* operation using the four different implementations, using the small dataset. For each implementation we plot min/max completion time as well as the 25th, 50th, and 75th percentile completion time among all the tasks.

During a large RDMA transfer, the buffer block is a transferring unit. The receiver explicitly acknowledges each block received. On receiving the ACK from receiver, the sender releases ACK'ed buffer block and returns it to the BufferManager.

On the receiver side, the BufferManager passes a received buffer block to the `InputStream`, and the application can deserialize the object directly from the buffer. After a buffer block is read and deserialized, it is immediately released back to the BufferManager, without waiting until the object is completely deserialized. This early buffer release also reduces the memory utilization during a large transfer.

By combining the BufferManager with the Input/Output stream interface compatible to the original Spark implementation, we can eliminate the memory copy while keeping the changes to Spark Executor minimal.

IV. Evaluation

A. Experiment setup

Topology is very important for networking performance. Different from many other projects that use Infiniband with HPC topologies such as hypercubes, we evaluate our approach on a typical data center networking topology as Figure 2 shows. Servers connect to the top-of-racks (TOR) switches with 10GE passive copper links, and each TOR connects to the aggregate switch using three 40GE links. Each TOR connects to 11 or 12 servers and we have 34 servers in total. We believe this topology represents a typical cloud computing cluster.

Our TOR switches are all Arista 7050 and the aggregate switch is a Mellonax SX1710 running in Ethernet mode, and we adjusted the PFC buffer size on the Arista switches to 37024 bytes. We have 34 servers each contains two 6 core Intel Xeon CPUs with 24 threads and 128GB memory. The servers connect to the network using a Mellonax ConnectX3 ethernet NIC with latest drivers from the manufacture. We run Spark 1.5.

The off-the-shelf Spark offers two network transfer protocols, JAVA NIO and Netty [13](But the NIO transfer will soon be deprecated). We compare network performance among these four implementations: the NIO, the Netty, a naive RDMA implementation and our NetSpark implementation. Both NIO and Netty are over the default Linux network stack. The naive implementation is similar to [14], which copies the data buffer to RDMA buffer, and will introduce extra copies compared with NetSpark.

B. GroupBy

GroupBy is an important operation that is heavily utilized in various tasks such as evaluating SQL queries and Map Reduces. *GroupBy* operation is composed of one *map* stage and one *reduce* stage, and involves lots of data to shuffle. The *GroupBy* performance is the key to improve SparkSQL performance. Thus, we first use *GroupBy* as a micro-benchmark to evaluate the improvement by NetSpark. Since Spark does not perform network transfers in the map stage, we only plot the result of reduce stage here. We have confirmed that the performance of map stage is the same across all four different implementations.

For the first micro-benchmark on latency, we keep the task duration about one second, similar to the choice of latency sensitive queries [15]. The total amount of data shuffled is about 2.5GB for the reduce stage. Figure 3 gives the comparison results of the running time. The reduce stage is composed of 792 tasks,and each bar presents the min/max task completion time as well as the 25th, 50th, and 75th percentile value among all the 792 tasks.

Netty, as a heavily optimized network library, achieves a good improvement over the legacy NIO implementation. The naive RDMA provides a slightly faster task completion time than Netty. NetSpark achieves a 17% improvement over the naive RDMA for the max, and about 33% for the 50th, showing the effectiveness of our network buffer management scheme.

To figure out the cause of this performance improvement, we plot the time taken for CPUs to wait for the network data in each task in Figure 4. The result shows that NetSpark lowers the CPU wait time significantly, and this is the reason why we can lower the task completion time over the network with the same 10GE bandwidth.

We also evaluated the tasks completion time(or the reduce stage cost time) using a much larger dataset of about 107.3GB,

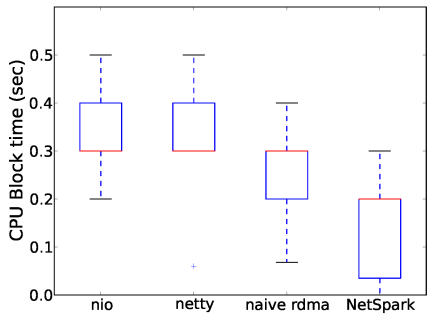

Fig. 4: CPU block time during a *GroupBy* task using the four different implementations.

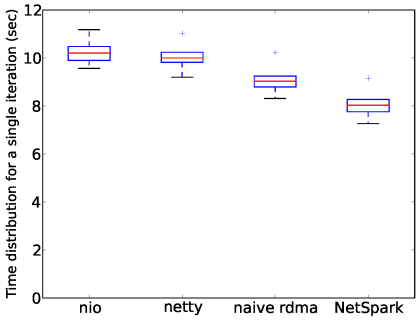

Fig. 6: Time taken to complete one iteration in the PageRank algorithm.

