2021 IEEE International Conference on Cluster Computing (CLUSTER) | 978-1-7281-9666-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/Cluster48925.2021.00033

Thinking More about RDMA Memory Semantics

Teng Ma[*,†], Kang Chen[†], Shaonan Ma[†], Zhuo Song[*], and Yongwei Wu[†]
[†]Tsinghua University, {*mt16,msn18*}*@mails.tsinghua.edu.cn,* {*chenkang, wuyw*}*@tsinghua.edu.cn*
[*]Alibaba, *songzhuo.sz@alibaba-inc.com*

***Abstract*—RDMA (Remote Direct Memory Access) provides memory semantics to access the remote memory directly bypassing remote CPUs. It can provide low latency and high throughput that can benefit many data center applications. Though a lot of efforts had been made in the literature, this paper tries to find more opportunities to boost the performance of memory semantic operations in the RDMA network. Similar to the optimizations for local memory operations, we find that the performance can be improved in the RDMA network after considering the vector IO mechanism, the performance asymmetry between sequential and random access, IO consolidation, NUMA effects, as well as the atomic operations (such as compare and swap) provided by the underlying hardware. We have done a comprehensive empirical study on the influences from these factors for the memory semantic operations in RDMA network and provide guidelines to improve applications. Experimental results show that four typical applications, disaggregated hashtable, distributed shuffle, distributed join, and distributed log are improved by 2.7×/5.8×/5.3×/9.1× respectively after considering memory semantics related optimizations.**

I. Introduction

RDMA network is now widely used in modern data centers. Such network infrastructures are widely deployed in cloud platforms including Microsoft [20], Alibaba Cloud [34] and other ISPs to reduce the heavy CPU loads. RDMA network can support both *one-sided* verbs (`RDMA Write/Read/Atomic`) and *two-sided* verbs (`RDMA Send/Recv`). *One-sided* verbs are *memory semantic operations* because they can access the remote memory directly, similar to normal local memory loads and stores. They can even bypass remote CPUs, i.e., accessing the remote memory without remote CPU involvement. *Two-sided* verbs have similar interfaces as traditional message passing mechanisms, and remote CPUs are constantly interrupted to process network messages. According to prior studies [55], one-sided RDMA verbs (memory semantic operations) have at least two advantages: 1) higher performance than two-sided RDMA in terms of both throughput and latency, and 2) bypassing kernel and avoiding the involvement of CPU on the receiver side, effectively reducing the remote CPU overhead.

Because the memory semantic operations can provide high bandwidth and low latency, it is adopted as the fundamental building blocks [2] for many current data center applications. FaRM [15] is a memory distributed computing platform that adopts one-sided RDMA. Herd [23] presents an RDMA-based key-value store, which provides many useful optimization techniques from the aspect of network. InfiniSwap [19] manages remote memory with a back-end allocator to gain high utilization, and memory semantic RDMA makes it possible that different functionalities can be distributed to different kinds of blades.

Though the one-sided RDMA model boosts many cloud applications, most of them use RDMA network as a fast data transfer method. For instance, based on RoCE, DCQCN [58] proposes a novel window-based algorithm implemented in the NIC to mitigate network congestion. However, we find out that we can also **consider the RDMA network as part of the memory system**. Many optimizations for local memory are also applicable for remote memory. Specifically, this paper investigates the following aspects to improve the memory semantic operations in RDMA network.

Hardware-assisted vector IO (Section III-A). The standard POSIX programming APIs provide the vectorized version of read/write, i.e., `readv` and `writev`. They are used to copy vectorized data from/to the IO system. Despite the convenient programming model of transferring multiple blocks of data, the performance can also be improved by batching the data and reducing the system call overhead. In the hardware level, RDMA-enabled NICs (RNICs) are attached to PCIe buses, and the underlying PCIe buses can transfer the vectorized data. Thus, the vector IO can also benefit the memory semantic operations in RDMA network.

Performance asymmetry between sequential and random memory access (Section III-B). In local memory, the sequential write is nearly 2.92× and 6.85× faster than random write and inter-socket random write (NUMA effects as mentioned below). Base on this observation, Polymer [57] can bring about 1.58∼53.09× improvement in graph processing. As the destination of RDMA network is remote memory, and RDMA network has low latency and high bandwidth, such asymmetry also exists and is visible in RDMA supported system.

IO consolidation (Section III-C). Transferring very small amount of data such as 1 to tens of bytes (e.g., 64 bytes) can not fully utilize the underlying bandwidth of the network, and incurs a phenomenon called Packet Throttling (Section II-B1). Thus, it is meaningful to consolidate multiple small IO requests in the same data block and make a single RDMA request at once instead of multiple requests. This will effectively reduce the number of data transfer operations.

NUMA effects (Section III-D). Many local optimizations must rely on NUMA (Non-uniform Memory Access) effects [27]. For the RDMA network nowadays, it is common to attach multiple NICs in one server (for the reliability

Teng Ma, Kang Chen, Shaonan Ma, and Yongwei Wu are with Department of Computer Science and Technology, Beijing National Research Center for Information Science and Technology (BNRist), Tsinghua University, China

978-1-7281-9666-4/21/$31.00 ©2021 IEEE
DOI 10.1109/Cluster48925.2021.00033

and performance reasons), and each NIC will be assigned to a distinct socket. NUMA effects also exist in the RDMA network.

Hardware provided atomic operations (Section III-E). For multi-thread/process coordination, we need atomic operations such as `compare and swap` and `fetch-and-add`. The RNICs can provide similar instructions as local CPUs. For certain workloads, it is better to use such hardware facilities instead of achieving the same goal in a software approach.

We have done a comprehensive empirical study on the above aspects of memory semantic operations in RDMA network. Observations verify that we can actually use similar optimizations for local memory to be used in remote memory. To this end, we use four typical applications, including disaggregated hashtable [3], distributed shuffle algorithm, distributed join algorithm, and distributed log (Section IV), and then modify them taking the considerations on the above aspects. The improvement is promising. To summarize, this paper makes the following contributions.

- We conduct an empirical performance analysis by evaluating the performance of *memory semantic* RDMA from the above aspects using various benchmarks.
- Using four representative case studies, we show the power of the remote memory access design choices outlined in our discussion. Our experiments on a cluster with eight machines show that four applications can achieve 2.7×/5.8×/ 5.3×/9.1× speed-up.

II. Background

A. RDMA Preliminaries

Memory Semantics. RDMA has two types of verbs: *memory semantic* verbs (one-sided) and *channel semantic* verbs (two-sided). Memory semantic verbs provides `RDMA Write/Read/Atomic` (`compare and swap/fetch and add`) to directly access remote memory without remote side CPU's involvement. Channel semantic verbs provide `RDMA Send/Recv`, similar to traditional message passing model. Both sides need to get involved during the communication. Because memory semantic verbs post requests directly to the remote NIC queue bypassing remote kernel and CPU, such verbs have lower latency and high throughput than channel semantic verbs. This paper focuses on memory semantic verbs.

Transportation Type. RDMA supports three transportation types: Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD). All transportation types support the channel semantics. For memory semantics, `RDMA Write` is supported by RC and UC, but `RDMA Read` and `RDMA Atomic` are only supported by RC. RNIC hardware is responsible for the reliable delivery of each packet in RC. Thus, we only discuss the RC mode.

