Synchronizing Disaggregated Data Structures with One-Sided RDMA: Pitfalls, Experiments and Design Guidelines

MATTHIAS JASNY, Technische Universität Darmstadt, Darmstadt, Germany
TOBIAS ZIEGLER, Technische Universität München, München, Germany
JACOB NELSON-SLIVON, Google, Boulder, United States
VIKTOR LEIS, Technische Universität München, München, Germany
CARSTEN BINNIG, Technische Universität Darmstadt, Darmstadt, Germany and DFKI, Darmstadt, Germany

Remote data structures built with one-sided Remote Direct Memory Access (RDMA) are at the heart of many disaggregated database management systems today. Concurrent access to these data structures by thousands of remote workers necessitates a highly efficient synchronization scheme. Remarkably, our investigation reveals that existing synchronization schemes display substantial variations in performance and scalability. Even worse, some schemes do not correctly synchronize, resulting in rare and hard-to-detect data corruption. Motivated by these observations, we conduct the first comprehensive analysis of one-sided synchronization techniques and provide general principles for correct synchronization using one-sided RDMA. Our research demonstrates that adherence to these principles not only guarantees correctness but also results in substantial performance enhancements. This article is an extended version of [72] in which we investigate modern 400G NICs. Our findings reveal that the challenges persist even with new generations of NICs. Consequently, we turn our attention to alternative networking hardware, such as smart switches, to address some of the limitations associated with one-sided synchronization.

CCS Concepts: • **Information systems** → **Parallel and distributed DBMSs**; • **Networks** → **Programmable networks**; **Network protocols**; • **Hardware** → **Networking hardware;**

Additional Key Words and Phrases: Distributed database management systems, RDMA, synchronization

ACM Reference Format:
Matthias Jasny, Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2025. Synchronizing Disaggregated Data Structures with One-Sided RDMA: Pitfalls, Experiments and Design Guidelines. *ACM Trans. Datab. Syst.* 50, 1, Article 4 (March 2025), 40 pages. https://doi.org/10.1145/3716377

The work of Jacob Nelson-Slivon was done while at Lehigh University, USA.
This work was partially funded by the German Research Foundation priority program 2037 (DFG) under the grants BI2011/1 & BI2011/2, the DFG Collaborative Research Center 1053 (MAKI), and the state of Hesse as part of the NHR Program.
Authors' Contact Information: Matthias Jasny, Technische Universität Darmstadt, Darmstadt, Hessen, Germany; e-mail: matthias.jasny@cs.tu-darmstadt.de; Tobias Ziegler, Technische Universität München, München, Bayern, Germany; e-mail: t.ziegler@tum.de; Jacob Nelson-Slivon, Google, Boulder, Colorado, United States; e-mail: jakeslivon@gmail.com; Viktor Leis, Technische Universität München, München, Bayern, Germany; e-mail: leis@in.tum.de; Carsten Binnig, Technische Universität Darmstadt, Darmstadt, Hessen, Germany and DFKI, Darmstadt, Hessen, Germany; e-mail: carsten.binnig@cs.tu-darmstadt.de.

ACM 0362-5915/2025/03-ART4
https://doi.org/10.1145/3716377

1 Introduction

RDMA & Disaggregated Databases. **Remote Direct Memory Access (RDMA)** has quickly become one of the indispensable tools for building disaggregated database systems. Not only does RDMA provide single-digit microsecond network latencies, but it also provides efficient primitives for remote memory access. In particular, RDMA's *one-sided verbs* allow a compute server to read or write directly to a remote memory server while bypassing the remote CPU. Since memory servers frequently possess near-zero computational capacity [59] and most computational power is concentrated within the compute layer, one-sided RDMA proves to be well suited for disaggregated DBMSs. Consequently, recent literature has explored how disaggregated DBMSs can leverage one-sided verbs [6, 11, 13, 27, 35, 54, 67–70].

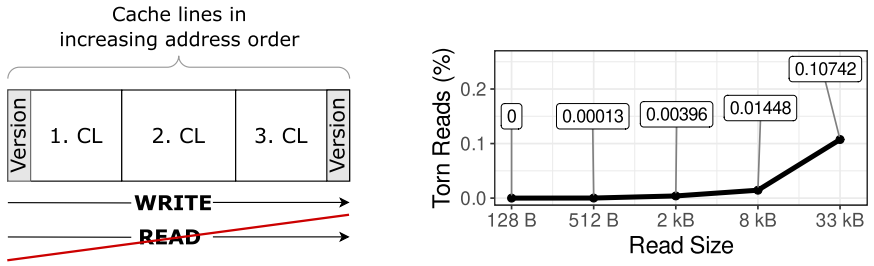
Synchronization of Remote Data Structures. A key building block of these disaggregated DBMSs are remote data structures such as one-sided hash tables [38, 60, 74], B-Trees [58, 73], or SkipLists [37] which enable efficient access to remote data. But because one-sided operations bypass the remote CPU, traditional storage server-side synchronization techniques[1] where the remote CPU is in charge do not work. Instead, various one-sided synchronization techniques have been proposed [11, 40, 64, 73]. Those techniques can be categorized into *pessimistic* and *optimistic* schemes. While pessimistic schemes *prevent* concurrent modifications, optimistic schemes *detect* (and handle) concurrent modifications. These approaches fundamentally differ in their scalability and performance characteristics.

Performance Is Key. Remote data structures may need to serve thousands of clients connecting from several compute servers. With such a high degree of concurrency, the performance depends on how well the implemented one-sided synchronization scheme performs. While individual articles have proposed various one-sided synchronization schemes [11, 39, 73], it is surprising that there has not yet been a systematic study of these schemes under comparable workloads and conditions. This article provides the first in-depth performance analysis. We show that small design choices when implementing a scheme can severely impact its performance and lead to performance bottlenecks. For example, contrary to expectations, data alignment hinders the scalability of pessimistic one-sided latches, even in uncontended workloads. In fact, if not carefully implemented, the performance for an uncontended workload can be as dismal as that for a highly contended one. To this end, this paper proposes design principles to mitigate those pitfalls and presents several optimizations that improve the performance of a well-known disaggregated RDMA-optimized DBMS [67] by 2×.

Correctness Is Hard. Achieving high performance in synchronization is unquestionably valuable, but ensuring correctness is mandatory. We have discovered that early techniques proposed in the literature fail to accurately synchronize concurrent operations, potentially resulting in hard-to-detect data inconsistencies. For example, consider an optimistic synchronization scheme as implemented in [40] (shown in Figure 1(a)), which presumes that RDMA operations occur in ascending address order—a common assumption in many articles. This scheme implements an update by writing the head version first, then the data modification, and eventually updating the tail version. Under the assumption of operations being performed in increasing address order, a concurrent reader can detect concurrent modifications by comparing the head and tail versions.

Incorrect Assumptions. Unfortunately, in contrast to the general assumption in many articles [40, 58], RDMA reads are not guaranteed to be performed in increasing address order. In fact, the RDMA specification does not state the ordering within a single RDMA read. As a result, in the previously mentioned synchronization scheme, an RDMA read may first read both versions (i.e., the first and

[1] In this paper, *synchronization techniques* refer to the low-level synchronization mechanism, i.e., latching, not higher-level concurrency control.



(a) Faulty Optimistic Scheme

(b) Percentage of Torn Reads

Fig. 1. Incorrect optimistic synchronization.

third cache line) and then retrieves the data from the second cache line. At the same time, a writer concurrently modifies the data in address order. The concurrent data update may not be detected because the versions were read first before the concurrent writer started. However, the reader and the writer overlapped at the second cache line leading to inconsistent data.

We validate the existence of this behavior with a simple experiment consisting of one storage node and two compute nodes. A single-threaded remote writer on one compute node repeatedly fills a block of its local memory, e.g., 512 bytes, with the same 8-byte version number and then writes it to a remote buffer (50 MB) on the storage node with a single RDMA write. The version number is incremented on each iteration, and the new block is written to the next slot in the buffer. Concurrently, a reader on the other compute node reads a block in the remote buffer with a single RDMA read and then checks whether the header and footer version numbers are identical. If the header and footer version numbers match, then the intermediate values are examined to determine if an inconsistency exists. "Torn" reads—having an identical header and footer version but inconsistent intermediate values—are *undetectable* by a validation scheme that checks the block's leading and trailing version numbers.

Figure 1(b) shows the percentage of how many such torn reads are undetectable due to changes in the read order. Note that no torn read appears with 128 bytes as only the header and footer cache lines are read, i.e., if they are inconsistent, this can be detected. However, inconsistencies happen for more than 128 bytes, and while not frequently, often enough to corrupt the data. Surprisingly, this problem is not widely known, and techniques that assume ordering are still very popular [58]. We believe the main reason for this assumption is that a single RDMA request requires many protocols—not only RDMA but also PCIe, and cache coherence—to work in concert. Thus, it is challenging to understand which guarantees are provided by the respective specification.

Contributions. This article is an extended version of [72].

Part 1: In the first part (Sections 2 to 4), we cover the material from the original article, where we distill general principles for correct one-sided synchronization techniques. To our knowledge, this is the first principled analysis comparing the performance, scalability, and correctness of one-sided synchronization techniques. Our work demonstrates that understanding the specification and low-level hardware details is crucial for correct and efficient synchronization. Our underlying goal is to guide researchers and developers on how and when to use the different synchronization techniques.

Part 2: Since the publication of our previous study [72], more than a year has passed. During this period, the landscape of **Network Interface Cards (NICs)** has significantly evolved. As such, in the second part (Section 5), we analyze whether and how current hardware trends can help mitigate the problems regarding correctness and performance identified in the first part. In particular,

we investigate the newest generation of Nvidia's RDMA NICs (ConnectX-7) and programmable switches (Tofino 2) that can provide scalable and correct synchronization primitives.

Artifacts: Finally, we open-sourced all benchmarks[2] that help to transfer our findings to different hardware setups and future developments.

2 Background and Methodology

While many systems [11, 39, 40, 73] implement various synchronization schemes, they do not isolate the impact of their synchronization on the overall performance. However, as we will show, synchronization techniques significantly affect scalability and system performance. This section describes the necessary background on RDMA and the typical RDMA hardware stack, existing synchronization techniques, and our experiment methodology.

2.1 Remote Direct Memory Access

Remote Direct Memory Access (RDMA) is a networking protocol that provides high bandwidth and low latency access to a remote node's main memory [46], using zero-copy transfers from the application space. Several RDMA implementations are available—most notably InfiniBand [19], RDMA over Converged Ethernet[3] (RoCE) [20], and iWARP [46]. RDMA offers four transport types: reliable or unreliable, which can be connected or unconnected. This article focuses on the *reliable connected* transport as it is the only configuration that fully supports one-sided primitives.

RDMA One-Sided Verbs. RDMA implementations provide two communication paradigms (called *verbs*) (1) *two-sided* and (2) *one-sided* verbs. Two-sided verbs are similar to traditional socket-based programming in that both sides (sender and receiver) are involved. In contrast, one-sided verbs (read, write, and atomics) provide remote memory access semantics, in which a process specifies the memory address of the remote node that should be accessed. The CPU of the remote node is not actively involved in the data transfer, i.e., only one side is involved. In this article, we solely focus on one-sided primitives. RDMA read and write enable applications to read and write remote memory directly without remote CPU involvement. To support highly concurrent applications, RDMA specifications provide atomic operations [19, 49]. One-sided **compare-and-swap (CAS)** and **fetch-and-add (FAA)** operations atomically read, modify, and write memory at a remote destination. Those operations work similarly to local CPU, CAS, and FAA instructions. CAS atomically swaps the current value with a new one if it equals the expected value. FAA increments the current value with some user-defined value and then returns the original value to the caller. RDMA atomics are limited to 64-bit, 8-byte aligned values.

RDMA Two-Sided Verbs. Two-sided verbs are widely used in high-performance RDMA systems [23, 25, 63] to send messages between distributed processes. The sender issues an RDMA send request, which consumes a waiting RDMA receive request at the destination. The receiving process issues the receive request to dictate the destination address for the sent payload. After the payload is written, the receiving process is notified of its arrival. Since the receiving process is explicitly involved in the communication, synchronization of remote processes is managed using request handlers and traditional multiprocessing approaches. In contrast, disaggregated memory systems leveraging one-sided verbs must synchronize processes directly through remote memory access, which requires careful consideration due to the behaviors we highlight in this article. In addition, two-sided RDMA requires explicit processing on the storage side, which is typically not ideal with limited computational resources on the storage servers. Therefore, two-sided RDMA is

[2] https://github.com/DataManagementLab/RDMA_synchronization

[3] RoCE is an attempt to combine RDMA with Ethernet. Refer to [16] for the shortcomings of RoCE in modern datacenters.

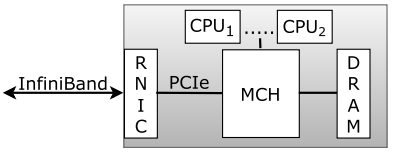


Fig. 2. Hardware components involved in RDMA.

