Ghostwriter: a Distributed Message Broker on RDMA and NVM

Hendrik Makait
Hasso Plattner Institute
University of Potsdam
hendrik.makait@guest.hpi.de

Bonaventura Del Monte
Observe Inc.
ventura@observeinc.com

Tilmann Rabl
Hasso Plattner Institute
University of Potsdam
tilmann.rabl@hpi.de

ABSTRACT

Modern stream processing setups heavily rely on message brokers such as Apache Kafka or Apache Pulsar. These systems act as buffers and re-readable sources for downstream systems or applications. They are typically deployed on separate servers, requiring extra resources, and achieve persistence through disk-based storage, limiting achievable throughput. In this paper, we present Ghostwriter, a message broker that utilizes remote direct memory access (RDMA) and non-volatile memory (NVM) for highly efficient message transfer and storage. Utilizing the hardware characteristics of RDMA and NVM, we achieve data throughput that is only limited by the underlying hardware, while reducing computation and disaggregating storage and data transfer coordination. Ghostwriter achieves performance improvements of up to an order of magnitude in throughput and latency over state-of-the-art solutions.

VLDB Workshop Reference Format:
Hendrik Makait, Bonaventura Del Monte, and Tilmann Rabl. Ghostwriter: a Distributed Message Broker on RDMA and NVM. VLDB 2024 Workshop: Fifteenth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2024).

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/hpides/Ghostwriter/.

1 INTRODUCTION

Over the past years, the advancements in networking and storage hardware technology enable novel designs for distributed data systems. Researchers have shown that distributed data systems such as databases or stream processing systems (SPSs) benefit significantly from fast networks, such as Infiniband (IB) networks, but they require architectural changes to achieve this goal [4, 6, 22, 28]. Remote Direct Memory Access (RDMA) enables low-latency data access [4, 10], making data locality less relevant from the network side. This enables disaggregated architectures that decouple computation and storage while retaining high performance [4, 7]. Furthermore, Non-Volatile-Memory (NVM) has become publicly available and provides a fast and byte-addressable alternative to secondary storage. Several approaches exist that introduce NVM as part of the storage hierarchy for databases or key-value stores [2, 8, 23, 24, 27]. Overall, efficiently using NVM and RDMA requires careful design that reflects their beneficial access patterns, which differ significantly from common Ethernet networks and main-memory.

In this paper, we fully utilize RDMA and NVM in distributed message brokers (DMBs). DMBs, such as Apache Kafka, Apache Pulsar, and RedPanda, are used in big data tool-chains as ingestion and storage layer for data-streams. Data producing applications, e.g., application loggers and Internet-of-Things sensors, send event records to a DMB, which stores them and enables other application to consume and process them at a later stage. Among possible data consuming applications, SPSs consume data-streams stored in DMB to perform further data-analytics tasks in real-time [21].

To enable SPS to provide accurate and timely results, DMB must fulfill several requirements: First, they need to provide high throughput and low end-to-end latency to avoid a data-transfer bottleneck. Second, DMBs must store all messages for a time period and allow consumers to re-read them if necessary. Re-reading event records is important for SPSs as they must cope with failures and provide exactly-once semantics. Third, no data record must get lost to enable accurate results on the data path among the data-producing application, the DMB, and the SPS.

To meet these requirements, DMBs have been designed with TPC/IP networks and HDD/SSD storage in mind. DMBs cache data in local storage and coordinate the access of producing and consuming applications to the storage, to avoid network traffic whenever possible. Furthermore, DMBs persist data on secondary storage, which induces significant overhead for small messages, as they result in small disk writes. Finally, messages are commonly stored in an append-only fashion as they arrive at the broker. This makes parallel data transfer more complex, if a given message ordering must be kept. These design choices allow current DMBs to be efficient when scaling-out to handle large volumes of data [12]. However, their single-node efficiency is limited, as they are not designed for modern hardware. In this paper, we re-think DMB design to fully exploit RDMA and NVM for stream processing workloads. To this end, we re-design the access pattern to network and storage via RDMA and NVM, as current state-of-the-art systems do not benefit out-of-the-box from these modern technologies. As a result, we propose efficient RDMA-based communication patterns to write to disaggregated NVM-based storage.

We combine the above techniques in our prototype called *Ghostwriter*: a DMB that leverages modern hardware for fast and CPU-efficient data transfer while guaranteeing message delivery and persistence. Ghostwriter combines high bandwidth and low latency of RDMA data transfer with the byte-addressability and off-the-shelf persistence of NVM. Compared to current state-of-the-art DMB, Ghostwriter improves throughput and latency by one order of magnitude.