RDMA Programming. In the programming level, queue pairs (QP) are used. RNIC communicate with each other using queues. There are three types of queues: send queue (SQ), receive queue (RQ) and completion queue (CQ). SQ and RQ

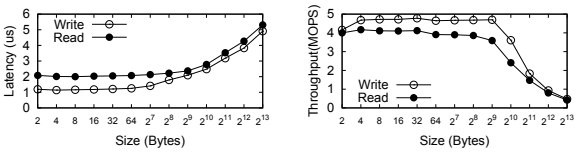


Fig. 1: Packet Throttling.

are always grouped and managed as a QP. We can post a work request (WR) by generating a work queue entry (WQE) into the work queue (SQ or RQ). Once a request is completed, the hardware posts a completion queue entry (CQE) into CQ. SQ and RQ can use distinct CQs or share a common CQ. Despite the queues, memory regions (MR) are created and attached to a QP. Remote protection keys (rkey) are used for a QP to access remote memory.

B. Hardware Features related to RDMA Memory Semantics

1) Packet Throttling: As with other communication mechanisms, there is a limitation on the packet send rate (called as *packet throttling*) in RDMA network. The access latency (the time to transfer certain amount of data to destination) for small payload sizes is kept steady. This phenomenon is also observed by other researchers [50], [33], [7]. As in Figure 1, the access latency of `RDMA Write` and `RDMA Read` increases from 1.16/2.00μs to 1.79/2.22μs, and the throughput is nearly identical (around 4.7/4.2MOPS) when the payload sizes are small than 256 bytes. The access latency increases rapidly when the payload sizes increase from 2KB. The reason is that the performance of RNIC is limited by both *link bandwidth* and *execution unit throughput*. Batching can alleviate the packet throttling for small packets.

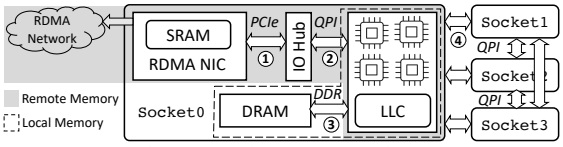


Fig. 2: Hardware Connections.

2) RNIC Metadata Cache: Each RNIC has limited on-device SRAM to cache necessary metadata including (1) *address translation table*, (2) *QP information*, and (3) *other metadata caches* [23], [53], [7]. The SRAM size is quite small due to the cost and space considerations. Usually, the commonly used commercial RNICs (e.g., ConnectX-3/4) have in the scale of megabytes [53], [41]. Such small sizes inevitably influence the performance from various aspects.

For the address translation table, on the translation misses, RNIC fetches the translation entries from DRAM and replace previous entries in SRAM. It is not surprise that with large number of MRs, the performance will degrade greatly. We use $10\times$ MRs, the access latency of 32 bytes drops about 60%.

The small size SRAM also influences connection performance in the scenario with large number of connections. More connections needs more QPs (as well as associated MRs). Chen et al. [7] observe the throughput of file system operations

(`Stat` and `ReadDir`) decreases by almost 50% when the number of clients increases from 40 to 120. SRAM can only hold a subset of QP information. When some QP information is missing, loading/unloading QP information between SRAM and DRAM brings a lot of overhead.

Above all, the limited capacity of SRAM in RNIC is the root cause of poor scalability issues [53]. To some extent, such cache architecture for metadata is similar to the CPU LLC.

3) CPU-RNIC Communication: RNICs are connected to on-board PCIe buses. Each RDMA operation issues one of three kinds of PCIe transaction layer packets (TLP) [26]: read request, write request, read completion. The communication between CPU and RNIC is initiated using memory mapped IO (MMIO), and the data transfer is through directly memory access (DMA). PCIe buses have a special operation mode to improve the data transfer for buffers with scattered data located in multiple data addresses. Scatter/Gather DMA [11] is proposed to deal with this situation, and it is supported by most commercial PCIe. Harnessing this underlying mechanism can benefit the transfer of small scattered data.

4) Inter-socket Communication: Current commercial CPUs (e.g., *Intel Xeon*) usually adopt the NUMA architecture, which consists of several *sockets*. Under NUMA, a processor (plugged in one socket) can access local memory much faster than non-local memory attached to some other processors (plugged in other sockets). Figure 2 illustrates the data access paths between sockets. Non-local accesses bring about 40%∼150% [57] more latency depending on the number of hops reaching destination memory module.

Such scenario also exists in RDMA network. It is quite common that multiple RNICs will be used in a single server. Each RNIC is associated with one socket [46]. Thus, similarly, RNIC can access the remote memory attached to the same socket as its corresponding RNIC in the remote machine faster than remote memory associated with other sockets (also see the data paths in Figure 2).

III. Observations on Remote Memory Access

To explore the memory semantic features of RDMA, we use a cluster based on InfiniBand for our evaluation. The cluster contains eight machines, each of which is equipped with dual-socket 8-core CPUs (*Intel Xeon* E5-2640 v2, 2.0 GHz), 20MB L3 cache, 96 GB memory space, and a Mellanox ConnectX-3 *Dual-Port* InfiniBand NIC (MT27500, 40 Gbps). The CPU has two sockets (named as socket 0 and socket 1) and the memory is equally allocated to each socket. The RNIC is installed over the PCIe link which belongs to the socket 1. An 18-port Mellanox InfiniScale-IV switch connects all of these machines. The machines run MLNX-OFED-LINUX-4.2-1.0.0 driver provided by Mellanox for Ubuntu 14.04.

In the following sections, based on RDMA verbs, we implement related benchmarks[1] to explore the memory semantics and case studies to show how to optimize applications with memory semantic features.

[1]The evaluation results are averaged over ten runs.

A. Vector IO

In local memory systems, vector IO is an efficient approach to accelerate the transmission bandwidth of PCIe [11]. When using vector IO, a single procedure call will read data from multiple buffers and coalesce them into a single piece of data, or reads data from a single data and writes it to multiple buffers. With vector IO, the number of system calls for issuing multiple memory read/write operations are reduced significantly. In most cases, this can improve IO performance linearly. Similar to local memory access, remote memory also supports vector IO. Several techniques in Algorithm 1 can be classified as vector IO in the RDMA scenario. Note that, only the data of small size can benefit from vector IO since large-size data will be bound by the bandwidth of RNIC (see Section II-B1).

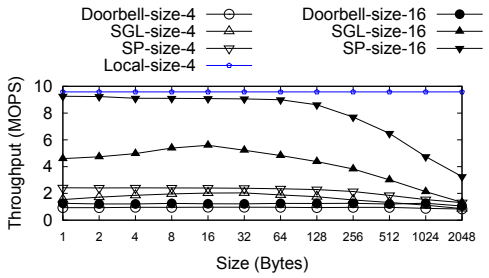


Fig. 3: Comparisons between three batch strategies, size-n means batch size is n. (We use one-to-one connection here.)

SP: SP stands for redesigning the Software Protocol. In SP, we exploit the packet throttling as defined in Section II-B1. The working thread copies multiple different pieces of address data to a temporary user-level buffer. After that, the buffer will write to remote memory as one WR. Due to the packet throttling for small payload sizes, if these buffers are small enough, the latency will decrease from N RTTs to a little larger than one RTT. However, this mechanism will consume extra CPU resources for gathering multiple pieces of data.