Table 1. Pessimistic and Optimistic Techniques for One-sided RDMA Synchronization

	Variant	Ops.	Systems	Correct
Pess.	Reader/Writer	2	[6], [64], [67]	Yes
	Exclusive	2	[56], [8],[41]	Yes
Opt.	CRC	2	[51],[74],[39],[55]	Prob.
	Versioning	2	[73], [40], [58]	No†
	Cache line ver.	2	[11], [62]	Yes

†can be fixed with an additional RDMA read (see Section 4.3)

not in the scope of this article; however, several works do explore the relative merits of one-sided vs. two-sided RDMA [23–25, 63, 73]

Interface. Modern network interfaces typically provide asynchronous networking. This means that a network operation is dispatched to the NIC, which notifies the application once the operation is completed. RDMA's interface uses so-called send/receive queues to post operations: (1) While a *send queue* is used by the requester to issue operations such as read, write, send as well as atomics, (2) the *receive queue* is used by the responder to issue receive requests. With RDMA, a connection between a requester and a responder bundles these two queues and is therefore called *queue pair (QP)*. More precisely, the application must create queue pairs on both ends and connect them to initiate a connection between a requester and a responder. To issue RDMA operations, a host creates a **work queue element (WQE)**. The WQE specifies parameters such as the verb and other metadata (e.g., the remote target address). The requester then adds the WQE to its send queue and informs the local **RDMA NIC (RNIC)** via **Programmed IO (PIO)** to process the WQE. For a signaled WQE, the local RNIC pushes a completion event into the **completion queue (CQ)** via a DMA WRITE once the remote side has processed the WQE. To enforce synchronous communication, the application can also block until the network card generates the completion event. Typically, RDMA is used asynchronously, meaning multiple WQE are simultaneously registered with the RNIC, and later the application checks for completions in the CQ. This technique is often referred to as *doorbell batching*.

2.2 RDMA Hardware Stack

As mentioned in Section 1, synchronization techniques rely on certain hardware guarantees; thus, we briefly present the required hardware in Figure 2. An RNIC connects via the PCIe bus to the **memory controller hub (MCH)** of a multi-core CPU [44]. The MCH is responsible for handling memory access by both the CPU and peripheral devices. Among others, the MCH contains the memory controller, the coherency engine, and acts as the root complex for the PCIe bus. Because modern server architectures implement cache coherent I/O, e.g., Intel DDIO [21] and ARM CCI [1], knowing that an RDMA access to system memory will snoop (look up) CPU cache for conflicting addresses is essential. If a conflicting address is found, it can be served from the CPU cache; otherwise, it is fetched from main memory. Cache coherent I/O is a core enabler for many of the techniques described in Section 4.

2.3 Existing Synchronization Techniques

The one-sided synchronization techniques of modern distributed systems can be categorized into pessimistic and optimistic approaches, as shown in Table 1. Pessimistic approaches mirror traditional latching with the distinction of having only a single latch mode or two for reader-writer support; meanwhile, optimistic approaches can be further subdivided. Each optimistic technique offers different characteristics, like memory and computation overhead, but all embed metadata in

the object, which is updated by writers and validated by readers. We now briefly introduce existing optimistic techniques from Table 1: *CRC*, *versioning*, and *cache line versioning* before we discuss them in more detail in Section 4.

Checksums. A common detection technique used to identify potential inconsistencies [39, 51, 74]. After updating the object, writers write back a new checksum (typically a 64-bit CRC). Readers then recompute a checksum locally and compare it to the observed checksum. While effective in practice, there is still a non-zero probability of a collision causing validation to succeed on corrupted data.

Versioning. Versioning is a strategy that augments the data with a single version. During execution, writers increment the version before and after updating the data. Readers read the version before and after their operation to validate whether an update occurred concurrently. Optimistic lock coupling is a special case of versioning in which sequence locks are used for both write-write synchronization and validating reads (e.g., [29] and [73]). As we will explain in Section 4.3, there are pitfalls with this approach that impact its correctness.

Cache Line Versioning. Finally, cache line versioning maintains the object's version in each constituent cache line [11]. Writers increment all versions before updating the data, then increment them upon completion. Unlike versioning, readers detect inconsistencies with a single remote read by validating that all versions match.

To understand the differences between the sundry approaches, we perform a principled analysis and evaluation of one-sided synchronization techniques. To the best of our knowledge, this article is the first to do so.

2.4 Evaluation Methodology

Framework. We implemented all techniques in the same code base to isolate the fundamental properties of the synchronization schemes from incidental differences. We highlight each synchronization scheme's overhead using a perfectly sized remote hash table to avoid hash collisions. In addition to the hash table, we use a remote B-Tree to show how the synchronization schemes behave when contention is inherent to the data structure due to its hierarchical nature (all workers start their traversal at the root node). We use multi-threading and execute one operation on a single thread (worker) until completion, i.e., no batching or asynchronous execution. This allows a fair comparison of the approaches.

Setup. We conducted all reported experiments on an 8-node cluster running Ubuntu 18.04.1 LTS, with a Linux 4.15.0 kernel. All nodes are connected to a SB7890 InfiniBand switch using one Mellanox ConnectX-5 MT27800 RNIC (InfiniBand EDR 4x, 100 Gbps) per node. Each node has two Intel Xeon Gold 5120 CPUs (14 cores) and 512 GB main-memory split between sockets.

Since the ConnectX-5 is connected to one NUMA socket, we inevitably have NUMA effects when using more than 14 cores (i.e., 28 threads). We allocate the memory on the socket of the NIC, and to alleviate the NUMA effects, we assign threads round-robin (interleaved) to both sockets. More details of NUMA effects on RDMA can be found in [42] and [47]. Besides ConnectX-5, we also validate our results for different generations of RNIC, namely, ConnectX-3 and ConnectX-6. Mellanox is widely used; from 30 articles, we recently analyzed 28 used Mellanox cards. Besides on-premises, only Microsoft offers RDMA of the top three cloud providers and they use Mellanox cards in their instances. [26]. That being said, we believe that most findings are independent of the network card. The correctness discussion mainly depends on the protocol specifications underpinning the RDMA communication. In particular, our testbed leverages the widely-used InfiniBand specification [19] which shares commonalities with alternative RDMA protocols and make our results applicable to a broader range of deployments. The performance considerations shed light on possible pitfalls worth investigating when building an RDMA application. We open-source our

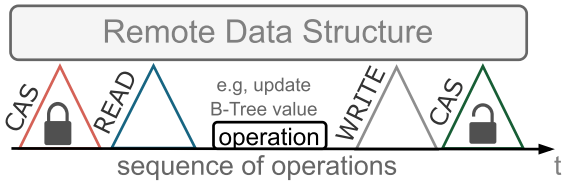


Fig. 3. Example of an exclusive latch acquisition.

benchmarks to help developers uncover performance and correctness bottlenecks in other RDMA system configurations.

3 Pessimistic Synchronization

In order to prevent concurrent modifications of remote data structures such as a B-tree or a hash table, one-sided pessimistic synchronization techniques implement latches using one-sided RDMA operations. In this section, we present the basic implementation of such a latch and use it as a running example to discuss possible optimizations. Since RDMA atomics are the fundamental building block of these one-sided latches, we will then drill down into the performance and scalability characteristics of these RDMA primitives. Afterward, we outline and evaluate possible optimizations for one-sided latches.

3.1 Running Example of a Pessimistic Latch

Table 1 shows that existing pessimistic schemes are subdivided into two types of latches: While some latches such as RCC-NOWAIT [56] support only one latch mode (latched and unlatched), others distinguish between shared and exclusive modes, i.e., reader/writer latches. Both latch types can be implemented with atomic RDMA operations. We will use a reader/writer latch for the running example, but all optimizations generalize to both latch types.

We implement a typical reader/writer latch using an 8-byte value [36, 64]. A worker uses an RDMA **compare-and-swap (CAS)** operation to set the latch bit (usually the trailing bit) for exclusive access. Readers increment the reader count, encoded in the remaining bits, with a **fetch-and-add (FAA)**. In the basic variant, all operations are executed synchronously, i.e., the worker blocks after every operation until its result is returned.

Figure 3 gives an intuition on how this latch is used to access a remote data structure exclusively. First, the remote data structure is latched with an RDMA CAS operation on the 8-byte latch by setting the lock bit. Afterward, the desired data is read from the data structure, modified locally, and written back with an RDMA write operation. Finally, the remote data is unlatched with another RDMA CAS operation on the 8-byte latch.

To access the remote data structure in shared mode, the clients use RDMA FAA to increment the reader count of the latch speculatively. The return value (i.e., the full 8-byte) of the operation allows the worker to check if the latch is in the exclusive mode, in which case the worker decrements the reader count and retries. Otherwise, the worker reads the data using an RDMA read and then decrements the counter to unlatch.

3.2 Performance of RDMA Atomics

Because every pessimistic approach relies on RDMA atomics (CAS and FAA), it is important to understand their isolated performance before discussing how our basic latch can be optimized.

Uncontended vs. Contended RDMA Atomics. In the first experiment, we examine the scalability behavior of contended and uncontended RDMA atomics. Both scenarios are equally vital, and while heavy contention is typically rare, it is unavoidable in some workloads, e.g., having hot

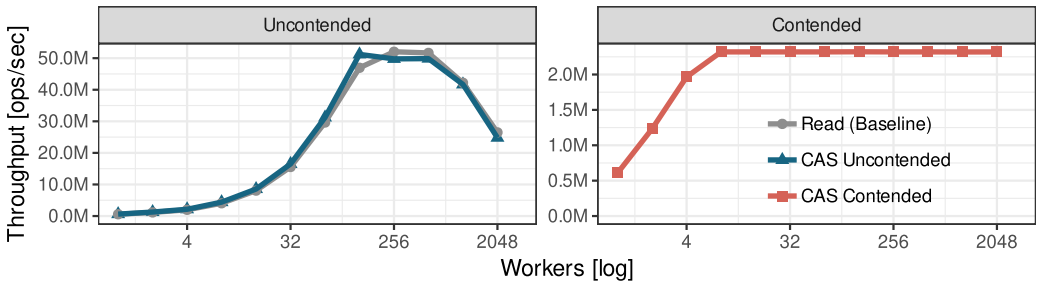


Fig. 4. Scalability of contended and uncontended atomic RDMA operations when increasing the number of workers (4 compute nodes and 1 storage node).

tuples. To show the effect of contention, we perform an experiment in which all workers issue an RDMA CAS operation of the same 8-byte atomic counter. To understand how uncontended atomics scale, we assign a private 8-byte remote atomic counter to each worker. For reference, we compare uncontended atomics to the performance of an RDMA read.

Figure 4 shows the scalability behavior of both uncontended and contended RDMA atomic operations when increasing the number of workers. As we can observe, uncontended atomics scale like one-sided RDMA read operations, peaking at around 51.2 million operations per second with 128 workers. This suggests that the parallel uncontended atomic requests do not interfere, but we will demonstrate later that this does not hold for all scenarios. Note that the performance drop of uncontended atomics at 512 workers has nothing to do with the atomic operation but can be attributed to QP-thrashing [11] on the client machines. QP-thrashing means that the QP state cannot be cached on the RNIC and is swapped in and out to host memory. This occurs at around 128 utilized parallel QPs per client NIC in our hardware, i.e., 512 workers.

As expected, when running the contended atomic workload, the peak is significantly lower at 2.32 million operations per second and atomic operations only scale until 8 workers.

In the literature, RDMA atomics seem to have a bad reputation for being fundamentally unscalable [24]—even for uncontended workloads. However, the above experiments demonstrate that uncontended atomics scale well w.r.t. the number of workers. In fact, they show comparable scalability to RDMA read operations. While this experiment offers valuable insights into the scalability of atomics, it is not the complete picture, as we will demonstrate.

Atomic Stride Size and Alignment. Obviously, the scalability of RDMA atomics depends on the degree of parallelism. However, subtle details such as the data alignment can also interfere with the scalability. So far, in our experiments, values are placed in a 64-byte stride, i.e., a cache line. We only used the first 8 bytes for the atomic counter and ignored the remaining 56 bytes. However, in practice, RDMA atomics are placed at larger strides as they protect data of various sizes, e.g., a 4 KB B-Tree node.

In the following experiment, we measure the effect of larger stride sizes by varying the distance between the atomic counters. As in the previous uncontended experiment, each worker has a private latch to avoid contention. Consequently, the expected outcome should be similar to the uncontended result in Figure 4. Surprisingly, Figure 5 shows that the stride size impairs the scalability tremendously. That is, with larger stride sizes, the inflection points w.r.t. throughput (highlighted in red) are reached earlier. With a 64-byte stride, the peak performance is 50M operations with 128 workers, i.e., the same upper-bound as in Figure 4. With a 256-byte stride, the peak performance is at 40M operations with 128 workers. With a 512-byte stride, the peak performance is halved and reached with fewer workers (20M operations with 64 workers). Remember that there is no true

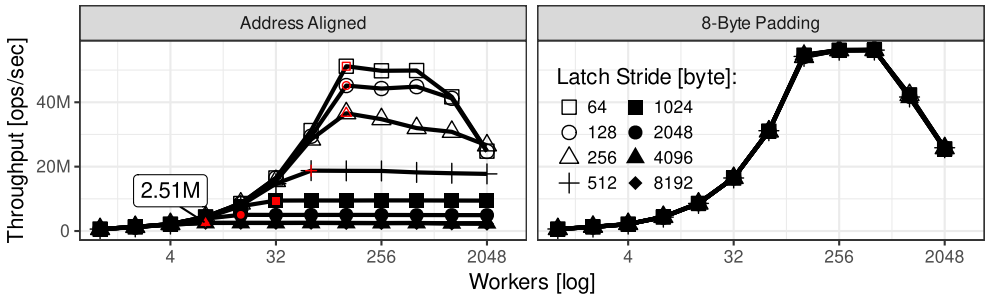
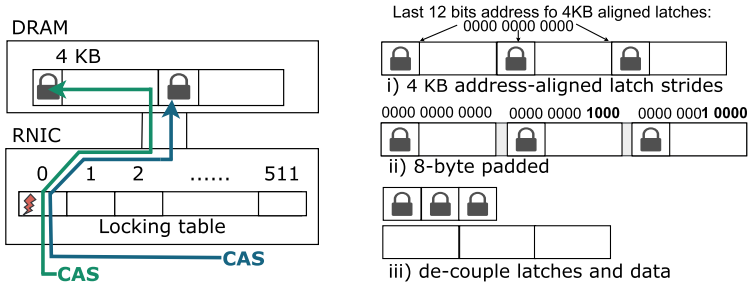


Fig. 5. Scalability of uncontended atomics with varying strides between the latches (4 compute nodes and 1 storage node).