In summary, we contribute the following: First, we propose Ghostwriter, a DMB that leverages RDMA and NVM to provide

Proceedings of the VLDB Endowment. ISSN 2150-8097.

fast and efficient data transfer with guaranteed message delivery and intra-partition message ordering for parallel message transfers. Second, we demonstrate how optimizations that exploit the sequential access patterns of DMB workloads and the byte-address-ability of RDMA can further reduce the coordination overhead. Finally, we evaluate Ghostwriter, showing that it significantly outperforms existing DMBs and can scale its performance on a single partition close to the bandwidth limit of the underlying hardware.

2 BACKGROUND

In this section, we introduce persistent memory (NVM) and remote direct memory access (RDMA). Furthermore, we discuss the combination of the two solutions.

2.1 Persistent Memory

NVM devices bridge the gap between Dynamic Random Access Memory (DRAM) and fast secondary storage, such as Solid Storage Disk. They combine the persistence of secondary storage with the byte-addressable data access of DRAM and offer near-DRAM performance. Compared to DRAM, Optane DIMMs (Dual In-line Memory Modules) offer higher density as well as larger capacity of up to 512 GB, and a reduced cost/GB ratio by an order of magnitude.

NVM supports two operating modes: *Memory Mode* and *App Direct Mode*. In Memory Mode, NVM increases the volatile memory capacity at the expense of persistence. To hide NVM's access latency, it is combined with DRAM as a hardware-controlled, transparent direct-mapped cache. In App Direct Mode, applications and the OS explicitly choose between DRAM and NVM as separate memory devices. NVM can be accessed as a raw character device (*devdax*), or a block device (*fsdax*) that supports NVM-enabled file systems. Both devices are mapped into the virtual address space of the application process to allow byte-addressable access using `mmap`. Furthermore, both modes enable interleaved or non-interleaved storage layouts. Non-interleaved NVM may lead to poor utilization of DIMMs, if data accesses are not explicitly spread [27]. Interleaved NVM enable large sequential data accesses to occur on multiple NVM DIMMs. They can be parallelized, which improves the hardware utilization and achieves a higher combined throughput.

In Ghostwriter, we use NVM as a *devdax* character device in App Direct Mode for direct control over persistence. Furthermore, we opt for interleaved NVM for improved parallelism of sequential data accesses.

2.2 Remote Direct Memory Access

Modern high-performance network technologies, such as InfiniBand, provide RDMA, which enables applications to directly access memory on a remote machine without the involvement of the operating system on either machine.

The RDMA verbs API supports two different modes of data transfer: *channel* and *memory* semantics. Channel semantics or *two-sided* verbs enable traditional message-passing using `SEND` and `RECV` verbs that function similar to socket-based communication. While channel semantics avoid the overhead of TCP/IP-based communication, they involve the remote CPU. Memory semantics or *one-sided* verbs, e.g., `READ` and `WRITE`, allow an application to access remote memory without involving the remote CPU. To expose memory for remote access, the application must first register the memory locations as a memory region (MR).

2.3 Combining NVM and RDMA

By combining RDMA with NVM, applications benefit from the increased capacity of NVM as well as its persistence. To persist remotely written data, it must be flushed from the RNIC and PCIe caches into the DRAM domain. Due to the lack of a dedicated persistence mechanism for RDMA, applications have two different options depending on whether the remote machine offers direct cache access (DCA). If DCA is enabled, the RNIC bypasses memory and writes data directly to the CPU cache. Therefore, the remote host must explicitly flush the data from the caches to NVM. Since this requires active involvement of the remote CPU, this diminishes the benefits of using one-sided RDMA verbs for data transfer. If DCA is not available or disabled, the RNIC writes directly to NVM. Thus, the application can force persistence by performing an RDMA READ after one or several WRITES. The subsequent READ flushes all prior WRITES from the remote host's NIC and PCIe caches onto NVM. While this avoids involvement of the remote CPU, previous research has shown that the added latency reduces the performance benefit of one-sided verbs compared to DRAM.

3 GHOSTWRITER

In this section, we give an overview of the system design and architecture of Ghostwriter.

3.1 System Design and Architecture Overview

Ghostwriter is a DMB designed for NVM persistent storage and RDMA acceleration. We follow the following design principles to build Ghostwriter.

1) Decoupled architecture. Ghostwriter decouples the broker node from storage nodes. As a result, Ghostwriter performs CPU-intensive tasks on the broker or the producers/consumers clients to minimize the CPU overhead for storage nodes.

2) Hardware Acceleration. Ghostwriter focuses on efficient hardware utilization through RDMA-based data transfer and low-overhead persistence enabled by NVM.