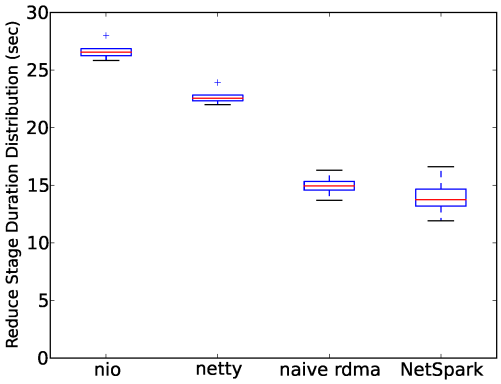

Fig. 5: *GroupBy* performance on the large dataset. Otherwise the figure is same as Figure 3.

which means each node processes about $43\times$ more data than the previous experiment. We repeated the experiment 100 times, Figure 5 shows the comparison for the tasks completion time. The result shows that NetSpark is about 40% faster in the reduce stage than Netty. This is because with a larger dataset, each Executor runs multiple tasks, and the improvements for these tasks add up to a more significant speed up.

C. PageRank

In addition to the *GroupBy* micro-benchmark, we also evaluate NetSpark on the PageRank algorithm in GraphX [2]. We used the Twitter graph [16] as our test data. This dataset contains approximately 41 million nodes and 1.5 billion edges. The algorithm involves multiple iterations, and we also repeated 100 times. Figure 6 shows the running time distribution for a single iteration. Similar to the micro-benchmark, we can see a 20% improvement over Netty and about 9.5% over the naive RDMA implementation.

V. Related Work

A. Improving data center networking performance

Data center network is significantly different from the traditional enterprise network or the high performance computing interconnects. Due to cost concerns, the data center network is usually slower than the HPC interconnects. The typical data center network nowadays is still at 10GE and gradually moving to 25GE, while HPC has implemented 56Gbps or 100Gbps Infiniband network.

However, data center network is much faster both in latency and throughput than enterprise networks, and thus many assumptions based on such networks no longer apply. For example, DCTCP [17] changes the TCP assumptions to reduce the delays and throughput problems caused by occasional congestions. [18] improves large data transfers using application layer protocols such as a multicast tree.

With the latency in a data center network gets lower, many projects target lowering the operating system overhead to send/receive messages, especially on a multi-core machines with vast data processing power. Megapipe [19] improves the TCP/IP network stack performance on multi-core machines through a novel API design, while mTCP [20] moves the entire TCP/IP stack to the user space to reduce the number of memory copies and lock contentions. Modern network interface cards (NICs) provide hardware functionalities to improve the network performance. For example, Affinity-Accept [21] uses Receive Side Scaling (RSS) to provide a local accept queue for each CPU core, and [22] utilizes TCP offloading to reduce CPU utilization.

B. Non-HPC RDMA applications

Many projects focus on using RDMA to accelerate the key-value storage systems. [23], [24] show how to lower CPU utilization by migrating to RDMA. Projects [7], [8], [25], [26] focus on improving the latency using RDMA. More recent storage systems use RDMA with advanced consensus protocols to guarantee certain level of data consistency, while keeping the low latency and low CPU feature of RDMA [27], [28], [29].

There is a major difference between key-value storage and computation in terms of network workload and buffer management. In a storage system, the data are not processed and can stay in the serialized form. In contrast, in a computation system like Spark, all data are computed and thus the life

time of a buffer is just the network transfer time, adding extra memory allocation and copy overhead. Also as we mentioned previously, the transfer size distribution varies vastly in a data processing system.

There are also projects focusing on improving data processing system performance. For example, [30] uses RDMA to accelerate Hadoop, and [31] accelerates graph computation. [32] also tries to accelerate Spark using RDMA over Infiniband network. However, it focuses on reducing the throughput loss from the IP-over-IB layer, rather than optimizing for network latency. We show that with our buffer management optimization, we can achieve almost the same task completion time on a 10GE network as their 32Gbps high performance Infiniband fabric, using similar *GroupBy* benchmark.

C. Improving Spark Latency

Many people have realized large-scale data analytics frameworks are shifting towards shorter task durations and larger degrees of parallelism to provide low latency [15]. This is especially true for SQL systems [3], [33] and streaming systems[4], [34]. People have build systems to reduce the latency for scheduling lots of small tasks [15], and even introduce hardware accelerators to reduce task completion time [35]. We believe as the task running time becomes shorter, our reduction for the network performance will be even more crucial for the overall system performance.

VI. Conclusion and Future work

In this paper we present our preliminary results on designing and implementing NetSpark that reduces Spark network latency by leveraging the RoCE network fabric. We show that to take full advantage of RDMA, we need to perform application specific buffer management, in order to eliminates all unnecessary memory copies through the entire network stack. We evaluate our system on a real cluster with 34 nodes and the preliminary evaluation results shows 20% improvements in both micro-benchmark and task completion times using real Spark workloads.

As future work, we would like to further improve Spark performance by leveraging the remote memory accessible through RDMA to perform data shuffles and avoid involving local disks for the shuffle processes. We will further investigate the interactions between RDMA and TCP traffic with in the same Ethernet fabric, especially under high network utilizations and network visualizations. We would also like to apply our RDMA-specific memory management framework to other systems that involve a mixture of data transfers and computation, such as real time monitoring and log processing systems.