(a) RNIC Internal Locking Table (b) Stride Size and Alignment

Fig. 6. Pessimistic synchronization performance can be impacted by RNIC architecture and by host memory layout.

latch contention, and we only vary the distance between the atomic counters and nothing else. The observed behavior must be based on a physical contention point in the RNIC.

NIC Internals - Physical Contention. Through reverse engineering, we believe that the RNIC uses an internal locking table to serialize atomic operations. Since the locking table works similarly to a hash table, collisions can happen. Unrelated atomic operations can be assigned to the same slot, severely limiting the throughput of concurrent uncontended atomic operations. The lock slots are determined based on the last 12 bits of the atomic operation's target address. For instance, if we use a 4 KB-aligned address as illustrated in Figure 6(b)(i), the last 12 bits of the address are zeros, and they are assigned to the same lock slot. Even though the atomics do not contend on the same latch, the way the RNIC handles atomics results in physical contention inside the locking table, as exemplified by the arrows in Figure 6(a). The two CAS operations target different latches and are still serialized in the same lock slot, negatively impacting performance. Consequently, logically uncontended atomic operations do not scale very well when the data alignment is not carefully chosen, as shown in Figure 5. When we compare the results of our initial contended scalability experiment from Figure 4 with the performance of 4 KB-aligned stride sizes, we can see that the performance and scalability characteristics are very similar. Given the underlying hardware mechanism, this performance is now explainable: the operations are serialized in the same lock slot, whether through a collision of the address or the operations targeting the same latch. We validated the existence of a performance signature matching our hypothesized 12-bit lock table in three RNIC generations: ConnectX-3, ConnectX-5, and ConnectX-6 RNICs. Also, Kalia et al. [24] observed similar findings for an older network card (Connect-IB). However, it is hard to generalize

our findings to all NICs since the implementation details are not made publicly available by the RNIC vendors. Therefore, like other articles, we can only infer implementation details [24, 58]. Instead, we emphasize that NIC hardware details are essential and demonstrate potential bottlenecks that should be carefully evaluated when building a high-performance system.

Improving Scalability of Uncontented Atomics. To improve the scalability of uncontended operations, we must avoid collisions in the locking table. The only way one can control this is through the data layout, i.e., the addresses of our latches. The goal is to place the latches so that the last 12 bits (used for the lock-slot calculation) are different. Consider the example in Figure 6(b)(ii); instead of allocating the 4 KB blocks back-to-back, a padding of 8-byte is placed before the latches. Now, the last 12 bits of the latch addresses are not all zeros and, thus, will be assigned to different lock slots. The effect of this mitigation technique is illustrated in Figure 5. We can observe that all stride-sizes scale equally well (all measurements happen to be on the same line). Another possibility to better utilize the lock-slots is to decouple latches from the data as depicted in Figure 6(b)(iii). In this technique, the latches are placed back-to-back. Since the latches are only 8 bytes, the last 12 bits of the latch address will vary and achieve the same scalability behavior as for the 8-byte padding. Note, in this experiment, we test the performance of the atomic operations, i.e., we do not read the data. This allows us to isolate the atomic contention effect, but in practice separating the latch from the data may have adverse effects due to locality. In particular, the translation from physical-to-virtual memory could suffer as every data access invokes two translations, i.e., one for the latch and one for the data. However, this depends on other parameters, such as the working set, tuple size, and NIC-cache size [24], and thus requires a holistic evaluation.

To conclude, despite contrasting beliefs, RDMA atomics can scale well w.r.t. the number of workers. However, the scalability depends on the number of concurrent accesses (contention) and the data alignment. While the first is workload-dependent, the latter can be carefully tuned to best utilize the internal RNIC hardware. While we demonstrate that padding is beneficial for RDMA atomics, it can also have consequences elsewhere, such as making page table lookup less efficient. Therefore, we argue that it is crucial to understand the internals of the RNIC and holistically optimize the DBMS system.

3.3 Optimized Pessimistic Latches

Equipped with our findings on how to utilize atomics optimally, we can now focus on how to design optimized pessimistic latches. Recall that the basic latch variant executes all operations synchronously, as illustrated in Figure 3. After every operation, the completion is awaited by polling the completion queue (cf. Section 2). While this is certainly correct, it is inefficient and increases the per-operation latency.

Latch optimizations. To reduce the operation latency, many systems use optimizations. Unfortunately, those optimizations are often only briefly discussed by the authors. We compiled a list of existing optimizations following numerous small hints in related work and a careful study of published source code. This is the first paper that describes these optimizations in detail and discusses why those optimizations guarantee correctness. In the following, we present those optimizations and highlight possible pitfalls.

The *speculative read* optimization overlaps the latch with the read operation (cf. Figure 7) to hide the latency of the read. If the latch request is successful, we can proceed, and if it is unsuccessful, the latch request is restarted. However, the correctness of this optimization relies on the order of the operations. That is, it must be guaranteed that the latch operation takes place before the read happens. Fortunately, RDMA atomics are executed prior to subsequent RDMA operations on the same QP as per the InfiniBand specification [19].

Similarly, *write combining* overlaps the write and the unlatch operation (CAS) as illustrated in Figure 7. The intuition is that the unlatch operation can mask the write latency. This optimization is only applicable for exclusive access since read-only operations do not update the data and thus do not involve an RDMA write.

The *asynchronous unlatch* optimization is an optimization that goes even further and does not wait for the last completion event synchronously and thus immediately processes the next operation. In contrast to the synchronous variants, however, the memory buffer containing the write's payload cannot be reused immediately for the next operation. The issue is that when immediately re-using the buffer, the data from the previous operation could be overwritten as the previous RDMA write may still be outstanding; as such, we need to ensure that the operations are finished before re-using the buffers. The typical solution to avoid overwriting in-flight buffers is to use multiple buffers per QP. For instance, if the worker wants to modify two tuples, then the first tuple is written to the first (local) buffer. Subsequently, the remote content of the first tuple is updated and asynchronously unlatched. Since the remote operations are executed asynchronously, the RDMA operations (CAS and write) may still be outstanding, and the first buffer should not be re-used for the second tuple. Therefore, the second tuple is is written to a second (local) buffer. Using separate buffers gives the outstanding operation from the first buffer time to complete without any risks of overwriting the content. The first buffer can be safely re-used when the second operation on the same QP generates a completion event, i.e., after the payload is read.

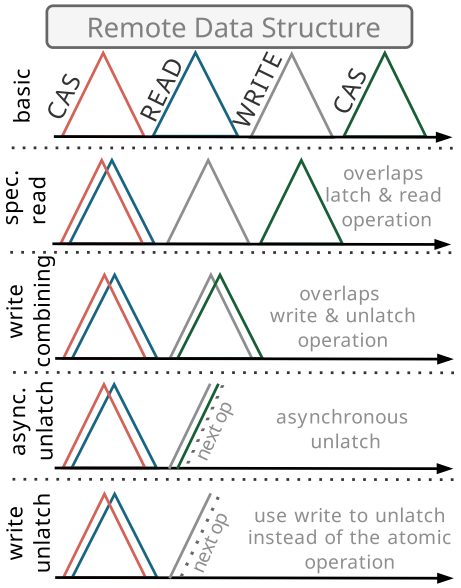


Fig. 7. Evolution of exclusive latch optimizations.

Write unlatch is an optimization that relies on the fact that RDMA writes are performed in increasing address order. This optimization often works in practice but is similar to RDMA reads, not specified by the RDMA standard. However, because many essential applications such as MPI [33] and many other systems rely on last-bit polling [11, 14, 37, 71], RNIC vendors typically implement RDMA writes in increasing address order [11] for compatibility reasons. Note that the latch must be located at the highest address (typically as a footer) to protect the data until the full write has been completed. For example, the last 8 bytes of the value of an item residing in an RDMA-enabled key-value store could encode the lock protecting it. The payload of an RDMA write to update the value would be suffixed by an unlocked value to also release the lock. Since this optimization uses the write to unlatch the data, it saves a complete atomic operation. Unfortunately, there is a catch; the optimization only works in *certain cases*. The InfiniBand specification makes no guarantees that non-atomic and atomic RDMA operations are ordered and visible to each other when issued from different QPs [19]. That means that the RDMA write that unlatches the data item may not be visible to subsequent atomic operations from other workers. This can lead to behavior that might seem surprising because sequential consistency is not guaranteed.

We believe this is because the RNIC can buffer atomic operations for fast completion in a special on-chip buffer. When an atomic operation arrives at the RNIC, the cache line in which the value is stored, is read into this buffer from the memory. But because there is no guarantee

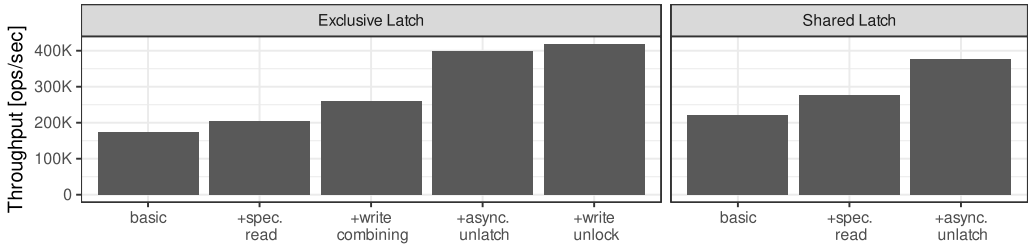


Fig. 8. Single-threaded ablation study of latch optimizations (1 compute node and 1 storage node).

w.r.t. interference of non-atomic operations, it may happen that once the atomic operation reads the current value in the buffer, an RDMA write changes the value on the cache line, i.e., unlatch. To make this more concrete, assume the data item is exclusively latched. Now, an atomic FAA operation wants to increment the reader count. The value is read to the RNIC buffer, but in the meantime, the latch-holder unlatches the data item with an RDMA write. Unfortunately, there are now two incoherent states: (1) in the RNIC buffer, the latch is still latched (2) in the cache line, the latch is unlatched. The RNIC increments the old (latched) value and overwrites the unlatch state on the cache line. Therefore, the original unlatch operation is lost, and the latch will remain latched, leading to a deadlock. Hence, the write unlatch optimization cannot be used with FAA operations and only works in combination with CAS operations, as used in RCC-NOWAIT [56]. The reason is that CAS operations conditionally overwrite the state. The operation detects that the old state is still latched and does not modify the latch. Thus, a concurrent worker may detect the unlatch operation delayed for the incoherent states, but eventually, the cache coherence protocol ensures that the RNIC sees the newest version. Consequently, creating a reader/writer latch in combination with the write unlatch optimization is impratical and only one latch mode can be supported (no distinction between reader and writer) (cf. Table 1 RCC-NOWAIT).

3.4 Ablation Study of Latch Optimizations

To better understand how the latch optimizations perform, we first enable them step-by-step in a single-threaded ablation study. We show the throughput for exclusive and shared latch acquisitions in Figure 8. The numbers include all necessary operations: *CAS, read, write, and CAS* for a modification, and *FAA, read, FAA* for a read operation. As mentioned, some optimizations are only applicable for the exclusive latch acquisition. For the analysis, we repeatedly latch and perform the operations on a 256 byte-sized tuple placed on a storage node.

We can see that all exclusive latch optimizations in Figure 8 (left) increase the single-threaded performance. One of the most effective optimizations is asynchronous unlatch since we can immediately start the next operation. In contrast, write unlatch does not significantly increase the performance further. Combined with the fact that it is impractical to implement a reader/writer latch, we will not further consider this optimization in this section. However, we will leverage this optimization again in Section 4.

When focusing on the shared latch performance in Figure 8 (right), we can observe that the basic performance is higher since the read consists of only three sequential operations. The advantage of fewer operations vanishes with the higher optimization levels as they aggressively overlap messages (cf. Figure 7). Eventually, the performance of exclusive and shared latch acquisitions converges with higher optimizations and becomes latency bound.