3) Delivery guarantees. Ghostwriter provides intra-partition message ordering and persistence guarantees. It also allows the user to trade them for increased performance.

Figure 1 gives an overview of its components. In general, our architecture consists of four independent components:

(1) *Storage nodes* provide one large region of NVM, that is split into individual segments. These segments are used to store the data and can be accessed remotely.

(2) *Broker nodes* focus on allocating segments from the storage nodes and coordinating access to them.

(3) *Producers* publish data by sending metadata to the broker, which provides them with a remote location in which to store the data. The data can then be directly persisted in NVM using one-sided RDMA verbs.

(4) *Consumers* consume data written to storage by requesting its remote location from the broker and using a one-sided RDMA read to efficiently retrieve it.

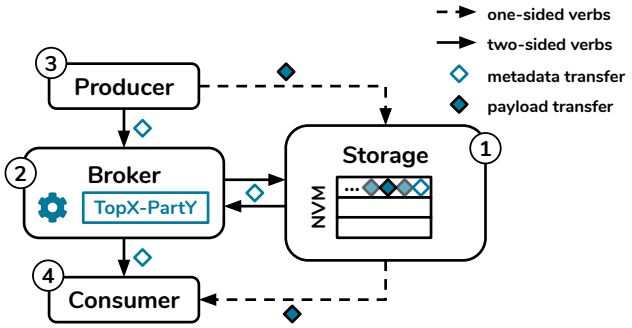


Figure 1: Ghostwriter's Architecture.

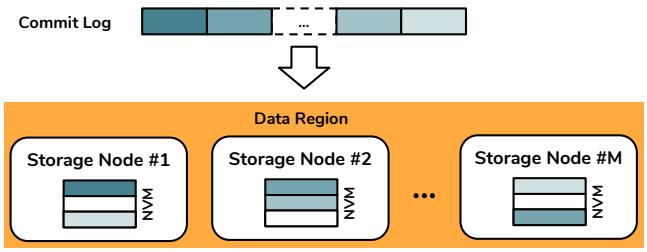


Figure 2: The commit log generated by each partition is split into segments that are stored across the data region.

Our architecture separates the responsibility for storage and computation by introducing dedicated storage nodes and leaving the broker node to handle the access and coordination for partitions.

4 COMPONENTS OF GHOSTWRITER

In this section, we present Ghostwriter's components that allow for data access: its storage nodes, brokers, producers, and consumers.

4.1 Storage Nodes

Ghostwriter stores each data partition as append-only log of messages, to enable sequential access and persist the messages.

To exploit the capabilities of modern IB networks and NVM as well as to leverage the workload characteristics of DMBs, Ghostwriter decouples computation from storage by using a segment-based design. Through segment-based storage, storage nodes provide one large data region in which fixed-size chunks are allocated by a broker to store produced messages. We illustrate this approach in Figure 2.

Ghostwriter distributes and replicates segments across storage nodes, using high-bandwidth and low-latency RDMA-capable networks. Furthermore, Ghostwriter uses NVM as persistent data storage. To scale the system, users may add additional storage nodes, which increases the number of free physical storage segments available for allocation. Storing data in distributed segments mitigates the problems created by imbalanced partitions.

To make the log independent from the actual storage locations of the individual segments, we differentiate between *logical* segments and their *physical* segments. A logical segment represents a virtual section of the log, in which messages are stored and can be addressed using their logical offset. Each logical segment contains one or several physical segments, which correspond to physical segments allocated on the storage nodes.

Storage nodes directly expose the persistent storage for remote access as a memory region. As previous research has pointed out, dynamically adjusting the size of the exposed memory region is an expensive operation [4, 26]. Therefore, storage nodes do not allocate additional segments on-demand, but they rely on a pool of pre-allocated segments. Ghostwriter's nodes perform remote, durable writes to this persistent storage using RDMA.

Performing data transfer using one-sided RDMA verbs directly to the exposed NVM does not engage the CPUs of storage nodes. Thus, our storage nodes perform negligible work, resulting in the separation of computation and storage. The primary tasks of the storage nodes are to track the usage of their memory segments, allocate new segments for brokers, and perform garbage collection.

As our storage nodes do not perform computation-intensive work, we can co-locate them with other compute-heavy applications, such as the task executors of an SPE. By co-locating storage segments with the task executors that consume the stored data, we avoid additional network traffic for transferring the data to the worker. Overall, our approach benefits from the increased throughput and reduced latency of local memory access.

4.2 Broker

A broker in Ghostwriter handles all partition access requests issued by producers and consumers. To this end, a broker allocates new storage segments from storage nodes to which producers publish their data, keeps track of existing segments as well as messages, and coordinates requests by producers and consumers.