Doorbell: Kalia et al [26] introduce a doorbell batch mechanism based on PCIe. Multiple working requests can post to RNIC with the benefit of reducing CPU-generated MMIOs to only one. Accordingly, it can reduce the latency of transmission trips between RNIC and memory. But Doorbell cannot reduce network round trips (i.e., RDMA operations).

SGL: Scatter/Gather List. An SGL in the WR can coalesce multiple different pieces of address data and send them to one remote address via only one RDMA operation [44], [14] — accordingly, only one MMIO and one DMA are required. Notably, the PCIe supports transferring multiple buffers (i.e., scatter/gather list) at once. SGL is the most similar approach to vector IO of local memory. Compared with SP, it assigns the task of buffers gathering work to RNIC, not the CPU.

Figure 3 shows the performance trend of these three batch mechanisms with increasing payload sizes (batch sizes are 4 and 16 separately). To extract the irrelevant influence factors,

Algorithm 1: Three Batch Algorithms

```
   /* SP                                                    */
1  for buffer in buffers do
2  |  memcpy(buffer, tmp_buffer);
3  end
4  wr = gen_rdma_wr(&tmp_buffer);
5  rdma_write(wr);
   /* Doorbell                                              */
6  for buffer in buffers do
7  |  wr = gen_rdma_wr(buffer);
8  |  wr_list.push(wr);
9  end
10 rdma_write(wr_list);
   /* SGL                                                   */
11 for buffer in buffers do
12 |  sgl.push(buffer);
13 end
14 wr = gen_rdma_wr_with_sgl(sgl);
15 rdma_write(wr);
```

we use a one-client to one-server model. When the payload size is below 128 bytes, the test cases share a similar pattern that remains almost unchanged. After that, the exception is the Doorbell case which remains still, while the others turn into a linearly decreasing trend as the payload size grows.

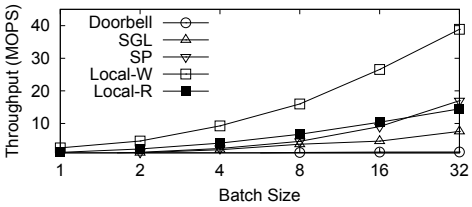


Fig. 4: Comparisons between different batch sizes.

Figure 4 illustrates the impact of different batch sizes on the throughput in different vector IO mechanism. To compare with local vector IO operation, we add the evaluations of batching local memory write/read operation by using Linux `readv/writev` APIs. The results show great scalability in both SP and SGL. As an opposite, Doorbell suffers little improvement (153%) when the batch size grows from 1 to 32. In particular, the throughput of SP grows 1.11×∼2.14× faster than SGL and 1.16×∼13.37× compared with Doorbell. Another observation is that batching remote memory access scales similarly to batch access to local memory. For instance, the highest throughput of SP (batch size is 32) can reach nearly 44%/117% of the local memory write/read.

TABLE I: Comparisons between three vector IO mechanisms.

Type	Programmability	Performance	Scalability
DoorBell	Good	Low	Poor
SP	Poor	High	Good
SGL	Moderate	High	Good in a small range

Figure 5 shows the performance with different thread number, and the thread number is from 1 to 8. The rank in throughput is aligned with Figure 3 and 4, showing SP, SGL, and Doorbell according to the order from high to low. The throughput of SP is 1.05×∼1.20× higher than SGL and 2.21×∼4.47× higher than Doorbell. Apart from this, changing the number of threads performs little influence on the throughput of SP and SGL. For instance, the throughput of SGL drops from 2.27MOPS to 1.69MOPS when the number increased to 8, with only a 25% reduction. On the contrary, the throughput of Doorbell decreases around 60% from 1 thread to 8 threads.

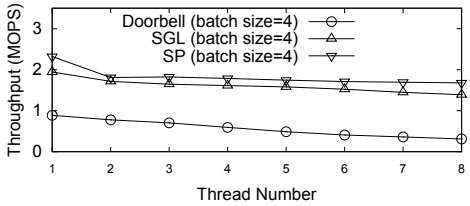


Fig. 5: Per-thread throughput with increasing thread number. The payload size is 32 bytes.

Discussion. Table I shows the comparisons for the three batch methods evaluated, and we explain the details from the following three aspects.

Programmability: For Doorbell, the required code changes are minor, and it is only required to rewrite a few lines of code. Although SP has a higher IOPS, its programmability is limited and inflexible. Especially, SP needs extra configurations on the software layer and hence requires heavy re-implementation of applications case-by-case. Furthermore, it can just scatter or gather multiple buffers on the one side while using remote memory APIs. SGL has similar issues, and only the send/recv semantics can support scatter and gather simultaneously. Thus, even one of the vendors has a plan to extend its protocols [37], processing logic on the remote memory side has to be redesigned. The major difference between SP and SGL is that in SP, the local memory side gathers data with CPU involvement. Thus, SP has the most complex programming logic and requires reconstructing applications.

Performance: In SGL, the RNIC is responsible for gathering multiple pieces of data. On the other hand, while using SP, the CPU will collect the data in a temporary buffer. Both SGL and SP require to post an RDMA operation at once, which means only one network round trip. Distinct from the other two methods, Doorbell only reduces the overhead from PCIe transmissions (i.e., MMIO). Thus, the Doorbell mechanism shows a relatively low throughput, and it could not fully utilize the potential of RDMA since the performance is almost independent of the tested parameters. SP and SGL have better performance than Doorbell. In particular, SP has the highest throughput since it stresses the host's memory bandwidth than the PCIe bandwidth. Besides, compared with SGL, the performance of SP is insensitive to changing the batch size, and we attribute this to the packet throttling.

Scalability: The scalability is captured in Figure 4 and Figure 5, Doorbell exhibits poor scalability with regard to the

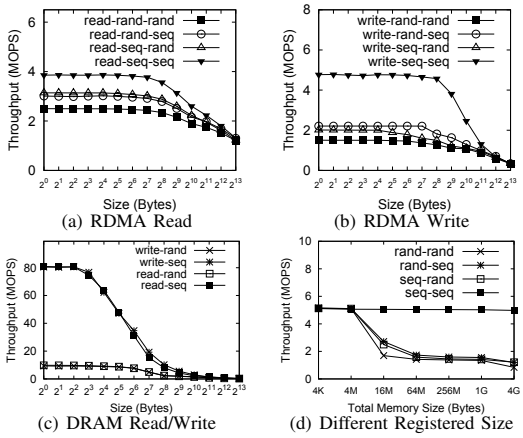
change of thread number and batch size. Our evaluation results coincide with prior works [26]. The reason for low scalability is that the PCIe transmissions time only constitutes a tiny fraction of RDMA operation processing. Another observation is that the performance of SGL will degrade seriously with an increased payload size and therefore the high performance only exists in a small range (less than 512 bytes).

B. Random and Sequential Access

As for local memory access, Figure 6(c) shows the difference between random/sequential read/write. In modern CPU cache architecture, once a row is read out, all the bits are available in the cache without having to perform another read-out. Therefore accessing the memory within row by row (i.e., sequential access) would be cheaper than another full cycle (i.e., random access). The cache module brings this benefit of the sequential memory access in computer architecture. From the perspectives of remote memory, this brings up the question of whether this benefit still exists.