Scalability of Latches. So far, we have focused on the single-threaded performance for pessimistic latch acquisitions—however, most disaggregated database systems run with multiple

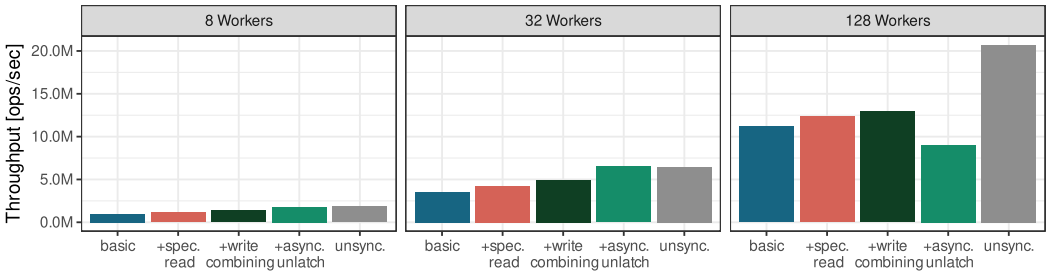


Fig. 9. Multi-threaded performance of optimized latches (4 compute nodes and 1 storage node).

workers dispersed across several compute nodes. Thus, in the following, we discuss how the optimizations perform with an increasing number of workers equally distributed across 4 compute nodes. We use a data set with 20 thousand 256 byte-sized tuples stored on one storage node. We also measured with 20 million tuples, but the results behaved similarly. The accesses are randomly distributed across the dataset.

We only show the performance of the write-only workload, i.e., the exclusive latch optimizations, since the read-only workload behaves very similarly in this experiment. As the upper bound, we include an unsynchronized version (an RDMA read and RDMA write). Figure 9 shows the results for 8, 32, and 128 workers. The first observation is that the performance is on par with the unsynchronized version when all optimizations are enabled, i.e., *asynchronous unlatch*, for 8 and 32 workers. The optimizations effectively hide the latch operations with the data movement operations. Furthermore, the performance scales near-linear when increasing the worker count from 8 to 32. However, with 128 workers, the highest optimization level seems to be counterproductive. We attribute this behavior to the fact that the next operation may start sooner. Acquiring the next lock earlier introduces additional lock contention in the presence of more workers. For instance, one worker already acquires the next lock even though the old lock is still latched due to the asynchronous nature, i.e., a worker can hold two locks for a limited period. It can also lead to increased RNIC contention on the storage node, with more in-flight operations. In practice, the *asynchronous unlatch* optimization is still worthwhile since there are typically more storage nodes, as we will see in the following experiment.

3.5 Effect on a Disaggregated DBMS

As stated earlier, latch optimizations improve performance tremendously. However, the previous results were only obtained in micro-benchmarks. Let us substantiate that claim by integrating them into an existing disaggregated DBMS. We use NAM-DB [67], which is a disaggregated RDMA-optimized DBMS. In our setup, we use 4 compute nodes with 28 workers each (112 total workers) and 4 storage nodes. Among those storage nodes, we distribute 20 million tuples. We measure the throughput in operations per second for representative tuple sizes of 256 and 512 byte. We implemented the highest optimization level., i.e., *asynchronous unlatch*, and call this version NAM-DB++. In addition, we show the effect of manipulating the latches' data layout, as we discussed in Section 3.2. We compare the original NAM-DB with NAM-DB++ in a write-only, mixed, and read-only workload. To show the effect of contention, we vary the access skew.

Read-Only Performance. Let us first focus on the 256 byte-sized tuples and the read-only workload (top right in Figure 10). The optimizations double the performance with uniform and slightly skewed (Zipf 1) access patterns. When the contention increases, the hardware limits the performance, and both systems converge. The dramatic performance degradation is nevertheless surprising in the read-only scenario. In theory, the read-only performance should not dramatically

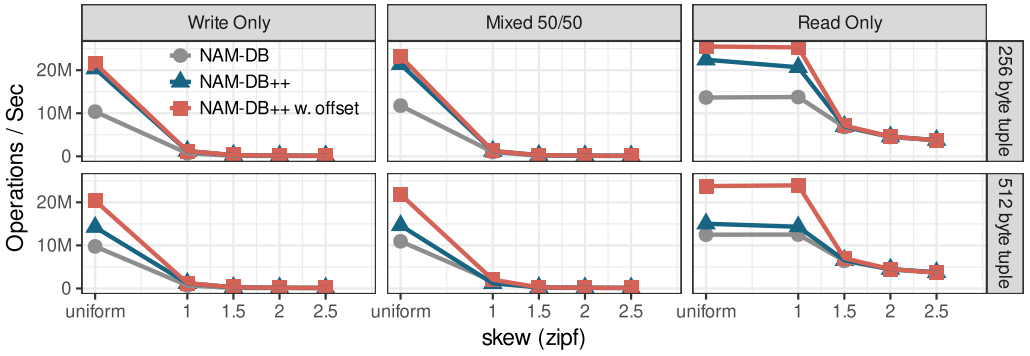


Fig. 10. Effect of optimizations in NAM-DB (4 compute nodes and 4 storage nodes).

collapse since multiple readers can acquire a latch simultaneously. Unfortunately, the RNIC cannot handle the concurrent atomic RDMA operations necessary to acquire the reader latch in the first place. As mentioned in the previous experiment, the RDMA atomics are serialized in the RNIC, which does not scale well when the lock slot is contended. Despite the hardware limitations, the read-only workload still performs much better than the write-only workload under high contention. For instance, with a Zipf factor of 2 the write-only performance is $170K$, whereas the read-only performance is $4.5M$.

Write Performance. The contention exacerbates the performance issue in the write-only and mixed workload. The locks now logically contend in addition to the physical contention on the RNIC. In other words, the performance of the atomic operations decreases since those operations often target the same latch and are serialized in the same RNIC lock slot. Once the RNIC processes the atomic operation, the latch may have been already latched exclusively, which requires a restart and aggravates the problem.

Larger Tuples. When looking at the 512 bytes-sized tuples in the read-only workload, we can see that the effect of applying the padding is more pronounced. As mentioned earlier, the larger latch strides lead to more contention inside the RNIC's lock-table, thus limiting the performance. We use 4 storage nodes, as opposed to the experiment in Section 3.2, and still, the collisions inside the RNICs become the primary bottleneck. Therefore, with the padding optimization, the performance significantly improves by removing the bottleneck of physical contention, allowing the latch optimizations to achieve their potential.

4 Optimistic Synchronization

In the previous section, we have seen that pessimistic synchronization scales when the latches are uncontended. However, in some data structures, latch contention is unavoidable and, in fact, inherent to the data structure design. For instance, B-Tree operations have to traverse the B-Tree root node. Although the root node is mainly latched in shared mode, this creates (physical) contention on the RNIC when using pessimistic synchronization, negatively impacting the performance. Figure 11 shows this effect for a B-Tree with 4 KB nodes stored on a single storage machine

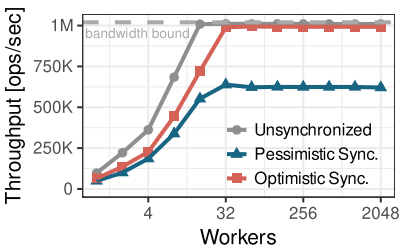
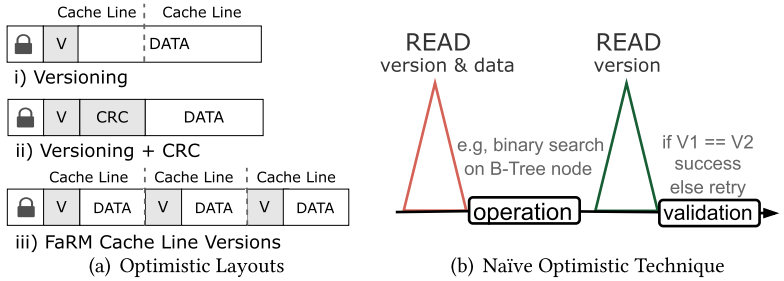


Fig. 11. Lookups in one-sided B-Tree (4 compute nodes and 1 storage node).



(a) Optimistic Layouts (b) Naïve Optimistic Technique

Fig. 12. Implementation of optimistic synchronization.

and accessed read-only from 4 client machines. As we can observe, the unsynchronized B-Tree can saturate the bandwidth with only 16 workers, whereas the pessimistic synchronized version stagnates much earlier and never achieves the full bandwidth. Note that the pessimistically synchronized version includes all the optimizations presented above, and even then, the performance is disappointing in this experiment.

This is why many papers eschew RDMA atomics and propose an optimistic synchronization scheme in which reads do not physically latch the data but detect concurrent modifications. In contrast to pessimistic synchronization, Figure 11 demonstrates that optimistic synchronization can achieve the full bandwidth. We start this section by providing an intuition on how optimistic synchronization works. After that, we discuss PCIe's ordering guarantees and its devastating effects on some optimistic schemes leading to an incorrect synchronization. We then present correct optimistic synchronization schemes andgt evaluate their performance.

4.1 Intuition for Optimistic Synchronization

Optimistic Reads. The intuition for optimistic synchronization is that readers proceed optimistically and then validate, while writers physically acquire an underlying pessimistic lock to avoid write-write conflicts. To achieve the same guarantees as shared pessimistic latches[4] readers must check that the data item did not change during their operation. This is typically realized by augmenting the data item with a version counter and incrementing it upon each modification, which allows readers to detect concurrent writes. The layout of such an augmented optimistic lock is shown in Figure 12(a). It consists of a pessimistic latch used for exclusive access and the version counter. Using this version counter, readers validate that the version did not change during their operation. If the validation fails, the operation is restarted.

Naive Implementations. One way of implementing a one-sided optimistic lock is depicted in Figure 12(b), which we call the *naïve implementation*. This approach uses a single RDMA read to copy the latch, version, and data to the worker. The worker can then check if the item is exclusively latched and possibly restart. Otherwise, if the check on the latch succeeds, the operation, e.g., a binary search in a B-Tree node, is performed optimistically. Once the operation is finished, the version is read once more (via RDMA) and compared to the initial value.

This after-the-fact validation is crucial to detect concurrent modifications and to get the same guarantees as a pessimistic latch. Thus, the validation is effectively equivalent to an unlatch operation. These semantics are critical for higher-level synchronization techniques such as optimistic lock-coupling [30, 73] on a B-Tree since we need to ensure that the B-Tree node did not split and the child node is still valid.

[4]We will discuss relaxed guarantees in Section 6.

4.2 PCIe's Ordering Guarantees

Intermediate Protocols. Unfortunately, the naïve implementation in Figure 12(b) is not correct. As pointed out by Taranov et al. [53], there are three factors that can affect the data delivery order of transmitted messages: (1) Message ordering, (2) packet ordering within a single message, and (3) DMA ordering. The first two factors are generally ensured by InfiniBand and RoCE.[5] But even though message ordering is guaranteed, it is not guaranteed that DMA operations are performed in-order. The InfiniBand RDMA specification [19, 20] does not itself provide any ordering guarantees concerning the order of bytes read by an RDMA read. In other words, RDMA operations are not designed with concurrency in mind (except for RDMA atomics). However, in practice, many academic and industry-grade systems use RDMA concurrently on the same memory regions. Since RDMA does not specify the behavior of those concurrent accesses, the intermediate protocols are important. For example, the lack of order for reads permits intermediate protocols involved in the operation, e.g., PCIe, to retrieve host memory as they see fit. Therefore, to fully understand why an RDMA read may not execute in increasing address order, it is critical to investigate the impact of PCIe [44] and cache coherence protocol. Understanding the underlying protocols subsequently allows us to extract correct synchronization techniques.

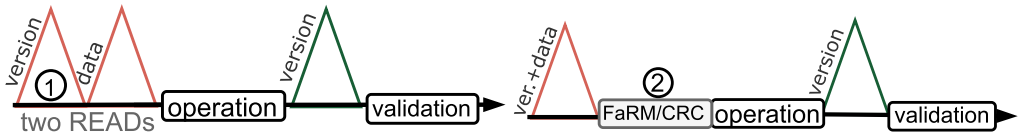
As shown in Section 2, an RDMA read request is sent over the network to the remote node. The RNIC then dispatches the request to the PCIe controller, which fetches the requested data region from the host memory. The data is transferred via PCIe to the RNIC and then sent back to the requester in one or more RDMA packets. An important aspect is that PCIe requests serviced by the host are cache coherent on modern server architectures. Modern architectures provide *direct cache access* [18, 31, 52] to allow high-performance I/O such as RDMA to access processor caches directly. For example, the x86 machines in our testbed are equipped with Intel's DDIO [21], and a similar mechanism is available for ARM [2].

PCIe Reordering. Since the actual data transfer from the remote memory to the remote RNIC is initiated and carried out via PCIe, we must look to the PCIe specification. RDMA requests are translated to PCIe transactions consisting of reads and writes that are processed by the so-called PCIe root complex. Hence, the guarantees provided at this layer of the hardware stack play a pivotal role. The root complex services PCIe read requests, which are coherent at a cache line granularity. Once a request is initiated, the PCIe protocol transfers that data to the endpoint via so-called *completions*.

Multiple completions are used for reads larger than a given size, e.g., 64 bytes. Herein lies the problem. The PCIe specification states, "Memory Read Request serviced with multiple completions, the completions will be returned in address order." [44]. This is hard to interpret, but it only determines the order of the completions and not the order in which the data is retrieved from memory. In fact, an implementation note permits that a "single Memory Read that requests multiple cachelines from host memory is permitted to fetch multiple cachelines concurrently" [44].

Implications on Correctness. Due to the concurrent cache line retrieval, we cannot reliably detect concurrent modifications with our naïve implementation from Figure 12(b). For example, assume a concurrent writer and a reader access a data item as depicted in Figure 12(a)(i). With the lack of order, the reader could first retrieve the second cache line in which the writer currently modifies data. In the meantime, the writer increments the version and unlatches the data item again. Only then does the reader retrieve the first cache line containing the latch and the version counter. Consequently, despite the validation step at the end in which it will retrieve the version again, the reader will falsely assume that the version did not change. In our introduction, we have already

[5] There are RNICs that deviate from this and offer out-of-order delivery [9].