4.2.1 Stateless Design. Brokers in Ghostwriter are stateless as Ghostwriter's storage segments are designed to include all metadata related to the state of the messages and segments, but the broker is responsible for updating this state and keeping it consistent. Brokers only need to coordinate the publish and consume requests by producers and consumers. As a result, partition metadata handling can be performed by any broker. Furthermore, recovering a broker after failure or rebalancing the assigned partitions between brokers requires to retrieve state from the segment metadata.

4.2.2 Metadata Storage. Even though the broker is stateless and all data is persisted on the storage nodes, the broker caches the metadata of the individual segments belonging to its assigned partitions. Maintaining this information serves two purposes: First, the broker needs to track its segments to be able to provide producers and consumers with the storage locations they request. Second, by caching all metadata on the segments locally, the broker avoids unnecessary remote data lookups and the associated overhead and latency when handling incoming requests [29].

To track the location of individual messages in the distributed storage segments, the broker maintains an index for each partition. Figure 3 illustrates the index structure, which is inspired by Apache Kafka. This index structure consists of a list of metadata structures for the individual logical segments belonging to the partition. The list is sorted by the *start offset* of each segment, reconstructing the order in which the segments were written. New segments may be

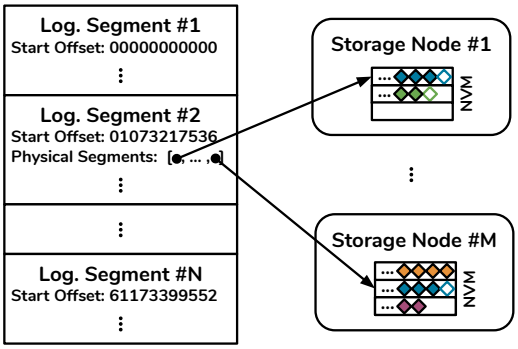


Figure 3: The segment index allows quick lookup of logical segments from logical offsets and maps them to their physical locations.

added to the list by appending them. Therefore, to find the logical segment in which a given logical offset is stored, the broker can use binary search to identify the segment with the largest start offset less than or equal to the given logical offset. The metadata stored for each logical segment consists of the data contained in the segment header as well as the metadata of all physical segments that are used to store the segment's data. The metadata stored for the physical segments in turn contain the storage node on which the physical segment is stored as well as its offset in the memory region of the storage node.

4.3 Producer and Consumer

In this section, we provide a description of the producer and consumer in Ghostwriter. The producer is the component used by a data-generating service to publish its messages to the DMB and make them accessible for consumers. In Ghostwriter, a message contains flat data structures, i.e., no pointer is stored. The consumer component enables data consuming applications such as SPEs to consume messages from Ghostwriter.

To keep the broker stateless, Ghostwriter uses pull-based consumers as inspired by Apache Kafka [13]. Compared to a traditional publish/subscribe model, pull-based consumers remove the overhead and state management for tracking consumers from the broker. This allows the developer to dynamically add and remove consumers without the involvement of the broker. As a consequence, the consumer needs to keep the state itself to ensure that it reads all messages in a partition. As consuming application reads partitions sequentially, the consumer tracks its current offset in the partition to ensure that it consumes all messages. It then uses this offset to retrieve the next message. By using purely NVM-based storage, data are not moved and thus locations do not change.

Ghostwriter provides two different protocols for the producer, each consisting of a staging, a storing, and a commit stage. In the staging stage, the producers acquire storing locations for messages. In the storing stage, messages are written to the location, and in the commit stage, the metadata is updated to indicate successful transmission. The *exclusive protocol* assumes exclusive access to the storage location and thus does not have to synchronize with other producers. An optimized version caches staging information and thus can skip the first stage on subsequent messages. The *concurrent protocol* maintains offsets between different producers and synchronizes access using atomic *compare and swap* operations. As an optimization, similar to the exclusive protocol, multiple messages can be written to the same offset as long as space permits.

5 EVALUATION

In this section, we evaluate Ghostwriter's performance through a series of microbenchmarks. In our experiments, we use the following hardware and software configurations:

Hardware Configuration. We run our experiments on a cluster with two types of nodes: *PMEM* nodes are equipped with two 18-core (36 threads), 2.60 Ghz Intel Xeon Gold 6240L CPUs, 192 GiB of DRAM, and 3072 GiB of PMEM via twelve 256 GiB Intel Optane Persistent Memory DIMMs. *Compute nodes* are equipped with two 64-core (128 threads), 2.25 Ghz AMD EPYC 7742 CPUs, and 512 GiB RAM. All nodes are connected via a Mellanox ConnectX-6 NIC running in HDR100 mode (100 Gbit/s) on *PMEM* nodes and in HDR (200 Gbit/s) mode on *compute* nodes.