Figure 6(a) and 6(b) display the performance of remote memory access via RDMA in both sequential and random pattern. We test four different access patterns with the payload size growing from 1 byte to 8192 bytes. If the access pattern is random, it will choose a random address in the RDMA-enabled memory (we fix it to 2GB) as the destination or source. The RDMA-enabled memory is allocated from system via `posix_memalign`. We could see at a glance that the sequential access pattern outperforms random memory access, especially in the write operations — the test case of choosing sequential address in both source and destination sides is more than 2× faster than others. However, as shown in Figure 6(d), if the size of RDMA-enabled memory is less than 4MB, the performance difference between sequential and random is little (less than 1%). This observation illustrates that the SRAM in RNIC could act as the cache in a memory access module to speed up the throughput of remote memory read and write operations. For both of the read and write operations, the trends in different test cases remain almost stable when the payload size is between 1 byte and 512 bytes. If the payload size exceeds 512 bytes, the throughput would be dropped gradually. It occurs due to the bandwidth saturation.

Discussion. A high in-chip cache miss ratio is the root cause of the asymmetry between random and sequential memory access. Typically, random remote access will trigger the miss of the translation table (virtual address to physical address) in the SRAM. Then it incurs frequent in and out of the cache in the translation table as we mentioned in Section II-B2. The asymmetry between the random and sequential patterns of remote memory is smaller compared with the local memory, whose number is 4×∼8×. The reasons are 1) there are multi-layer (typically, three) caches in local memory, but only one layer in remote memory, and 2) SRAM in the RNIC only caches the translation table, not the data. In conclusion, developers should avoid remote random access in the application design of remote memory.



(a) RDMA Read (b) RDMA Write (c) DRAM Read/Write (d) Different Registered Size

Fig. 6: Comparisons between sequential and remote memory random access with different payload size.

C. IO Consolidation

Storage systems always consolidate write/read operations in the *same memory region* to reduce the overhead in hardware and driver [22], [12]. With aggregating multiple write/read operation, less IO are required. As we mentioned in Section II-B1, packet throttling makes it challenging to improve the performance of small size WR. Similar to the storage system, we consider the IO consolidation mechanism is also applicable to sequential remote memory access. Figure 7 illustrates the consolidation processing, these writes/read operations to the same aligned memory region (S bytes) will be delayed to post to RNIC until 1) there are θ requests which will modify this aligned memory, or 2) time-out. So several RDMA requests whose sizes are smaller than S can reduce from θ to only one network round trip. Different from vector IO, which requires to write/read multiple buffers in the remote side, by using IO consolidation, write/read requests should intend to access the same aligned memory region (e.g., 4KB Page).

We measure the performance of 32 bytes random write workload using IO consolidation versus native access path as shown in Figure 8 (the aligned page size is 1KB). After applying IO consolidation, the throughput is significantly higher (7.49× when θ is 16) than using native approach.

Discussion. IO consolidation is only suitable in scenarios where extremely high throughput is required. The main scenario is a skewed workload, where most requests are written to a small range of memory areas. We can provide a "hint" interface that users can indicate the frequency access data, and any modifications to these remote areas will be consolidated.

For instance, *burst buffer* is a commonly used component to alleviate the stress of massive IO requests, such as using SSD as a burst buffer tier into the storage system to absorb application I/O requests. To some extent, we can understand this consolidating one-sided RDMA verbs approach as the *remote burst buffer*.

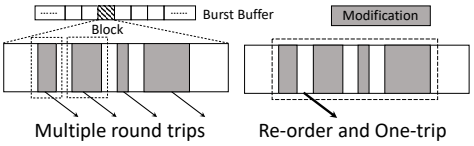


Fig. 7: IO consolidation.

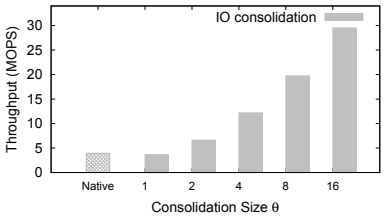


Fig. 8: The performance of IO consolidation.

D. NUMA Effects

As mentioned in Section II-B4, NUMA architecture is widely deployed in modern data centers. Under the NUMA architecture, the asymmetry between local socket memory access and remote socket memory access is significant, which is a major challenge in system design [13], [27]. We evaluate this asymmetry and show the numeric results in Table II by using Intel MLC [21]: the latency/bandwidth of remote socket memory access can be 43%/63% lower/higher than local socket memory access.

TABLE II: Throughput/latency of local inter-socket access.

Type	**Latency** (ns)	**BandWidth** (GB)
local socket	92	3.70
remote socket	162	2.27

In a data center, each machine will be equipped with at least one RNIC. Each NIC has one or two ports [36], which is known as multi-port/RNIC, and each port/RNIC is bound to one of the sockets. Previous works [43], [32] aim to benefit from multi-ports in order to gain a high throughput — the throughput is always linearly increasing with the number of ports. However, the inter-socket communication is inevitable in RDMA communication because each port/RNIC is affiliated to an indicated socket. Accordingly, such asymmetry between local/remote socket also exists in remote memory access cases [8]. Even more, this phenomenon will be amplified since the overhead is two-fold, it partly comes from RNIC to socket in the send side, partly from socket to memory. For each remote memory access, its end-to-end latency can be decomposed as: $T_{RNIC \rightarrow Socket} + T_{Socket \rightarrow Memory} + T_{Network}$. We name the socket where binds QP in use as *local socket*, and other sockets as *remote socket*. In the case that QP doesn't bind to the local socket, $T_{RNIC \rightarrow Socket}$ will increase due to the overhead of inter-socket communication. However, if RDMA-enabled memory is attached to a remote socket, the local socket will require to access memory of the remote socket in the same machine but eliminate the inter-socket access in the remote machine as well.

TABLE III: Throughput and latency of remote inter-socket access. (*alt* alternate memory means RNIC and core/memory is not resident in the same socket)

Read/Write $(\mu s$/MOPS)		own core		alt core	
		own mem	alt mem	own mem	alt mem
own core	own mem	1.78/0.93 6.17/7.06	1.78/0.94 6.13/7.06	1.92/1.06 3.11/4.76	1.97/1.17 3.13/4.76
	alt mem	1.79/0.93 6.15/7.06	1.79/0.94 6.15/7.14	1.94/1.07 3.11/4.76	1.97/1.17 3.11/4.76
alt core	own mem	2.06/1.07 6.15/7.06	2.06/1.08 6.15/7.06	2.20/1.20 3.11/3.36	2.24/1.31 3.11/3.19
	alt mem	2.16/1.17 6.15/7.06	2.16/1.19 6.15/7.06	2.30/1.32 3.17/3.30	2.34/1.43 3.13/3.19

Table III shows the diversity of latency/throughput while the CPU core or RDMA-enabled memory resides in or not in the RNIC bound socket. As we can see from this table, if the RDMA-enabled memory and CPU core are attached in the alternate socket without attaching RNIC (worst case), the latency/throughput will be nearly 55%/49% higher/lower than the case when all these three parts (RDMA-enabled memory, CPU and RNIC) are in the same socket (best case). Even more, this diversity will increase when the next generation RNIC is available since $T_{network}$ will be reduced significantly. Another observation is that the latency has no significant increase if the memory and RNIC are in the same socket. In most cases, this latency gap will be only 4∼10% if the memory and RNIC are in different sockets.