(a) Two RDMA reads (b) FaRM and CRC technique

Fig. 13. Correct optimistic techniques.

shown that this is a real risk for modern RNICs (cf. Figure 1(b).) This may be surprising for two reasons: (1) The PCIe standard guarantees that RDMA writes are performed in increasing address order while reads can be read out of order, as discussed before. (2) many articles rely on this unspecified behavior. In fact, an earlier article of ours [73] suffered from this wrong ordering assumption. More details on the memory semantics of RDMA can be found in [10].

Referring back to Figure 1(b), we can see that data is rarely corrupted, but it happens, and if data corruption occurs, it is almost undetectable. We tested this behavior on three generations of modern RNICs, including the RNICs available in the cloud, and observed that no RNIC provides additional ordering guarantees beyond the PCIe specification. Fortunately, there exist techniques that prevent this issue, but, as we will see in the following section, they are not for free and come with different performance characteristics and tradeoffs.

4.3 Correct Optimistic Synchronization

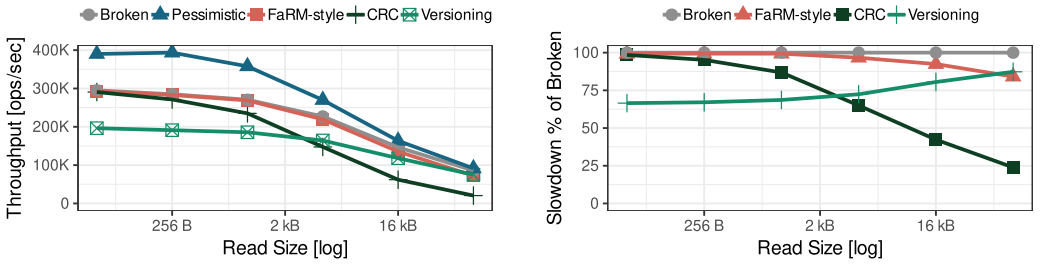
Optimistic synchronization relies on the initial RDMA read to observe a consistent view of the version and data. Due to the lack of ordering at the PCIe bus, we must rely on an additional mechanism to provide this capability. Only then the after-the-fact validation will correctly detect concurrent modifications. We will discuss existing mechanisms[6] in the following.

Versioning (Using Two RDMA Reads). This technique is not very different from the naïve version; however, it requires two dedicated RDMA reads in the beginning, as shown in Figure 13(a). The first RDMA read targets only the latch and the version to ensure correct serialization, and only the second reads the data. Because the version is always read before the data, every concurrent update will be detected. When looking at Figure 13, one may wonder if the two reads can be overlapped similarly to the operations in Figure 7. Unfortunately, this is not possible with two RDMA reads since they could be re-ordered at the PCIe level or even in the network. This is due to another unexpected pitfall: Although RDMA operations are ordered in a connected queue pair, the ordering only holds for the RDMA completion events. There are some exceptions in which the order is ensured, e.g., for atomic operations. On the other hand, the two reads must be issued sequentially.

Besides the correct order of the two reads, the latch and version must be stored in one cache line to exploit the cache coherence protocol and read both consistently. Only this enables a reader to detect concurrent modifications reliably. Unfortunately, this technique needs two RDMA reads, which may be expensive. The following two approaches only require a single RDMA read by embedding additional metadata in the object as shown in Figure 12(a).

Checksum by CRC. This scheme detects inconsistencies by using a checksum in the data, e.g., CRC64, that allows the worker to verify the data with high probability. The CRC will not match the corrupt data if there is a concurrent writer. Therefore, in the best case, only one RDMA read is required (cf. Figure 13b)). The downside, however, is that (1) the CRC generation is computationally

[6]Note of caution: Some optimistic techniques rely on hardware details that may not hold. E.g., when data is not aligned or larger than the MTU.



(a) Throughput comparison

(b) Relative slowdown of correct schemes

Fig. 14. Single-threaded performance of optimistic reads (1 compute node and 1 storage node).

expensive and (2) it is only probabilistic. Admittedly, the probability of a collision is low for CRC64. However, if a collision occurs, the data corruption is hard-to-detect.

FaRM Cache Line Versions. As with CRC, FaRM [11, 12] proposes a technique that requires one RDMA read in the best case. However, instead of computing a checksum, FaRM relies on coherent cache DMA (specified by x86) to detect if a single RDMA is consistent. FaRM stores a version at the beginning of *every* cache line as illustrated in Figure 12(a). Therefore, we can detect if a concurrent modification happens by comparing all cache line versions. To enable this, writers first latch the data item, read it, modify the data, increment the version locally, and then write the data back via RDMA in address order. Although not as computationally expensive as CRC, every cache line must be accessed to validate the versions introducing additional cache misses, i.e., stalled CPU cycles. In addition, this technique imposes additional storage overhead to accommodate a version in every cache line.

4.4 Single-Threaded Performance

We initially focus on single-threaded performance to understand the different tradeoffs of the correct schemes. Figure 14(a) compares the single-threaded read-only performance with varying tuple sizes. We also include the "broken" optimistic scheme and the optimized (and correct) pessimistic scheme from Section 3.

Optimistic vs. Pessimistic. Maybe unexpectedly, the pessimistic synchronization scheme performs better than all optimistic schemes, including the incorrect one. Although the broken optimistic scheme only requires two messages as opposed to the pessimistic scheme that requires three messages, the pessimistic scheme can exploit the asynchronous unlatch optimization. This optimization unlatches asynchronously, and the subsequent operation begins immediately (cf. Figure 7). Analogous behavior is not possible for optimistic schemes as the validation, i.e., the unlatch operation, determines if the operation failed or succeeded. Consequently, the validation must be synchronous (see Figure 13).

Correct Schemes vs. Broken. Let us now quantify the induced overhead of the correct schemes compared to the broken scheme, which only consists of a single RDMA request and no consistency checks to acquire the data and version information (i.e., the first round trip in Figure 12(b)). From Figure 14(a), we can observe that the broken scheme performs better than the correct schemes. The closest competitor is the FaRM scheme, which performs nearly as well as the broken scheme and only drops with larger message sizes. CRC can only compete with small tuple sizes, contrary to the versioning scheme, which catches up with large tuple sizes.

To highlight those effects, Figure 14(b) shows the slowdown in percentage compared to the broken scheme. We can see that both CRC and FaRM suffer from larger tuple sizes. The computational

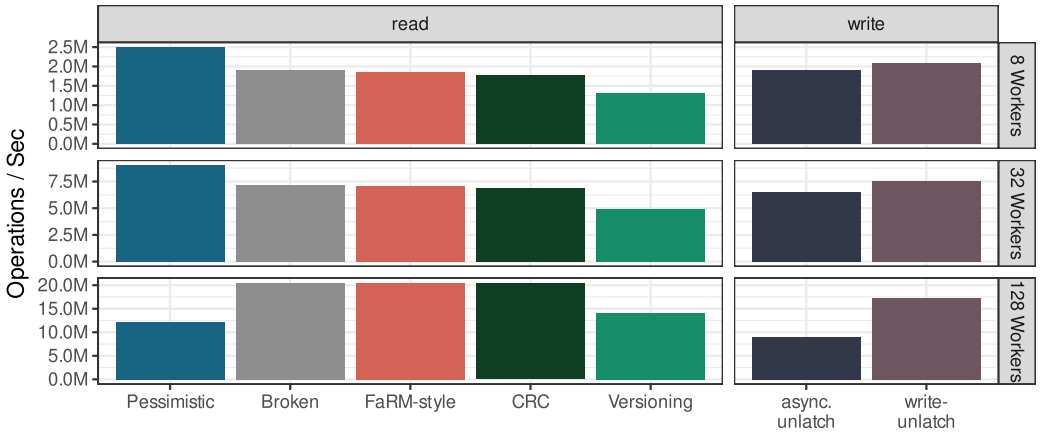


Fig. 15. Read-only and write-only optimistic scalability (4 compute nodes and 1 storage node).

overhead for both schemes is $O(n)$ since CRC calculates the checksum for every byte, and FaRM checks the versions in every cache line. Consequently, both schemes get linearly more expensive with increasing tuple sizes, more precisely, the consistency check (phase ②) as shown in Figure 13. However, because the CRC calculation is considerably more expensive than checking every cache line sequentially, the performance degrades more severely. For FaRM the sequential HW prefetcher and the out-of-order execution hide the cache misses. In contrast, the versioning approach incurs a substantial *constant* overhead by requiring an additional read for the version. Therefore, the versioning approach amortizes the initial cost with larger tuples sizes and performs better than FaRM.

To summarize, looking at the results of the single-threaded experiments, we found (1) the pessimistic schemes out-performs the optimistic schemes in the single-threaded setup, (2) FaRM performs better with smaller tuple sizes ($\leq$ 64 KB), (3) while versioning is beneficial with larger tuples.

4.5 Scalability of Optimistic Techniques

While the optimistically latched single-threaded performance was worse than the pessimistic scheme, it is not indicative of its scalability behavior. After all, the main benefit of optimistic schemes is that they avoid RDMA atomic operations during a read and, thus, avoid physical contention on the RNIC generated by multiple workers.

Read-Only Scalability. Figure 15 shows the scalability w.r.t. the number of workers for small, i.e., 256-byte sized, tuples. Again with fewer workers (8), the pessimistic approach performs better than the optimistic schemes. In line with the previous results, CRC and FaRM are close to the broken scheme, whereas the versioning scheme is far behind. However, under the highest contention when using 128 workers, all the optimistic schemes perform better than the pessimistic scheme. In particular, FaRM and CRC perform almost twice as well as pessimistic synchronization. This confirms our intuition that by avoiding RDMA atomics, the physical contention effect does not limit the performance in optimistic schemes.

We also compared the scalability of larger, 16 KB sized tuples and found that the pessimistic scheme performs better in this scenario. The reason is that with larger tuples, the workload becomes network bound before the RNIC contention limits the performance. In other words, only 8 workers are required to reach the full bandwidth (12 GB/s) with the pessimistic synchronization, and the RNIC can easily sustain RDMA atomic operations at such a rate.

Write-Only Scalability. Remember that only reads are performed optimistically, while modifications are still latched pessimistically. However, since the optimistic readers do not use RDMA

atomics, we can now leverage the write-unlatch optimization, which uses a write instead of an atomic operation to unlatch (see Figure 7). To remind ourselves, we have not considered them before because the write-unlatch optimization does not work with concurrent FAA operations required to implement reader-writer latches. But because optimistic schemes only require one latch mode, i.e., exclusively latched or unlatched, write-unlatch can be used to further reduce the number of RDMA atomic operations.

Figure 15 (right side) shows the effect of this optimization when increasing the number of workers up to 128. We can observe that the write-unlatch optimization enables better scalability by avoiding one RDMA atomic operation. When comparing the performance of the write-unlatch optimization with 128 worker and the results from Figure 9 we can see that it approaches the performance of the unsynchronized variant. Important to note is that the layout as shown in Figure 12(a) must be reversed so that the version and latch are placed at the end of the data to exploit address-ordered RDMA writes. That is, the last byte written must unlatch the record. The reversed layout does not affect the performance of the optimistic approaches since it merely changes the location of the version.

So far, we have focused on performance numbers. However, the correct techniques come with tradeoffs, such as the number of required messages. In the best case, the versioning approach requires three messages and can be a sub-optimal strategy when the workload is message-bound. In contrast, CRC requires only two messages but is computationally more expensive, which may be detrimental if the threads can do other valuable work. Moreover, CRC is probabilistic, and although the collision probability is low, the question remains: why risk data corruption when reliable schemes exist? Lastly, although FaRM is certainly efficient, it adds $2 - 8$ bytes storage overhead [11] per cache line, which means it is not only $O(n)$ in terms of computing but also storage overhead. In addition, the higher-level logic must handle the interleaved cache line versions. For instance, any string operation (comparison, regexp, and the like), e.g., on a value in a B-Tree, must explicitly deal with strings chunked across cache lines. Because only a few existing libraries support chunking, one would have to copy larger strings into a contiguous memory before being able to use them and potentially offset the gains by the efficient scheme. Consequently, choosing the correct scheme has tradeoffs that must be carefully weighed against each other and co-designed with the system.

Summary. To conclude, we have found that the optimistic techniques scale and perform better than pessimistic techniques for workloads in which the RNIC's capability of handling RDMA atomic is the limiting factor. For instance, this happens in a read-only workload with small tuples and 128 or more workers. When using larger tuples, the workload tends to become network bound, reducing the number of possible parallel operations and shifting the bottleneck for pessimistic schemes from the RNIC contention to the network. Thus, a pessimistic scheme often performs better in such scenarios. Another interesting finding is that in combination with optimistic reads, we can improve the scalability of writes by using the write-unlatch optimization. In contrast to pessimistic schemes, optimistic schemes come with different tradeoffs regarding computational and storage requirements.

4.6 Effect on a Disaggregated DBMS

Finally, we compare the optimistic schemes by implementing them in NAM-DB and call this system OPT-DB. Similar to the experiment in Section 3.5, we vary the contention and show write-only, mixed, and read-only workloads. The configuration is unchanged with four compute and storage nodes storing $20M$ tuples. Unlike the experiment in Section 3.5, we stick to 512 bytes but vary the number of workers from 112 to 224.