Software Configuration. In our evaluation, we use Ghostwriter and Apache Kafka 2.5.0 as Systems under Test (SUTs). We implement our prototype of Ghostwriter in C++ compiled with GCC 9.4 on Ubuntu 20.04 and uses UCX for networking. The code is open-source and available on Github[1].

5.1 System Configuration

To evaluate the efficiency of the system designs, each experiment runs a single producer and a single consumer on an individual Compute node each. For Ghostwriter, we run a single storage node on a PMEM node and a single broker on a Compute node. For Apache Kafka, we run a single broker node on a PMEM node storing its data on a PMEM *fsdax* device for persistent storage. We configure Apache Kafka to use IPoIB for networking. We use a single topic and a single partition. To evaluate the impact of different message sizes, we disable the batching of multiple records through the SUT.

5.2 Microbenchmarks

In these experiments, we compare the performance of Apache Kafka and Ghostwriter through various micro-benchmarks. To this end, we first evaluate the throughput and latency that can be achieved using our hardware to establish an upper bound. Next, we evaluate throughput and latency of producers and consumers of Ghostwriter and Apache Kafka. Unless stated otherwise, each experiment measures the performance of the SUT on a total of 80 GB of data. To establish connections before the experiment and warm up caches as well as the JVM, we precede the measurement phase by a warmup phase, in which we transfer an additional 10% (or 8 GB) of data. We split the total amount of data into fixed-size messages, where the size may vary between runs depending on the experiment.

5.2.1 Baseline RDMA performance. In the first experiment, we discuss the data transfer performance of UCX - the underlying framework we use to build Ghostwriter - to establish a limit for the performance Ghostwriter can achieve. We use the `ucx_perftest` to measure the throughput, message rate and latency of performing

[1] https://github.com/hpides/ghostwriter

several communication routines offered by the UCP API, namely *PUT* (*RDMA WRITE*), *GET* (*RDMA READ*) as well as *STREAM*-based SEND/RECV. The experiments are performed using a single thread on the sender and receiver which synchronously executed the routines.To collect the measurements, we execute each routine 1M times with a preceding warmup of 10K executions for messages with sizes from 2^1 B to 2^{23} B. We show the results in Figure 4.

In Figure 4a and Figure 4b, we study the throughput and message rate of the different routines. Moreover Figure 4c shows the latency of the routines. All transfer strategies show similar shapes and scale almost linearly in throughput until they start to saturate the underlying bandwidth limit. The latency increases slowly before reaching the size of the maximum transport unit of IB at 8 KiB as it does not fully utilize a single network packet and therefore experiences large overhead. With larger message sizes the latency increases linearly for *PUT* and *GET*. For message sizes of at most 32 KiB, we reach latencies below 10 μs for all operations, as well as latencies below 100 μs for messages of at most 1 MiB for *PUT* and *GET*. Contrary to our expectations, the performance of *STREAM* degrades for messages of at least 1 MiB, showing a decrease in throughput and a super-linear increase in latency. Since we only use two-sided verbs for small metadata transfers, we do not investigate this issue further. Moreover, with 2 μs, the latency of *GET* is double the latency of the other routines (1 μs) for small message sizes. This is explained by the fact that READ operations perform a full network round trip, first sending the request to the remote node then receiving the data into the local buffer, whereas the other operations only perform a half round trip, moving data from the local to the remote node. As the message size grows, the time to transfer the data dominates the round trip time.

In summary, UCX can saturate the bandwidth limit of the underlying IB network with synchronous communication using a single thread for sufficiently large messages. These findings are in line with previous research [16, 19].

5.2.2 Baseline NVM performance. We refer to the work of Benson et al. [3] for the NVM performance, as their experiments are conducted on the same hardware as we ran out microbenchmarks. In particular, our testbed resembles their Apache-128 server, which achieves up to 13 GB/s throughput for persistent writes and up to 40 GB/s throughput for persistent reads [3].