VII. Acknowledgements

This research is supported in part by the National Natural Science Foundation of China Grants 61033001, 61361136003, 61532001, China 1000 Talent Plan Grants, Tsinghua Initiative Research Program Grants 20151080475, a Microsoft Research Asia Collaborative Research Award, and a Google Faculty Research Award.

References

[1] Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *NSDI*, 2012

[2] Gonzalez, Joseph E, et al. "Graphx: Graph processing in a distributed dataflow framework." *OSDI*, 2014.

[3] Armbrust, Michael, et al. "Spark SQL: Relational data processing in Spark." *SIGMOD*, 2015.

[4] Zaharia, Matei, et al. "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters." *HOTCLOUD*, 2012.

[5] Kissel, Ezra et al. "Efficient wide area data transfer protocols for 100 Gbps networks and beyond." *SC*, 2013.

[6] Stuedi, Patrick, et al. "jVerbs: ultra-low latency for data center applications." *SoCC*, 2013.

[7] Dragojevic, Aleksandar, et al. "FaRM: Fast remote memory." *NSDI*, 2014.

[8] Kalia, Anuj, et al. "Using RDMA efficiently for key-value services." *SIGCOMM*, 2014.

[9] Ophirmaor, How To Run RoCE and TCP over L2 Enabled with PFC. https://community.mellanox.com/docs/DOC-1415

[10] Bonwick, Jeff. "The Slab Allocator: An Object-Caching Kernel Memory Allocator." *USENIX* Vol. 16. 1994.

[11] Atikoglu, Berk, et al. "Workload analysis of a large-scale key-value store." *SIGMETRICS*, 2012.

[12] Frey, Philip Werner and Gustavo Alonso. "Minimizing the Hidden Cost of RDMA." *ICDCS*, 2009.

[13] Netty, http://netty.io/

[14] Barthels, Claude, et al. "Rack-Scale In-Memory Join Processing using RDMA." *SIGMOD*, 2016.

[15] Ousterhout, Kay, et al. "Sparrow: distributed, low latency scheduling." *SOSP*, 2013.

[16] Kwak, Haewoon, et al. "What is Twitter, a social network or a news media?" *WWW*, 2010.

[17] Alizadeh, Mohammad, et al. "Data center tcp dctcp." *ACM SIGCOMM computer communication review*, 2011.

[18] Raiciu, Costin, et al. "Improving datacenter performance and robustness with multipath TCP." *SIGCOMM*, 2011.

[19] Han, Sangjin, et al. "MegaPipe: A New Programming Interface for Scalable Network I/O." *OSDI*, 2012.

[20] Jeong, EunYoung, et al. "mTCP: a highly scalable user-level TCP stack for multicore systems." *NSDI*, 2014.

[21] Pesterev, Aleksey, et al. "Improving network connection locality on multicore systems." *EuroSys*, 2012.

[22] Mogul, Jeffrey C. "TCP Offload Is a Dumb Idea Whose Time Has Come." *HOTOS*, 2003.

[23] Mitchell, Christopher, Yifeng Geng and Jinyang Li. "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store." *ATC*, 2013.

[24] Stuedi, Patrick, et al. "Wimpy Nodes with 10GbE: Leveraging One-Sided Operations in Soft-RDMA to Boost Memcached." *ATC*, 2012.

[25] Wang, Yandong, et al. "C-Hint: An Effective and Reliable Cache Management for RDMA-Accelerated Key-Value Stores." *SoCC*, 2015.

[26] Panda, Dhabaleswar K, et al. "High-Performance Design of HBase with RDMA over InfiniBand." *IPPS*, 2012.

[27] Ousterhout, John, et al. "The case for RAMClouds: scalable high-performance storage entirely in DRAM." *ACM SIGOPS Operating Systems*, 43.4 2010: 92-105.

[28] Dragojevic, Aleksandar, et al. "No compromises: distributed transactions with consistency, availability, and performance." *SOSP*, 2015.

[29] Wei, Xingda, et al. "Fast in-memory transaction processing using RDMA and HTM." *SOSP*, 2015.

[30] Lu, Xiaoyi, et al. "High-Performance Design of Hadoop RPC with RDMA over InfiniBand." *ICPP*, 2013.

[31] Lin, Haoxiang, et al. "GraM: scaling graph computation to the trillions." *SoCC*, 2015.

[32] Lu, Xiaoyi, et al. "Accelerating Spark with RDMA for Big Data Processing: Early Experiences." *HOTI*, 2014.

[33] Kornacker, Marcel, et al. "Impala: A Modern, Open-Source SQL Engine for Hadoop." *CIDR*, 2015.

[34] Apache Storm, https://storm.apache.org/

[35] Li, Peilong, et al. "HeteroSpark: A heterogeneous CPU/GPU Spark platform for machine learning algorithms." *NAS*, 2015.