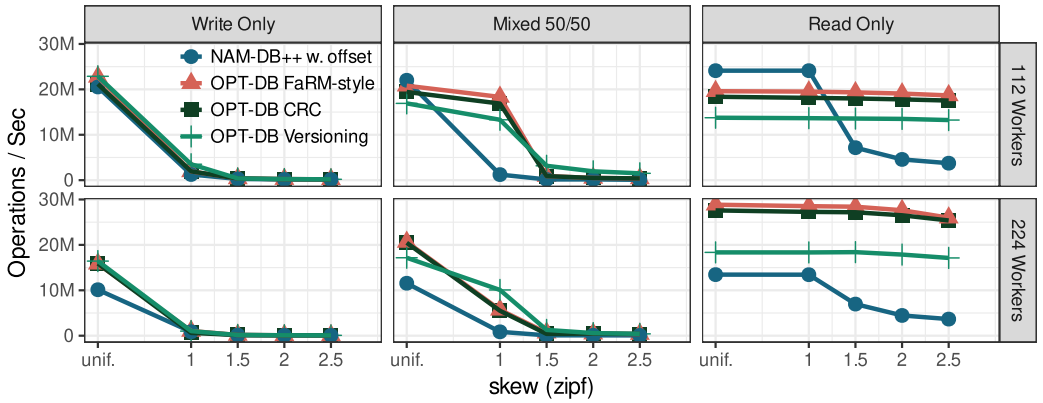


Fig. 16. Optimistic techniques in a disaggregated DBMS (4 compute nodes and 4 storage nodes).

Contended Shared Latches. In the read-only workload in Figure 16, we can observe that all optimistic techniques are pretty robust against contention, contrary to the pessimistic synchronization in NAM-DB++, which already degrades with light skew (1.5). Furthermore, we can back up the previous findings that the pessimistic scheme performs better with fewer workers than the optimistic schemes. The optimistic approaches perform better when increasing the number of workers to 224. Overall, the OPT-DB implemented with FaRM performs the best, followed by CRC and Versioning (for 512 byte tuples).

Reader/Writer Contention and Aborts. There is a well-known tradeoff for optimistic approaches in reader-writer contention: optimistic approaches typically lead to more restarts, resulting in wasted work. Nevertheless, the mixed workload shows that the optimistic approaches out-perform NAM-DB++ despite the restarts.

The reason lies again in the fewer RDMA atomic operations. Especially under contention (skew), the atomics fall in the same lock slot, which creates physical contention in the RNIC. Moreover, in this experiment, the logical contention exacerbates this physical effect. Once the RDMA atomic operation is executed, it does not necessarily mean that the lock operation was successful, e.g., it may be already latched exclusively by another worker. Thus, for a pessimistic scheme, the operations may restart, which incurs additional atomic operations. To this end, for the mixed workload, the optimistic approaches, which get away with a single atomic operation for exclusive latches and none for optimistic (read) acquisitions, perform much better.

The same holds for the write-only benchmark; since the optimistic approaches harmonize with the write-unlatch optimization, they perform on par or sometimes even better (e.g., with low contention and 224 workers as shown in Figure 16). Moreover, in the write-only workload with 112 workers, the versioning approach is slightly faster since the writers do not need to perform any additional computation, such as CRC checks or incrementing the cache line versions. When contention increases, all the approaches converge at merely 100K operations per second.

5 Will Hardware Advances Solve Performance and Correctness Issues?

Since the publication of our previous study [72] and the results presented in Sections 3 and 4, which utilized 100G NICs, more than a year has passed. During this period, the landscape of **Network Interface Cards (NICs)** has significantly evolved, particularly with the introduction of faster RDMA NICs, which support 400G. These networks are now well-established in the server market, with even faster interconnects like 800G and 1.6T on the horizon [32]. To accommodate these higher

bandwidths, NICs must be more powerful, as evidenced by the fact that with 400G NICs, new packets can arrive approximately every 1.67 nanoseconds [32] at a server. Given these technological advances, one question remains: Do modern NICs resolve the scalability issues of RDMA atomic operations? Considering the improved performance of modern NICs, it is reasonable to hypothesize that such hardware advancements could alleviate the scalability challenges, allowing systems engineers to benefit directly from these improvements without resorting to workarounds like padding (cf. Section 3.2). In this section, we thus dive deeper into this question and, unfortunately, see that the answer is a clear "no". As such, we additionally visit other alternatives to provide scalable and correct synchronization methods.

5.1 400G NICs to the Rescue? Revisiting RDMA Atomics

To address the previously raised question, we now evaluate the performance of RDMA atomics on the latest 400G NICs of the ConnectX-7 generation. Given that the new NIC generation [43] supports network speeds up to four times higher than its predecessor (cf. Section 3.2), we anticipate significant scalability improvements, particularly in uncontended atomic operations. However, we are also interested in seeing whether the NIC-internal implementation of atomics changed and thus supported better scalability for contended workloads. To test these questions, we used the following setup.

Testbed. Our experiment uses a 400G testbed comprising two single-socket nodes, each powered by an AMD EPYC 9554P processor with 64 cores clocked at up to 3.75GHz and equipped with 768GiB of RAM. The nodes run Ubuntu 22.04.4 LTS on a Linux 5.15.0 kernel. Each node includes two Nvidia ConnectX-7 MT2910 RDMA NICs connected via PCIe-5, enabling a topology with four endpoints that we use for our experiments. The scope of RDMA atomic operations is limited to a single NIC, meaning that accessing the same memory region through multiple NICs does not ensure coherence. Therefore, we ensure that only a single NIC handles atomic operations to certain memory regions. We connect these four NICs using an Intel Tofino2 400G Ethernet Switch, configured as a traditional router for RoCE traffic. It is important to note that RDMA frames under RoCE are slightly larger than those under Infiniband, which may cause minor variations in latency and throughput. Despite this, RoCE and Infiniband share the same RDMA headers at the transport layer. RoCE is particularly interesting for data center applications as it integrates seamlessly with existing Ethernet-based infrastructure.

Results. With the updated hardware configuration, we executed the scalability experiment, as shown in Figure 5. With only two physical compute nodes, our testbed achieved a lower peak throughput than the original setup. Nevertheless, the performance trends observed in the new setup (cf. Figure 17) remain consistent with our previous findings. For instance, the maximum throughput for physically contended atomic operations is again capped at approximately 2.5 million operations per second. Similar to the initial experiment depicted in Figure 5, we observed that introducing 8-byte padding addresses the scalability issue.

The Problem Persists. Disappointingly, the answer to whether 400G NICs improve the scalability of RDMA atomics is a "no". Despite achieving higher throughput for standard RDMA operations, RDMA atomics are still serialized using the same internal lock table as previous NIC generations. Consequently, developers must create synchronization schemes that avoid physical or false contention by carefully addressing alignment issues.

We speculate that resolving this issue is feasible, but there appears to be insufficient industry interest. For example, Nvidia, the vendor of the ConnectX-NICs, primarily focuses on machine learning workloads for training and inference, which do not require such fine-grained locking. Potential approaches to mitigating this limitation could include hashing addresses to distribute contention more evenly or increasing the size of the lock table.

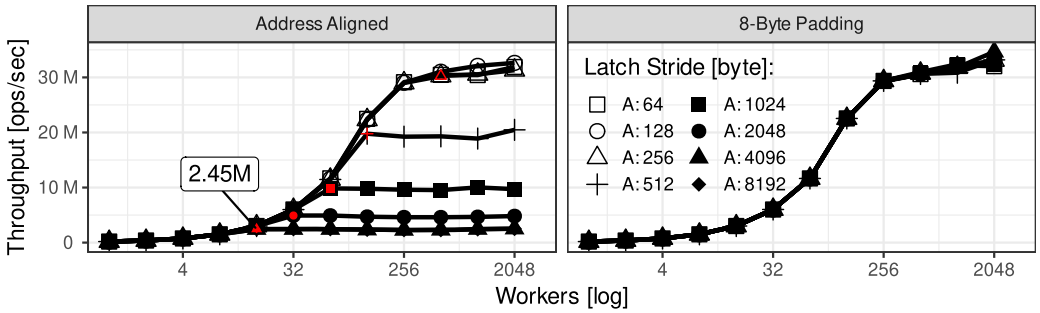


Fig. 17. Scalability of uncontended atomic RDMA operations on 400G ConnectX-7 NICs. The size of the internal lock table appears not to have changed, and the maximum throughput for contented locks is as high as that of 100G ConnectX-5 NICs. Due to a smaller testbed, the maximum throughput differs from Figure 5.

However, since using padding to correct alignment problems is merely a temporary fix, exploring alternative methods for scaling database workloads is crucial. Fortunately, recent advancements in network infrastructure go beyond mere performance improvements on NICs. An increasing number of networking components have become programmable, offering new opportunities for exploration. In the subsequent sections, we will investigate how these programmable components can be utilized to overcome the limitations of atomic RDMA implementations.

5.2 The Opportunity for Smart Networking Hardware

As network bandwidth increases, networking hardware is also becoming more programmable, driven primarily by the need for flexible deployment in data centers [4, 28, 45]. The introduction of programmability at both the NIC level, through SmartNICs, and the switch level, via programmable switches, marks a shift towards utilizing the network as an active computational resource. In the remainder of this section, we will explore how programmable network components can transform one-sided synchronization schemes and examine the benefits and limitations of current generations of smart network devices.

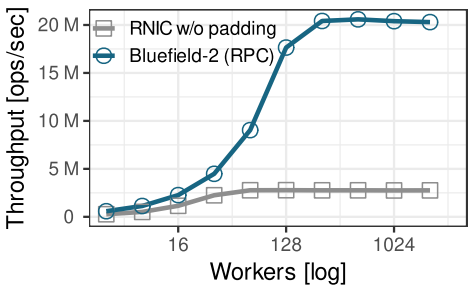


Fig. 18. CAS without padding on ConnectX-7 RNICs vs. Bluefield-2 SmartNIC, which uses a two-sided RPC approach and CPU atomics on the Bluefield's ARM-cores.

SmartNICs. **Smart Network Interface Cards (SmartNICs)** are central to the ongoing transition toward programmable networking hardware. These devices are engineered to offload processing tasks from the CPU to the network card, bringing computation closer to the data flow. Offloading reduces CPU load by directly managing tasks such as packet filtering, encryption, and traffic management on the NIC. By processing data en route, SmartNICs effectively decrease latency and free CPU resources for other operations [15, 34].

SmartNICs theoretically simplify the expansion of networking primitives. For example, instead of issuing a series of one-sided RDMA operations, one could implement a custom operation such as `latch_tuple` directly on the SmartNIC. Although one-sided operations are not inherently faster on devices like the Nvidia BlueField-2, the ARM cores on these NICs can be used to implement an RPC-style latching protocol that bypasses the internal locking table on the NIC.

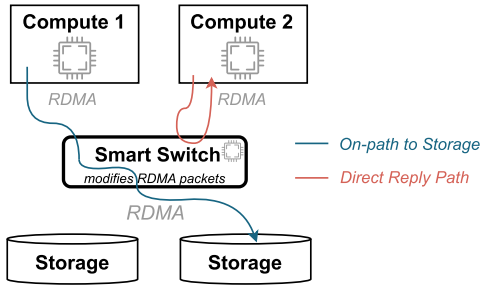


Fig. 19. The programmable switch sits on the data-path between compute and storage. The storage nodes hold all tuples which are accessed via RDMA. It can modify RDMA packets on-the-fly and optionally reply directly back to a compute node in 1/2 RTT.

The BlueField SmartNIC operates like a separate host, equipped with its own memory and ARM cores. While it is not directly on the data path, it is addressable from the network via its own IP address. This allows us to store latches in the device memory, with the option to place the tuples either in the device or in the host memory. The primary advantage of this approach is that it bypasses the traditional locking table required for one-sided atomic operations, instead relying on the memory attached to the ARM cores. In other words, the ARM cores handle the incoming latch request by issuing atomic CPU operations on the latch and responding with the result.

To see how such a synchronization scheme performs under high contention scenarios, we implemented an RPC-style protocol on a Bluefield-2 SmartNIC and compared it with atomics on conventional NICs. The performance of both schemes is shown in Figure 18. Here, the SmartNIC approach (blue) demonstrates better performance of up to 20M ops/sec compared to the one-sided synchronization results, which is capped at around 2.51M ops/sec due to unpadded atomics that map to the same physical slot in the locking table on a conventional RDMA NIC (cf. Section 3.2).

Given these findings, SmartNICs and synchronization schemes are promising for the future since they can provide scalable execution by offloading them to the SmartNIC. In the remainder of this article, we look at a second alternative: programmable networking hardware. In particular, we concentrate on programmable switches, which provide interesting properties. As we will show, these switches offer up to four times higher throughput for contended workloads and have the potential to implement lower latency latching schemes, which are crucial for many latency-sensitive database workloads.

Why Programmable Switches? Network switches, the backbone of networking infrastructure, have evolved to become programmable, too. Traditionally, introducing new networking protocols typically required the costly and inefficient process of replacing fixed-function switches. Programmable switches, however, can be reconfigured to support new protocols or to implement custom user logic at the data-plane level. Unlike SmartNICs, switches can process data at line rate, handling billions of packets per second—a capability that stems from their ability to manage the aggregated bandwidth of all connected nodes.

In the context of synchronization schemes, programmable switches exhibit three notable characteristics: (1) Latches located on the switch can be accessed in half the round-trip time compared to those on a storage node, as shown in Figure 20; (2) Switches consistently process at line rate, ensuring there is no performance degradation between contended and uncontended workloads, which we will demonstrate in the following sections; (3) As depicted in Figure 19, switches have a comprehensive view on all traffic flowing between compute and storage nodes.