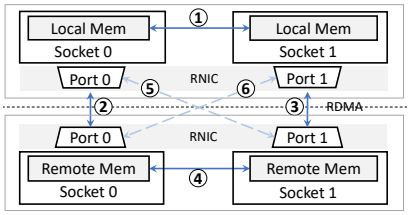


Fig. 9: Multi-port RDMA-based data access path under NUMA arch. It also fits into the multi-RNIC scenario.

Discussion. Figure 9 illustrates the simplest two-machine scenario in which each machine is equipped with two sockets/ports, and the ports are bound to distinct sockets separately. The data transmission can be performed in four paths: the local/remote inter-socket memory access (① ④) and inter-machine RDMA communication (② ③ ⑤ ⑥). As we discussed in Section II-B2, massive QPs would be the primary reason to incur performance degradation [24], [7], so the performance degradation is unneglected if each local socket establishes connections with all remote sockets. For example, all-sockets-to-all-sockets connection status requires at least $s \times s \times 2m$ QPs for each socket (assume there are m machines and each machine is equipped with s sockets). If each socket establishes the connection with a dedicated socket, the required number of QPs is only $s \times 2m$ ($\frac{1}{s}$ of all-to-all connections).

Assuming the local socket 0 is responsible for a request to access remote memory in socket 1 on the remote machine, and the original data path is ⑤ which requires massive connections (i.e., low performance) or ② and ④, which causes significant remote inter-socket communication. Our strategy to handle such remote memory access requests is using the *proxy socket*. In the case of accessing remote memory in socket 1, socket 0 will assign the request to local socket 1, i.e, proxy socket (①). After that, socket 1 processes this request as usual (②), and return the data to socket 0 after finishing the data transmission.

E. Remote Memory Atomic Operation

Atomic operations play a vital role in both remote memory and local memory. The classical application scenarios include sequencer [9], non-blocking data structure [35] and lock service [39], [33]. Here we compare the performance and scalability of atomic operations in remote memory and local memory and RPC-based implementations.

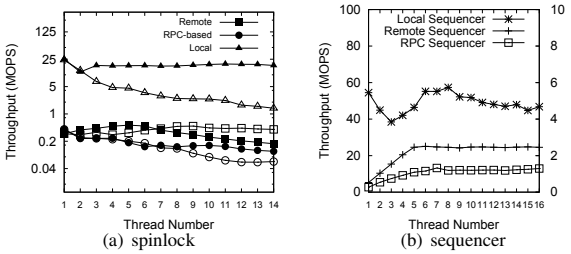


Fig. 10: Comparisons between local/remote/RPC-based atomic operations. (Note that solid points are back-off counterparts)

As an example, developers usually use an atomic operation to implement a spinlock. Figure 10(a) illustrates the difference between three approaches. We implement local/remote spinlock via the GCC builtin `__sync_compare_and_swap` and `RDMA compare_and_swap` respectively. Besides, we implement RPC-based spinlock with channel semantic verbs (i.e., send/receive).

The remote lock has a higher throughput than RPC-based lock, which is 1.54∼2.80× of the RPC-based lock. Besides, the throughput of remote lock only reduces to 14% compared to 1.2% of the local lock when the thread number increases from 1 to 14. Due to the high contention, the throughput will be convergence (0.33/0.31MOPS for lock/remote lock when the thread number is eight) as the thread number increases. To summarize, the remote spinlock based on RDMA atomic operations can achieve better scalability than the local lock and higher performance than the RPC-based lock.

We also improve remote spinlock with exponential back-off strategy [4] to eliminate high contention as shown in Figure 10(a) (solid points). The exponential back-off remote lock significantly eliminates the lock contention, which performs 2.32/3.63× higher throughput than local spinlock and RPC-based spinlock when the machine number is 14.

We also test another instance of using atomic operation, named sequencer, which will return a monotonically increasing number for each request. Figure 10(b) shows the performance difference between local sequencer (using GCC builtin `__sync_fetch_and_add`), remote sequencer (using RDMA `fetch_and_add`) and RPC-based sequencer (using channel semantic RPC). The remote sequencer exhibits a stable performance at around 2.6MOPS when the thread number is more than 5, which is 1.87∼2.25× higher than RPC-based sequencer.

Discussion. Compared with other one-sided verbs such as `RDMA Read/Write`, `RDMA Atomic` has lower throughput, only achieving 2.2∼2.5MOPS for each RNIC port. The RDMA atomic verbs are still essential since they could work in coordination with the local atomic operations [56]. Despite this, RDMA atomic operations will be more adaptable for remote memory semantics, even though [26] shows a better performance to implement remote lock or sequencer via UD RPC. Apparently, one-sided verbs which cost nearly no CPU consumption in the remote node has adaptable semantics for applications and can guarantee reliable. If RDMA atomic operations are not the performance bottleneck of applications, they are recommended as the primary choice for the sake of the benefits in reducing programming complexity as well as the lower CPU consumption.

IV. Case Study

A. Applications Classifications.

Generally, we can conclude that remote memory is leveraged to improve performance or system availability in three scenarios. **(I)** Use remote memory as a cache to reduce access latency, such as the front-end cache in distributed data structures [3] and client-side cache in key-value stores [56]. **(II)** Replace the local disk to buffer immediate data since the remote memory will be faster. Several classical applications such as the join and shuffle operation will produce massive amounts of temporary data and thus can benefit from this. **(III)** Support replicating data to remote memory [52], [42], [54]. The recovery time will be short with fast migration processing.

In this section, we discuss four application scenarios to show the capability of remote memory. These four scenarios can be seen as classical remote memory applications of Disaggregated HashTable (Section IV-B, **I**), Distributed Shuffle (Section IV-C, **II**), Distributed Join (Section IV-D, **II**) and Distributed Log (Section IV-E, **III**) separately.

B. Application 1: Disaggregated HashTable

As we mentioned earlier, data structures, especially disaggregated data structures [3], [35], [41], can gain performance improvement from leveraging remote memory, but this is still not well studied. Therefore, to show its power, we implement a disaggregated hashtable as shown in Figure 11. In disaggregated hashtables, request processing and storage are decoupled to the front-end and back-end respectively. The front-end will play a computation role to process insert/search requests and access data in the back-end via one-sided RDMA. In the

following, we present how to improve the performance of this disaggregated hashtable with step-by-step optimizations.

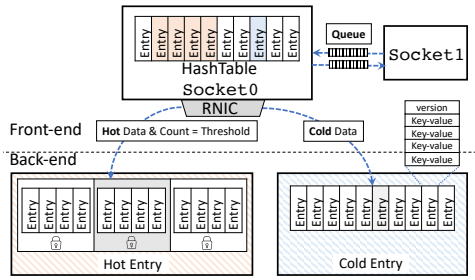


Fig. 11: Disaggregated HashTable Architecture.

NUMA-awareness. In our evaluation setting, each machine is equipped with one RNIC with two ports. As with the proposed mechanism in Section II-B4, each local socket only establishes a connection with the matched remote socket. If a local socket wants to access an unmatched remote socket, the memory access request will be assigned to another matched socket (i.e., proxy socket), which will not incur remote inter-socket communication. After the completion of data transmission, the proxy socket will return the results to the local socket. Generally, IPC (Inter-Process Communication) or message passing is available for interaction between a local socket and a proxy socket. Even so, we create two message queues in shared memory here: one for pushing requests and the other for pulling results by the local socket.