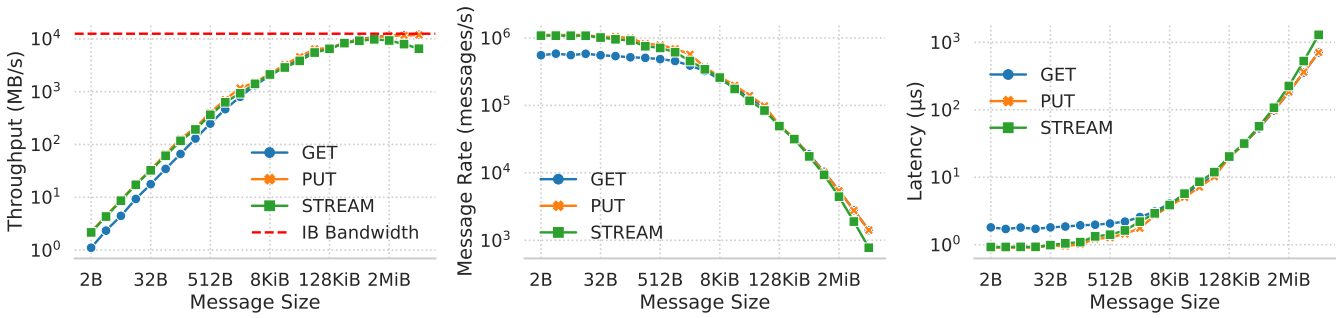
5.3 System Evaluation

In this section, we investigate the performance Ghostwriter can achieve. We execute the experiment for Apache Kafka and four protocol variants of Ghostwriter: the naïve implementation of the exclusive protocol (*Excl.*), its optimized version that caches segment metadata to avoid staging (*Excl. Opt.*) as well as the naïve implementation of the concurrent protocol (*Conc.*) and its optimized version staging ten messages at once (*Conc. Opt.*). To provide a upper limit for the performance, we include the results for the one-sided RDMA operations obtained in the previous Section. (*PUT/GET*). In the remainder of this Section, we first discuss the results for Ghostwriter and Kafka producers followed by the results for their consumers.

5.3.1 Producer Performance. In Figure 5a and 5b, we see that Ghostwriter outperforms Apache Kafka by least one order of magnitude depending on the transferred message size. While Kafka is not able to utilize the bandwidth offered by the IB network, Ghostwriter achieves a maximum throughput of around 10 GB/s with both variants of the exclusive protocol. The exclusive protocol scales with the increasing message sizes until 1 MiB, where it reaches almost 10 GB/s in throughput. At that point, the measured throughput becomes unstable as shown by the increasing standard deviation (illustrated as the shaded area) and the average drops. When comparing the naïve and the optimized implementation of the exclusive protocol, we see that the optimized version has a higher throughput for all message sizes up to 1 MiB. This is explained by the reduced control flow of the optimized variant. As we can see in Figure 5b, this effect diminishes with increasing message sizes, as the ratio between time spent on control and data flow shrinks. Comparing the performance against the upper limit provided by UCP's PUT operation, we see that even the optimized version of the exclusive protocol incurs significant overhead regardless of message sizes and is only able to reach 90% of the throughput enabled by UCP for large messages around 1 MiB. This is explained by the synchronous implementation of our protocol, in which the individual steps are not interleaved or parallelized and thus no data transfer is performed during the stage and commit steps. This effect is particularly large for small message sizes, where the overhead represents a larger percentage of the overall time spent publishing a message. When focusing on the concurrent protocol, we see that its variants approach a limit of 7.2 GB/s, which is 40% below the available bandwidth limit.

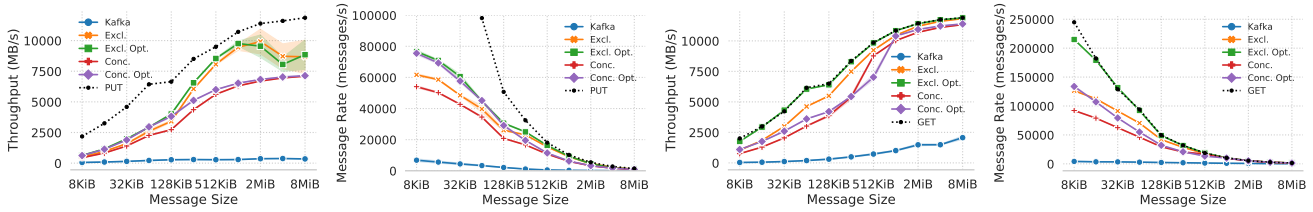
In summary, the maximum available throughput of Ghostwriter outperforms Apache Kafka by at least an order of magnitude. However, neither the exclusive nor the concurrent protocol show expected behavior for large message sizes with the former becoming unstable or the latter approaching an implicit lower limit.