Given these advantages, we propose that switches could effectively serve as a high-performance latch service. Figure 19 demonstrates how a smart switch can be utilized in a disaggregated architecture. Notably, the switch can directly respond to requests, such as latch requests, enabling very low latency. Meanwhile, other requests, like reading data from storage, are simply routed through the switch. An important feature to highlight is the smart switch's programming model (compiler), which ensures that any software running on the switch operates at line rate—even contended operations, as we will show later.

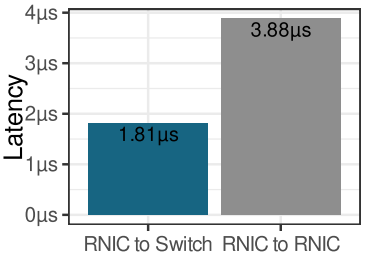


Fig. 20. Latency of 8 Byte RDMA operations to a programmable switch vs. remote storage node.

At the same time, this strict requirement that all operations on the switch must run at line rate significantly constrains the programming model and limits the switch's resources. In the subsequent sections, we will investigate how programmable switches are programmed and examine their capabilities in addressing scalability bottlenecks and correctness issues.

5.3 Background: Architecture of Programmable Switches

In this section, we explore the hardware architecture of programmable switches in more detail, which explains why they are interesting alternatives for implementing synchronization schemes. The background provided next is also crucial for designing and understanding new algorithms based on this unique architecture.

Overview. Programmable switches have two components: the control plane and the data plane. The control plane, which typically runs on the network device's (relatively slow) CPU, makes routing decisions based on the network's topology and policies. Conversely, the data plane is tasked with the actual packet processing, which involves efficiently routing traffic to its intended destinations. The data plane's programmability is achieved through a reconfigurable architecture using match-action tables, also known as the **Protocol Independent Switch Architecture (PISA)**. Match-action tables can be configured to execute user-defined actions on a packet whenever their key matches certain fields in the packet. These match-action tables are allocated in a fixed layout, forming a pipeline where packets can flow through [22].

Packet Processing Pipeline. The packet processing pipeline is the core of the data plane and consists of three components: The packet parser, the pipeline of match-action stages, and the deparser. All three components are realized within an ASIC but are reconfigurable and programmable, so their behavior can depend on specific fields in the packet headers.

Figure 21 shows the high-level architecture of the switch and how packets are processed from left to right. Once a network packet arrives as a byte stream, the parser takes this stream from the wire and instantiates different header instances based on the contents of a packet (cf. Figure 21). The parsed packet then moves into the first stage of the match-action pipeline.

The **match action unit stages (MAUs)** do the actual processing of the packets. Multiple MAU stages are chained together in a pipeline to allow for more complex processing (cf. Figure 21). In other words, a packet can flow through the match action unit stages. A MAU stage consists of multiple tables with associated actions. Such a table contains rows, each consisting of table keys and an action. Whenever a table key matches fields inside a packet header, the corresponding user-defined action can be executed on this packet. Therefore, a packet could match multiple tables in a single stage when there are no data dependencies between the tables. This can, for example, be setting the output port (the action) for a packet by matching on its destination IP address (which corresponds to a table key). Once the packet traverses all

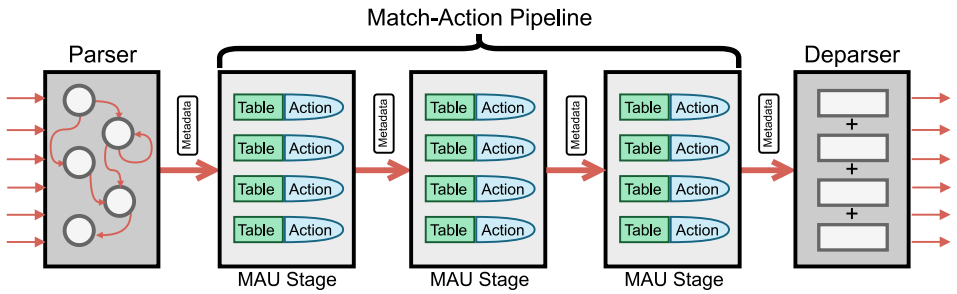


Fig. 21. The Protocol Independent Switch Architecture (PISA) enables switches to be programmable. Packets move sequentially through the pipeline and can be modified according to user-defined match-action rules in the MAU stages.

MAU stages and is modified according to the user-provided program (actions), it goes to the deparser.

The deparser resembles the packet headers back to a byte stream so that it can be sent out to the wire (cf. Figure 21). Throughout the whole pipeline, each packet may carry additional metadata that is not part of the packet itself but can store important information necessary for its processing.

Programming Model. The programming model of switches allows us to customize all three previously presented components: Parser, Match-Action Pipeline, and Deparser.

The main focus lies on the processing of packets, specifically allowing for the specification of how headers within a packet should be modified or rewritten, e.g., routing a packet to a specific port based on an IP address.

The P4 language has emerged as the de facto standard for programming the data plane in network devices, enabling the definition of specific packet processing behaviors through high-level match-action rules [3, 5]. Even though P4 was initially developed for programmable switches, it is also being used across various systems like SmartNICs and FPGAs for packet processing tasks [57].

Conceptually, P4's syntax is similar to C; however, it does not allow pointers and has many other restrictions regarding floating-point numbers, loops, or random memory accesses. There are well-known techniques like loop-unrolling, fixed-point arithmetic, or dictionary encoding for strings to work around these constraints. Also, one must remember that each packet can access only the resources in the stage it is currently in. This processing model is very different from what we have in CPUs, where an arbitrary order of many random accesses to memory is possible. This means switch programs must be designed differently from traditional CPU-based programs. Most of these architectural decisions come from strict timing constraints because packets need to be processed at line rate. Often, switch vendors also add different dedicated hardware-accelerators (e.g.: a checksum-engine or FPU) to the ASIC, which are then available through so-called P4-externs. Each P4 program that compiles for the switch can be run at line rate speed. Otherwise, it is rejected by the compiler [65].

TNA: Tofino Native Architecture. While PISA provides an abstract model for switch architecture, it allows vendors to introduce specific implementations, such as additional parsers for tunnel processing, specialized components for checksum computation, or stateful operations within the MAUs. The most important feature in the Tofino architecture is stateful operations because they allow a packet to access data stored by other packets in the processing pipeline at line rate. This is realized by register arrays that are located in each MAU stage. Register arrays can be accessed using a runtime index, allowing so-called combined operations to be executed in a single clock cycle through the internal ALU. These operations are similar to a combined CAS on a CPU and

are perceived as atomic modifications by other packets. Registers pose a major advantage to tables because their content can be modified directly by packets. In contrast, tables can only be modified through the control plane at a significantly lower rate. With that, packets do not rely solely on table lookups for their control flow but can also utilize the information from registers that previous packets have modified [22].

Guarantees. Switch programs have strict properties that are dictated by the packet pipeline and validated by the P4 compiler. The packet moves into the next adjacent pipeline stage at each clock cycle. During its stay in one of the MAU stages, the packet is confined to accessing only its local resources, e.g., tables or registers. Since, at most, one packet resides in a stage and packets cannot take over each other, memory (register array) accesses are consistent and well-defined. This means a packet sees all modifications its predecessor made to the same stage in the previous cycle. Thanks to these properties, switch programs do not require synchronization within the switch's data plane because it is already given inherently by the pipeline architecture.

5.4 A First Microbenchmark: Offloading RDMA Atomics to the Switch

With the programming model and the architecture in mind, we now want to explore the capabilities of programmable switches. As the first microbenchmark, we initially tested executing RDMA atomics directly in the switch's data plane, utilizing the switch's memory. The primary strategy involves configuring the switch as the target for one-sided atomic RDMA operations, thereby eliminating the need to forward requests to the storage node, as illustrated in Figure 19 (direct reply path). By implementing this design, the switch can effectively serve as a latch service while the data remains stored on the storage nodes. This approach leverages the programmable nature of the switch to address scalability and performance challenges previously observed with traditional RDMA atomics.

Implementation. For this use case, register arrays of the Tofino switch architecture are particularly well suited because they allow us to store state, e.g., the value of the atomic variable, which is persistent between packets. We added new Infiniband headers to the P4 firmware and extended the switch's control plane to simulate an RDMA endpoint and allow connections to it. Now, the switch can interpret RoCE packets from the RDMA NICs, where Infiniband headers are embedded inside traditional UDP packets.

After connection setup and exchange of QP information, the control plane installs the necessary match-action rules into the data plane such that the firmware can react to RDMA packets. Whenever the RDMA packet type of an RDMA-Atomic or RDMA-Read request arrives at the switch, the data plane uses the virtual address to get an index for a slot in the register array. The register is then accessed accordingly, and its result is written into the packet. The packet is then converted into an RDMA acknowledgment packet and sent back to the client. Such logic can be realized in just a few MAU stages.

Results: Contended and Uncontended Atomics Scale. We now benchmark the switch accelerated atomics using four nodes that are connected via 100G links to our Tofino1 switch. We replicate the setup in Section 3.1, although, with a different testbed (cf. Section 5.1). Each worker thread issues an RDMA operation and waits synchronously for its completion. Note, that multiple outstanding requests would be also supported, but latching is often synchronous. The results are shown in Figure 22 for up to 2,048 concurrent workers.

Compared to the results in Figure 4, interestingly, the switch accelerated atomics does not suffer from contention. Contention to specific addresses does not affect the switch pipeline because a packet is the only entity that can access the resources of its local MAU stage, and no further synchronization is necessary. Therefore the throughput of contended and uncontended operations in the switch's data-plane is exactly the same. This perfect scalability is also amplified by the

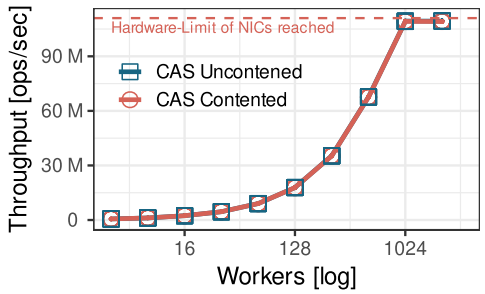


Fig. 22. Scalablity of atomics on a programmable switch. Due to its architecture, the switch does not require synchronization through a lock-table.

reduced round-trip time to the switch, also under significant load, which is less than half to that of a normal RDMA NIC. The throughput for four clients maxes out at around 110M operations per second, which is twice the amount of a NIC-to-NIC setup. However, the switch is barely utilized and uses only a fraction of its available bandwidth because it only needs to process the load of 4 connected ports.

In conclusion, an architecture such as the one provided by Tofino is very well suited for high-rate processing of RDMA operations that are resistant to contention and have predictable latencies.

5.5 Pushing it Further: Switch-assisted Optimistic Latching

In our previous experiment, we observed that normal RDMA atomics scale effectively on programmable switches and can provide a building block for pessimistic synchronization. Another article [66] explored a similar direction by implementing a pessimistic locking service on the switch. In this section, we now show how a full optimistic synchronization scheme can be implemented on a switch and analyze it in terms of correctness and performance. As discussed, optimistic synchronization schemes using one-sided RDMA suffer from various correctness problems (remember the "torn read" experiment in Figure 1(b)). In the following, we discuss using a switch to provide a scalable and correct optimistic synchronization scheme in a disaggregated setup. In such a setup, the switch will see all reads, writes, and modifications of the tuples by compute nodes on remote nodes via RDMA and can thus implement an optimistic concurrency scheme.

As discussed in Section 4, traditional RDMA reads suffer from weak ordering guarantees as the main problem and prevent optimizations such as reading the version together with the data. The main idea of using a switch is to offload the tasks of verifying whether an RDMA read was consistent to the switch and let it determine if the data was modified in-between reads by a concurrent write (cf. Figure 23). To do that, the switch stores a version map for each tuple in its internal memory, observes the RDMA traffic that flows through its data plane and uses it to coordinate reads and writes using an optimistic latching scheme, as we explain next.

Switch-Enhanced Writes. Let us assume a node wants to update the content of a tuple. In the optimistic latching scheme as described before in Section 4, an exclusive latch on the version needs to be obtained to update the version (and set it to an odd number) before a client can then go ahead and modify the tuple contents. Since the switch is in the middle, it can reply to this latch request directly without forwarding the packet to the storage node in less than half the round-trip time. The unlatch operation, which only increases the version back to an even number, is processed similarly. In our switch-enhanced write protocol, the switch intercepts this request, updates its local state (i.e., version map) to reflect the change, and replies directly to the client. The RDMA write of the tuple data is passed through to the storage node because a switch does not hold enough memory

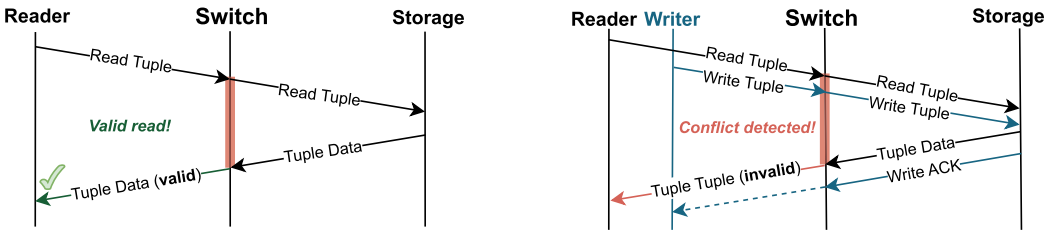


Fig. 23. The programmable switch sits in front of the storage node which hold the data. The switch can thus detect inconsistent tuple reads by observing the packet order in its pipeline. On the right, a tuple write happens in the critical red area, and the switch marks the ongoing read tuple data as invalid.

to store tuple contents besides the versions. On the network level, each RDMA Write Request generates an acknowledgment packet on the remote side. By combining this with the unlatch phase, the switch can automatically unlatch the tuple again internally whenever the RDMA write is completed. This way, the client does not need to issue a separate unlatch operation. In summary, latching a tuple requires half round-trip time and writing a tuple with unlatch requires a single round-trip time.