IO consolidation. According to recent surveys [10], the real-world key-value workloads have a skewed distribution. Under the skewed workloads, some entries will be accessed frequently. Thus we can divide these keys into a hot entry area and cold entry area. Front-end will buffer hot entries and write them to the hot memory area as a burst buffer. According to the value of an entry's key, we organize these hot entries as several *blocks* where each block has 2^t entries. Unlike cold entries, hot entries will write/read with block granularity. A block will be written to remote memory when there are enough modifications (θ) to i_{th} block or the lease is expired.

Atomic operation. According to the discussion in Section III-E, for cold entries, we adopt a multi-version approach with several available versions for each key-value entry to handle the concurrency. The writer first gets a version number and increments its value by using `RDMA fetch and add`, and then writes the data of key-value entry to the back-end via `RDMA Write`. In the hot entry area, there are remote spinlocks with the exponential backoff strategy for each block.

Firstly, We measure the optimizations with break-down. The skewed workloads are generated according to *Zipf distribution* with parameter 0.99. For 100% write workloads with 64-byte value-size, we warm-up the hashtable by selecting frequent access keys into the hot memory area. Figure 12 shows the impact of different optimizations. The number of front-ends is varied from 1 to 14, and each front-end can access any key-value entry in the back-end. After applying NUMA-aware

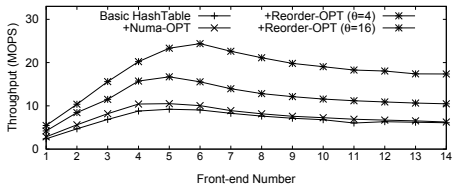
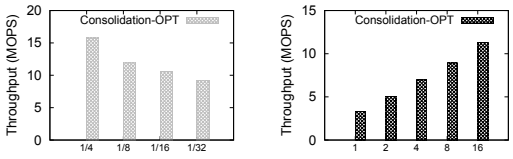


Fig. 12: The impact of disaggregated hashtable optimizations.



(a) Varying hot key proportion (b) Varying batch size

Fig. 13: The impact of re-order.

optimization, disaggregated hashtable can achieve a maximum throughput of around 10.5MOPS which is 14.1% higher than the baseline (*Basic HashTable*), mainly due to that the inter-socket communication time is saved. By avoiding the overhead of substantial sequential access and exploiting the characteristics of packet throttling, our strategy of caching hot entries and consolidating remote access provides a significant performance improvement. The peak throughput is around 24.4MOPS with six front-ends – nearly 1.85×∼2.70× higher as opposed to the basic implementation and NUMA-aware optimizations. The performance of IO consolidation in disaggregated hashtable is shown in Figure 13. In Figure 13(a), throughput decreases as the proportion of hot keys drops, but there is only around 6 MOPS drop in throughput while the proportion of hot keys changes from 1/4 to 1/32. It is witnessed in Figure 13(b) that the increasing rate of throughput is also not able to comply with the growing rate of the batch size.

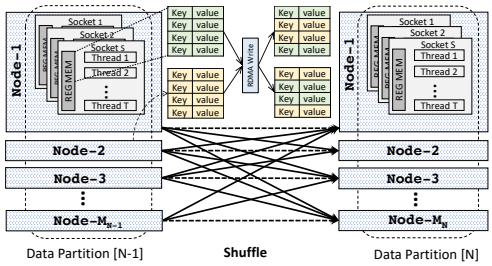


Fig. 14: Distributed Shuffle.

C. Application 2: Distributed Shuffle

Shuffle operation is well studied in parallel data systems. For instance, a simple query includes many sub-queries that are replicated across the cluster. Shuffle operation is critical component to transfer and aggregate data as well as to combine these queries. Based on RDMA, [5] describes how to manage RDMA-enabled buffers and distribute the data

efficiently, and [31] presents a pull-based shuffle operation to exploit the power of unreliable transport. Distinct from these works, we implement a push-based model [18] since in-bound `RDMA Write` has higher performance than out-bound `RDMA Read` [24], [35].

For the data stream [47] based on the shuffle algorithm, we spread the original shuffle algorithm onto a set of worker nodes, and each node has several executors to process shuffling. As shown in Figure 14, n executors will distribute data to m executors in the next round with an all-to-all (full-mesh) style. To start with, a *shuffle rule* should be decided (e.g., allocate entries to different nodes via each entry's hashing value). Each key-value entry will be sent to the corresponding node immediately after determining its destination based on the *shuffle rule*.

NUMA-awareness. Similar to the discussion in Section II-B4, we assign each executor to a dedicated socket with affinitive memory and RNIC port for mitigating unnecessary inter-socket overhead.

Batch Schedule. In consideration of the fact that entries will be distributed to different destinations, we cannot write multiple entries to a particular machine's remote memory directly. A naive approach is to buffer these same-destination entries in an RDMA-enabled buffer and write to remote memory at once when reaching a threshold (i.e., SP method). However, it will incur extra memory copying. The SGL method is appropriate for the shuffle operation, which can reduce both memory copy overhead and CPU involvement. While processing the incoming entries, the local address of these entries that have the same destination will be organized as a WR. When there are enough entries, an executor will post them to the RNIC via a single one-sided RDMA operation.

Atomic operation. According to the survey in Section III-E, we choose `RDMA fetch and add` to implement synchronization primitives between current/next stage executors, due to that the one-sided verbs cannot be perceived by the executors in the next stage.

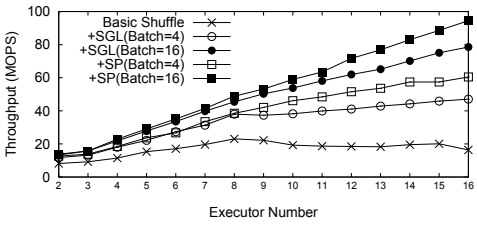


Fig. 15: The performance of shuffle.

Figure 15 presents the performance improvement of the above optimizations. As we expected, the batch approach shows its power to increase the throughput. To be specific, when the batch size is 16, and the executor number is 16, the throughput of SGL/SP is 4.8/5.8× better than basic shuffle (without optimization). Compared with SP, using SGL to batch remote access has a comparable effect. Another observation is that SGL has poor scalability with larger batch sizes. As we mentioned in Section III-A, this is because RNIC has a limited

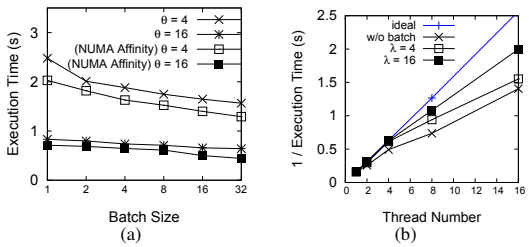


Fig. 16: The performance of join under different optimizations. (θ: executor/thread number, λ: batch size)

ability to fetch data directly. On the contrary, CPU has more power to fetch data to the buffer. Though this will bring extra overhead, the performance will be better due to the packet throttling (Section II-B1).

D. Application 3: Distributed Join

Join operation involves multiple shuffling phases, and we implement join operation using shuffle operation with the SGL method in Section IV-C. In our implementation, the join algorithm can be subdivided into two phases: the partition phase and the build-probe phase [6]. Especially, the partition phase relies on the shuffle operator with RDMA involvement. In the build-probe phase, each executor uses Intel TBB `concurrent hash map` [45] to store the inner relation of its partition, then probes with the tuples from outer relation.