5.3.2 Consumer Performance. Analogous to the maximum available throughput achieved by a single producer for the different variations of the publishing protocol, we analyze the throughput achieved by a single consumer for the consuming protocol variants. Figures 5c and 5d shows the experiment results for the different message sizes. Overall, we see both Ghostwriter as well as Apache Kafka scale with increasing message sizes. Apache Kafka is not able to utilize the provided bandwidth, yet it reaches a throughput of 2 GB/s, which is an order of magnitude higher than its maximum publishing throughput. For Ghostwriter, all protocol versions scale with the increasing message sizes, reaching a throughput of more than 11 GB/s. Comparing the exclusive and the concurrent protocol, the optimized concurrent protocol reaches the same performance for small message sizes as the naïve implementation of the exclusive protocol. This can be partially explained with the reduced control flow of the optimized concurrent protocol being offset by the increased data flow. Yet, we would expect both of these effects to diminish over time, resulting in all variants approaching the limit of the underlying network's bandwidth. While the naïve and the optimized version of both protocols converge to the same limit, the exclusive protocol reaches a higher limit than the concurrent one. This is partially explained by the slightly increased amount of data retrieved by the concurrent protocol, but may also be affected by other hardware-specific access characteristics. Given the small



(a) Throughput (MB/s) **(b) Message Rate (messages/s)** **(c) Latency (μs)**

Figure 4: Performance of UCX communication routines.



(a) Throughput **(b) Message Rate** **(c) Throughput** **(d) Message Rate**

Figure 5: Maximum producing and consuming throughput of Ghostwriter and Apache Kafka.

difference, we leave a further investigation to future work. Finally, we see that the optimized version of the exclusive protocol achieves the same performance as UCX's *GET* for all message sizes of at least 16 KiB, thus successfully eliminating the coordination overhead.

5.3.3 Discussion. In this section, we have provided a detailed evaluation of Ghostwriter against Apache Kafka as a state-of-the-art message broker. The evaluation showed that Ghostwriter scales with increasing message sizes and outperforms Kafka in throughput by at least an order of magnitude. With its exclusive protocol, Ghostwriter saturates the underlying bandwidth limit on a single partition without the need to scale out to multiple producers or consumers. Further, our optimized protocol versions reduce the coordination overhead and further improve the throughput of our system. In summary, we conclude that Ghostwriter leverages RDMA and NVM to achieve a throughput that is an order of magnitude higher than the current state-of-the-art message brokers, reducing latency by an order of magnitude at the same time. This enables it to fully leverage IB networks, which existing systems such as Apache Kafka do not achieve.

6 RELATED WORK

Modern hardware provides different ways for improving the data transfer performance in data-intensive distributed systems.

DBMS and RDMA. Binnig et al. [4] show that distributed database systems benefit significantly from modern IB networks, but they need to be redesigned with RDMA in mind. The authors propose the Network-Attached-Memory architecture, which separates computation and storage and uses RDMA to efficiently access the remote data from the compute nodes. Compared to such systems, a DMB for SPEs has very specific access patterns, i.e., writes limited to the log tail, immutable data as well as sequential read access, which we exploit in our architecture and protocol.

In addition to database systems, RDMA has been used to redesign distributed key-value stores [9, 11, 15, 18]. However, these systems optimize for random accesses that are typical for key-value stores, not the sequential accesses of a DMB.

DBMS and NVM. Over the last years, researchers have investigated ways to utilize NVM in data systems. Arulraj et al. [1] explore different storage and recovery techniques to replace traditional secondary storage with NVM. Pelley et al. [17] propose grouped commits to avoid synchronization, which improves the throughput compared to in-place updates. Additionally, Van Renen et al. [23] propose the use of NVM as a caching layer in-between faster DRAM and larger-capacity SSD storage. However, the above approaches focus on single-node systems designed for OLTP workloads, which differ highly from the access patterns of a DMB. Finally, most of the above approaches were designed without existing NVM hardware and evaluated by emulating NVM based on the assumption that NVM would perform similar to DRAM but slower. As recent research has shown, this assumption has been proven wrong for Intel's *Optane DC Persistent Memory* [5, 24, 27], and NVM performs significantly different to DRAM.

DMBs. Current DMBs [13, 14, 20, 25] adopt stateful design and use segment-based storage or partition-based storage. They all rely on disk-based persistence and use broker-centric message transfer, which increases data-access overhead. In Ghostwriter, brokers focus on coordination, while client transfer data directly to/from storage

using one-sided RDMA. This allows for parallel data transfer within a single partition while maintaining ordering guarantees. We also exploit remote-accessible NVM to guarantee instant persistence without CPU involvement.

7 CONCLUSION

In this paper, we present Ghostwriter, a disaggregated message broker utilizing RDMA and NVM. Ghostwriter reduces CPU utilization by using one-sided RDMA verbs and, thus, enables collocating the message queue with downstream stream processing engines. By optimizing write and read patterns, Ghostwriter achieves performance close to the hardware limits and is one order of magnitude faster than state of the art. Given the discontinuation of Optane NVM, we plan to explore porting Ghostwriter to Compute Express Link (CXL) based hardware in future work.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (ref. 414984028), the European Union's Horizon 2020 research and innovation programme (ref. 957407).