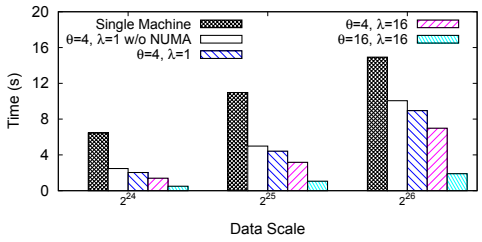


Fig. 17: The performance breakdown of join.

In Figure 16, we measure the performance of the distributed join operation using a fixed-size inner/outer relation [2], composed of 16 million tuples. Figure 16(a) illustrates the change in execution time when applying a NUMA-aware strategy and different batch sizes. With batching, when there are four executors, the execution time is reduced to up to 76% of the original standalone implementation (6.46 s) and we see up to 37% reduction compared to the non-batching implementation. On the other hand, applying NUMA-awareness optimization reduces join's execution time by 12%∼30%. With the SGL batch strategy, we observe a sub-linear reduction in execution time for the distributed join algorithm when increasing the number of executors, as shown in Figure 16(b). When the batch size is 16, there is only 22% performance degradation compared to the ideal, which means SGL has better scalability with increasing thread number. We also test the distributed join algorithm to explore the breakdown with large workloads as

[2]Note that the right figure uses $execution\ time^{-1}$ as the y axis

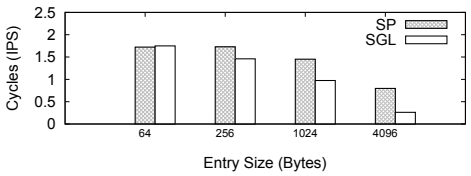


Fig. 18: The utilization of different entry size under SP/SGL. (IPS: instructions per second)

Figure 17 shows. When increasing the input sizes to 4×, we can observe that those optimizations still maintain a constant performance reduction. In total, with all optimizations, it has a 5.3×/10.3× performance improvement compared to native single-machine/distributed implementations.

We evaluate the separate impact of the entries' size and the batch's type on CPU consumption. The cost of CPU cycles for SP and SGL is measured in second. We then normalize the number of executors as seven. Besides, we use different entry sizes which vary from 64 bytes to 4096 bytes. As shown in Figure 18, SGL exhibits a lower CPU consumption compared with SP. When the entry size is 4096 bytes, the cost of using SGL is reduced by nearly 67.2%. The primary cause is that the fetching data phase has no CPU involvement.

E. Application 4: Distributed Log

The distributed transaction supports in-memory transaction processing through partitioning data into multiple transaction engines, in this architecture, each transaction engine will occupy more than one data table which can be accessed by other engines via RDMA. The distributed log is a sub-module of distributed transactions to enhance reliability, and it is an append-only, totally ordered sequence of distributed transaction records ordered by time. With the design principle of remote memory, the whole logging phase is "one-sided" which has a better performance and is transparent to different transaction engines [53]. At commit time, the writer will reserve consecutive space in the global log of the remote machine. After that, the transaction engine can write the record to the dedicated address in the global log via `RDMA Write`. To reduce the extra memory copies to the temporary log area, we can exploit the updated data in the data table and then directly write it to the remote global log as the record.

NUMA-awareness. To mitigate expensive inter-socket communication, if the data tables are in the alternative socket memory, the transaction engine first copies the records and coalesce them together. After that, it transfers them from the alternative socket memory to the buffers in the NUMA-friendly socket, and finally write (i.e., `RDMA Write`) to the global log by using SP.

Batch Schedule. According to the characteristics of vector IO, SGL can be applied to the distributed log by coalescing multiple records from the data table and buffers in the NUMA-friendly socket.

Atomic Operation. The consecutive space reserving stage exploits the remote sequencer using `RDMA fetch and add`. Each transaction engine can gain an independent offset of the global log and guarantee that no conflict happens between different transaction engines.

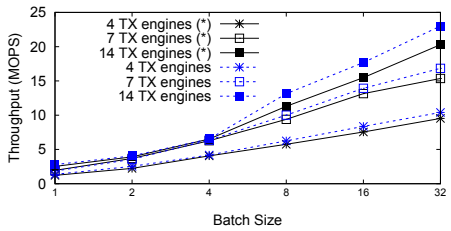


Fig. 19: The performance of distributed log under different optimizations (*: w/o NUMA awareness).

As expected, after applying the NUMA-aware design, Figure 19 shows distributed log achieves 17.7MOPS while the original one is 15.5MOPS (14% throughput improvement) with 14 transaction engines. Also, distributed log enables increasing throughput gains from the batch. In the 7 transaction engines scenario, with increasing batch size, the case whose batch size is 32 can perform 9.1× throughput improvement than the one without batching.

V. Related Works

This paper is an in-depth analysis of the RDMA-based remote memory access semantics, and some ideas are inspired by recent works. However, both remote memory and how to optimize RDMA communication are hot topics with thorough studying. Here we introduce several most related works from three aspects.

RDMA-Based Optimization. Herd [23] and Fasst [25] are key-value stores based on the server reply paradigm under UD mode. They inspire some commonly used RDMA network optimization methods like *inline* and *selective signal*. RDMA bench [26] introduces several optimization approaches including Doorbell, which appears in Section III-A. Frey et al [17] show in their analysis that the hidden overhead of RDMA is memory registration, and hence they design a memory registration strategy. Apart from system-level approaches, some works improve the RDMA protocol stack from flow control [20], QoS [29] and congestion control [51] separately. We give optimization guidelines from the view of memory semantics.

Remote Memory Trending. As mentioned in Section II, there are many recent and ongoing efforts to explore the usage of remote memory. Lim et al. [30] introduce a new trend of separating computation and memory with network interconnect and define the concept of disaggregated memory blade (i.e., remote memory). FaRM [15] is a distributed transaction system that exploits remote memory to achieve both low latency and high throughput. Based on remote memory, there are several works [16], [48], [49] that focus on optimizing shared memory systems, transaction systems and databases. These can be further subdivided into concurrent control [55], data-intensive operator [28] and consensus protocol [54]. Aguilera et al. [2] survey the research hotspots about remote memory and give various suggestions about applying it. Remote Region [1] rethinks the programming interface of remote memory and

proposes file system level interfaces as the most efficient and flexible approach. We systematically optimize four applications which can benefit from using remote memory.

NUMA architecture and RDMA. There are several works discussing how to make RDMA-enabled clusters aware of the NUMA architecture. Ren et al. [46] design a two-sided RDMA-based FTP protocol which assigns each NUMA socket to an indicated thread to handle local file IO request and hence reduces total access latency. On the hardware side, soNUMA [40] implements remote memory controller (RMC) on top of a NUMA memory fabric via a stateless messaging protocol. Recently, Mellanox/NVIDIA released a new product called Socket Direct Adapter [38]. It can be offered as two PCIe cards which install in two different sockets, so as to eliminate the extra network traffic between the two sockets. We analyze remote inter-socket access from the perspective of eliminating the inter-socket communication.

VI. Conclusion

In this paper, we have empirically analyzed memory semantics of RDMA. Specifically, we characterize the remote memory access properties from five aspects: vector IO, random/sequential access, IO consolidation, NUMA-aware access, and atomic operations. We also make detailed evaluations of how these factors influence performance. This work is essential for several reasons. First, it provides comprehensive performance results, showing features of one-sided RDMA operations under different settings. As such, our work will help developers utilize RDMA-enabled remote memory more efficiently. Additionally, this study contains guidance for performance optimizations in various types of modern data center applications. The results shown in the case studies indicate that remote memory access with RDMA is a powerful way of improving the throughput of four applications discussed.

Acknowledgment

We would like to thank *IEEE Cluster* anonymous reviewers for their helpful feedbacks. This Work is supported by National Key Research & Development Program of China (2020YFC1522702), Natural Science Foundation of China (61877035).

References

[1] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, et al. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 775–787, 2018.

[2] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote memory in the age of fast networks. In *ACM SoCC*, pages 121–127. ACM, 2017.

[3] M. K. Aguilera, K. Keeton, S. Novakovic, and S. Singhal. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 120–126. ACM, 2019.

[4] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel & Distributed Systems*, 0(1):6–16, 1990.

[5] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using rdma. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1463–1475. ACM, 2015.

[6] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. Distributed join algorithms on thousands of cores. *Proceedings of the VLDB Endowment*, 10(5):517–528, 2017.

[7] Y. Chen, Y. Lu, and J. Shu. Scalable rdma rpc on reliable connection with efficient resource sharing. In *14th EuroSys*, page 19. ACM, 2019.

[8] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1077–1091, 2020.

[9] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using rdma and htm. In *11th Eurosys*. ACM, 2016.

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

[11] J. Corbet. Scatterlist chaining. https://lwn.net/Articles/234617/, May 2007.

[12] J. Corbet. The end of block barriers. https://lwn.net/Articles/400541/, 2010.

[13] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: a holistic approach to memory placement on numa systems. *ACM SIGPLAN Notices*, 48(4):381–394, 2013.

[14] K. Dong, L. Huang, and Y. Zhu. Exploiting rdma for distributed low-latency key/value store on non-volatile main memory. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 225–231. IEEE, 2017.

[15] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: Fast remote memory. In *11th USENIX NSDI*, pages 401–414, 2014.

[16] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th symposium on operating systems principles*, pages 54–70, 2015.

[17] P. W. Frey and G. Alonso. Minimizing the hidden cost of rdma. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560. IEEE, 2009.

[18] R. C. Gonçalves, J. Pereira, and R. Jiménez-Peris. An rdma middleware for asynchronous multi-stage shuffling in analytical processing. In *Distributed Applications and Interoperable Systems*, pages 61–74. Springer, 2016.

[19] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX NSDI*, pages 649–667, 2017.

[20] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215. ACM, 2016.

[21] Intel. Intel® memory latency checker v3.7. https://software.intel.com/en-us/articles/intelr-memory-latency-checker, 2019.

[22] Y. Joo, S. Park, and H. Bahn. Exploiting i/o reordering and i/o interleaving to improve application launch performance. *ACM Transactions on Storage (TOS)*, 13(1):8, 2017.

[23] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *ACM Conference on SIGCOMM*, pages 295–306. ACM, 2014.

[24] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. *ACM SIGCOMM Computer Communication Review*, 44(4):295–306, 2015.

[25] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *12th USENIX Symposium on OSDI 16*, pages 185–201, 2016.

[26] A. K. M. Kaminsky and D. G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of USENIX ATC'16 2016 USENIX Annual Technical Conference*, page 437, 2016.

[27] B. Lepers, V. Quéma, and A. Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 277–289, 2015.

[28] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and rdma. In *Proceedings of the*

2016 International Conference on Management of Data, pages 355–370. ACM, 2016.

[29] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, et al. Hpcc: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 44–58. ACM, 2019.

[30] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *International Symposium on Computer Architecture (ISCA)*, volume 37, pages 267–278. ACM, 2009.

[31] F. Liu, L. Yin, and S. Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. In *20th Eurosys*, pages 48–63. ACM, 2017.

[32] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, and T. Moscibroda. Multi-path transport for rdma in datacenters. In *15th USENIX NSDI*, pages 357–371, 2018.

[33] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: an rdma-enabled distributed persistent memory file system. In *USENIX ATC*, pages 773–785, 2017.

[34] T. Ma, T. Ma, Z. Song, J. Li, H. Chang, K. Chen, H. Jiang, and Y. Wu. X-rdma: Effective rdma middleware in large-scale production environments. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.

[35] T. Ma, M. Zhang, K. Chen, Z. Song, Y. Wu, and X. Qian. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric nvm architecture. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 757–773, 2020.

[36] Mellanox. Connectx-4 en adapter card single/dual-port 100 gigabit ethernet adapter. http://www.mellanox.com/page/products_dyn?product_family=204&mtag=connectx_4_en_card, 2017.

[37] Mellanox. Extended memory windows. https://www.openfabrics.org/images/eventpresos/2017presentations/112_Extended_Memory_Windows-_TOved.pdf, 2017.

[38] Mellanox. Maximizing server performance with mellanox socket direct® adapter. http://www.mellanox.com/related-docs/whitepapers/WP_Mellanox_Socket_Direct.pdf, 2019.

[39] S. Narravula, A. Marnidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda. High performance distributed lock management services using network-based remote atomic operations. In *7th CCGrid*. IEEE, 2007.

[40] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out numa. *ACM SIGPLAN Notices*, 49(4):3–18, 2014.

[41] S. Novakovic, Y. Shan, A. Kolli, M. Cui, Y. Zhang, H. Eran, B. Pismenny, L. Liss, M. Wei, and D. Tsafrir. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 97–108. ACM, 2019.

[42] M. Poke and T. Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118, 2015.

[43] Y. Qian and A. Afsahi. High performance rdma-based multi-port all-gather on multi-rail qsnet ii. In *21st HPCS*, pages 3–3. IEEE, 2007.

[44] R. Recio. Rdma enabled nic (rnic) verbs overview. *dated Apr*, 29:1–28, 2003.

[45] J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[46] Y. Ren, T. Li, D. Yu, S. Jin, and T. Robertazzi. Design and performance evaluation of numa-aware rdma-based end-to-end data transfer systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 48. ACM, 2013.

[47] N. Rivetti, E. Anceaume, Y. Busnel, L. Querzoni, and B. Sericola. Online scheduling for shuffle grouping in distributed stream processing systems. In *17th Middleware Conference*, page 11. ACM, 2016.

[48] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. Dragojević, D. Narayanan, and M. Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data*, pages 433–448, 2019.

[49] Y. Shan, S.-Y. Tsai, and Y. Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337, 2017.

[50] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on OSDI*, pages 317–332. USENIX Association, 2016.

[51] A. Singhvi, A. Akella, D. Gibson, T. F. Wenisch, M. Wong-Chan, S. Clark, M. M. Martin, M. McLaren, P. Chandra, R. Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 708–721, 2020.

[52] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes. Tailwind: fast and atomic rdma-based replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 851–863, 2018.

[53] S.-Y. Tsai and Y. Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th SOSP*, pages 306–324. ACM, 2017.

[54] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. Apus: Fast and scalable paxos on rdma. In *ACM SoCC*, pages 94–107. ACM, 2017.

[55] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *13th USENIX OSDI*, 2018.

[56] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *25th SOSP*, pages 87–104. ACM, 2015.

[57] K. Zhang, R. Chen, and H. Chen. Numa-aware graph-structured analytics. *ACM SIGPLAN Notices*, 50(8):183–193, 2015.

[58] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.