REFERENCES

[1] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*. ACM Press, Melbourne, Victoria, Australia, 707–722. https://doi.org/10.1145/2723372.2749441

[2] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1544–1556.

[3] Lawrence Benson, Leon Papke, and Tilmann Rabl. 2022. PerMA-bench: benchmarking persistent memory access. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2463–2476. https://doi.org/10.14778/3551793.3551807

[4] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The end of slow networks: it's time for a redesign. *Proceedings of the VLDB Endowment* 9, 7 (March 2016), 528–539. https://doi.org/10.14778/2904483.2904485

[5] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China (SIGMOD '21)*. ACM.

[6] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Rethinking Stateful Stream Processing with RDMA. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1078–1092. https://doi.org/10.1145/3514221.3517826

[7] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. USENIX Association, 401–414.

[8] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware logging in transaction systems. *Proceedings of the VLDB Endowment* 8, 4 (Dec. 2014), 389–400. https://doi.org/10.14778/2735496.2735502

[9] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM - SIGCOMM '14*. ACM Press, Chicago, Illinois, USA, 295–306. https://doi.org/10.1145/2619239.2626299

[10] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Denver, CO, USA, 437–450.

[11] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Savannah, GA, USA, 185–201.

[12] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, Paris, 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

[13] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: a Distributed Messaging System for Log Processing. *Proceedings of the NetDB* 11 (2011), 1–7.

[14] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, María Pérez-Hernández, Bogdan Nicolae, Radu Tudoran, and Stefano Bortoli. 2018. KerA: Scalable Data Ingestion for Stream Processing. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1480–1485. https://doi.org/10.1109/ICDCS.2018.00152 ISSN: 2575-8411.

[15] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, San Jose, CA, 103–114.

[16] Nikela Papadopoulou, Lena Oden, and Pavan Balaji. 2017. A Performance Study of UCX over InfiniBand. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, Madrid, 345–354. https://doi.org/10.1109/CCGRID.2017.149

[17] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. 2013. Storage management in the NVRAM era. *Proceedings of the VLDB Endowment* 7, 2 (Oct. 2013), 121–132. https://doi.org/10.14778/2732228.2732231

[18] Marius Poke and Torsten Hoefler. 2015. DARE: High-Performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing - HPDC '15*. ACM Press, Portland, Oregon, USA, 107–118. https://doi.org/10.1145/2749246.2749267

[19] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar R. Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig B. Stunkel, George Bosilca, and Aurélien Bouteiller. 2015. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *23rd IEEE Annual Symposium on High-Performance Interconnects, HOTI 2015, Santa Clara, CA, USA, August 26-28, 2015*. IEEE Computer Society, 40–43. https://doi.org/10.1109/HOTI.2015.13

[20] The Apache Software Foundation. 2020. Apache Pulsar. https://pulsar.apache.org/.

[21] Andrew Torson. 2020. Application Log Intelligence & Performance Insight at Salesforce using Flink. https://www.ververica.com/blog/application-log-intelligence-performance-insights-salesforce-flink

[22] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. 2016. On The [Ir]relevance of Network Performance for Data Processing. In *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016*, Austin Clements and Tyson Condie (Eds.). USENIX Association. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/trivedi

[23] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data - SIGMOD '18*. ACM Press, Houston, TX, USA, 1541–1555. https://doi.org/10.1145/3183713.3196897

[24] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent Memory I/O Primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN'19)*. Association for Computing Machinery, Amsterdam, Netherlands, 1–7. https://doi.org/10.1145/3329785.3329930

[25] Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, et al. 2021. Consistency and completeness: Rethinking distributed stream processing in apache kafka. In *Proceedings of the 2021 international conference on management of data*. 2602–2613.

[26] Jian Yang, Joseph Izraelevitz, and Steven Swanson. 2020. FileMR: Rethinking RDMA Networking for Scalable Persistent Memory. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 111–125. https://www.usenix.org/conference/nsdi20/presentation/yang

[27] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, Sam H. Noh and Brent Welch (Eds.). USENIX Association, 169–182. https://www.usenix.org/conference/fast20/presentation/yang

[28] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing efficient stream processing on modern hardware. *Proceedings of the VLDB Endowment* 12, 5 (Jan. 2019), 516–530. https://doi.org/10.14778/3303753.3303758

[29] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, Amsterdam, Netherlands, 741–758. https://doi.org/10.1145/3299869.3300081