Switch-Enhanced Reads. The switch can determine whenever a tuple is latched by checking if the version of a tuple is an odd number. For each read-request to a tuple, the switch can also use its internal state to check whether the read was consistent or not. It stores the current tuple version in a register slot, which is private to each client whenever they read a tuple. As soon as the node replies with the data, the switch re-reads the current version of the tuple and compares it with the previously stored version. The result of this comparison is embedded into the header of the tuple. With this information, the client can verify whether the RDMA read to this tuple was consistent and re-issue the operation if necessary. To verify whether a tuple was modified before committing, the client can issue a short RDMA read to the tuple version stored in the switch's memory. When executed by the switch, like the write latch operation discussed before, this short version read operation takes only half a round-trip time. With this scheme, the switch can detect possible concurrent writes we would encounter when optimistically reading remote tuples using RDMA while reducing the number of necessary round-trips and latency.

Thinking like a Switch. Before implementing our design as a switch firmware, we need to go deeper into the hardware level and map our problem to the switch architecture. Switches operate on packet level only, and to implement accelerated reading and writing of versioned tuples, the switch must distinguish between 6 different packet types (cf. Table 2). The six packet types map to the logical *Latch Tuple*, *Unlatch Tuple*, *Read Tuple,* and *Read Version* operations. Matching on the RDMA opcode, the packet source, and optionally on the DMA length is sufficient to infer the tuple operation. The switch needs to do four lookups to process all six packet types.

The most essential lookup is done to obtain a tuple-id's current version using a register. The tuple-id is determined by using the virtual address. With that, the switch can reply to *Latch Tuple* and *Read Version* packets. In RDMA, both QPs on the sender and target node have **packet sequence numbers (PSN)** that are required to ensure a reliable data transfer. However, replying directly to the sender without forwarding a packet to the target node increments the PSN counter on the send side but not on the target side. Without adjusting the PSNs of packets of the same QP that are afterwards forwarded to the target, sequence number errors would be generated. Therefore, the switch uses another table to account for this offset in PSNs whenever it replies to a packet, forwards it, or receives a response from the storage node (cf. Table 2). *Read Tuple* and *Write Tuple* packets both require such adaption of the PSN. In RDMA, reads and writes contain a field in the

Table 2. The Switch Processes Each RDMA Packet Differently

			Tables for Switch pipelines			
Packet-Type	**Source**	**Target**	**Client to Tuple-ID**	**Tuple to Version**	**Client to Version**	**PSN Adjustment**
Read Tuple Req.	Client	Storage	Store Tuple-ID	Read Tuple-Version	Store version of Tuple	Add. PSN offset
Read Tuple Res.	Storage	Client	Read Tuple-ID	Read Tuple-Version	Verify stored version	Sub. PSN offset
Read Version	Client	Client	-	Read Version	-	Incr. PSN offset
Latch Tuple	Client	Client	-	Try-latch Tuple	-	Incr. PSN offset
Write Tuple Req.	Client	Storage	Store Tuple-ID	-	-	Add. PSN offset
Write Tuple Res.	Storage	Client	Read Tuple-ID	Unlatch Tuple	-	Sub. PSN offset

The tuple-id is stored for each request because it is unavailable in responses from storage nodes. Highlighted rows only require 1/2 round-trip.

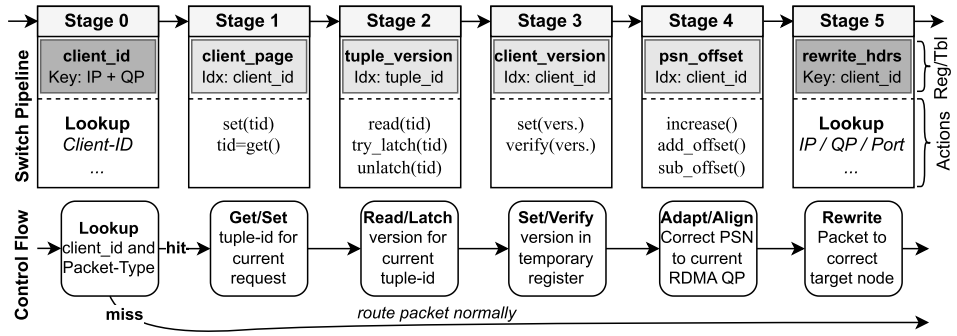


Fig. 24. Control flow and mapping to a switch pipeline to implement switch-assisted optimistic latching.

header that holds the target virtual address. Reads receive the read data in their reply packet, and writes generate a simple acknowledgment of the target side. The switch uses this virtual address to infer the tuple-id. However, neither the read nor write replies contain the virtual address, so the switch cannot infer this information.

To work around that, the switch needs to store the current tuple-id for each request so that it is available for the response in the *Client to Tuple-ID* table. Whenever the target acknowledges an RDMA write with new tuple content, the switch can directly unlatch the tuple using the previously stored tuple-id. Lastly, to verify the consistency of RDMA reads, the switch needs to compare the tuple version at the time the read request came in with the tuple version when the read response comes back from the storage node. The switch uses again the *Client to Tuple-ID* table to store the current tuple-id, but also a version number in a new *Client to Version* table. The client-id also indexes this table, and the verification result is embedded into the first bytes of the read tuple response. The client can then verify whether the RDMA read to the tuple was consistent or not.

Implementation. After determining the tables necessary for the switch logic, implementing the switch firmware is straightforward. Besides the logic to handle the operation on tuples and their versions, the switch requires additional logic to properly direct packets to their destination. This can be done using an additional table that maps the packet type and the client-id to mac-address, IP address, Infiniband-QP, and the like. As we have seen in Section 5.3, the order of accesses a packet can make to tables and registers is fixed. Therefore, we first need to map our control-flow from the previous section to the stages of the physical switch pipeline. The physical layout and the mapping to the execution pipeline of the switch is shown in Figure 24.

First, the switch determines the client using the destination IP and QP. Both request and response map to the same client id and the lookup can be done using a static table in stage 0. After that, a register-array follows, producing a mapping of client-id to tuple-id in stage 1. For requests, the current tuple-id is stored in the client's slot and for responses this value is read to get the current

tuple-id. With the known tuple-id, the switch can read or modify the tuple's version that is stored in another register-array. This register-array is indexed using the tuple-id in stage 2. To verify tuple reads, the tuple version needs to be temporarily stored on the switch until the response arrives. This happens in the *client_version* register in stage 3. The offset for packet sequence numbers is stored in another register array that is indexed by the current client-id. This register array has no logical dependencies on other steps; however, we opted to place it in stage 4. Finally, the packet needs to be routed to the correct destination with the correct packet fields. The required values for this step are retrieved from a table stored in stage 5.

Verification and Correctness. The protocol itself and the switch pipeline guarantee the correctness of our approach for switch-enhanced optimistic latches. First, the switch sits in the middle and sees all operations on tuples; therefore, the switch's pipeline serializes all modifications to the version field. Second, the tuple version is only modified after the write is completed; there is no in-between state because versioning and data are separated (cf. Figure 23).

We also empirically verified the correctness of our implementation using the torn read experiment from Figure 1(b). In this experiment, writers update the tuples by setting all the data to a specific counter value whenever they acquire a latch. Readers then read the tuples and check if the tuple contents are consistent by checking if all values are the same in the tuple-data. With our implementation, we could not detect any torn reads as in the original experiment without the switch. We use reliable **RDMA connections (RC)** where lost packets in RDMA are retransmitted after a timeout.

Another open problem is that a switch may potentially execute an operation multiple times due to duplicated packets issued by the NIC as part of the networking protocol. We prevent the execution of the same logical operation (e.g., a latch) multiple times by detecting duplicate packets using an additional register that stores the last processed PSN. If the switch crashes, the storage nodes can completely restore the versioning information. The switch state is recovered by reading the tuple-versions and installing them into the switch register. Whenever a write operation is issued, the switch embeds the new version on-the-fly directly into the tuple-header such that it is written with the data to the storage node.

Data-Center Deployments. While multi-switch deployments require additional configuration, they also unlock new opportunities. The proposed single-switch design can be extended to multi-switch topologies, such as fat-tree architectures commonly found in data centers. To ensure protocol correctness, all accesses to relevant tuples must be visible to a dedicated switch so that it always sees the most recent tuple versions. This can be achieved through fixed path routing or by deploying the program on the top-of-rack switch, which typically sits above the database node storing the tuples. Additionally, multiple switches increase the storage capacity for version data, increasing the number of tuple accesses that can be accelerated by programmable switches. The recovery mechanism discussed earlier can be used to support failover to another switch within the topology, enhancing fault tolerance.

Latency Breakdown. To gain insight from where the performance benefits offered by our switch-enhanced latching stem from, we conducted a microbenchmark contrasting our new method with a baseline that employs correct optimistic synchronization strategies. We measured latency under real conditions using 1KiB tuples to avoid packet fragmentation. The results include both a read-only and a write-only workload, as illustrated on the left and right sides of Figure 25, respectively. Notably, the read tuple operation using the switch approach proved faster, primarily due to the decreased load on the storage node. This shift effectively redistributes load from RDMA NICs of the storage node to the programmable switch. Consequently, latencies varied for different operations due to this load shift. The error margin for the measured latency in the results is negligible at approximately 0.1 microseconds. For both reading and writing, the switch approach achieves

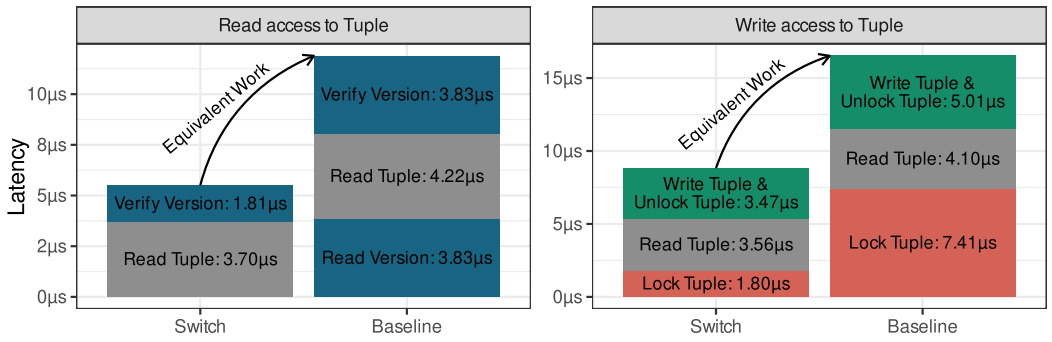


Fig. 25. Latency breakdown of different steps required for read and write accesses to a tuple. (1 compute and 1 storage node).

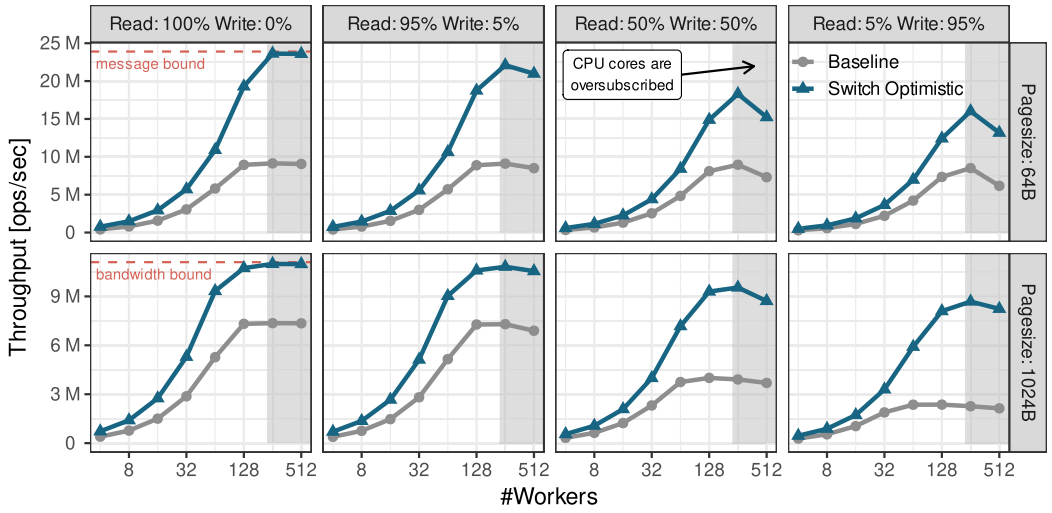


Fig. 26. Scalability of switch accelerated and corrected optimistic latches. (4 compute nodes, 1 storage node).

roughly double the performance of the baseline. This aligns with the theoretical round-trip times determined in Table 2.

Scalability. As a second experiment, we conducted a scalability experiment to assess how the system behaves under varying workload characteristics. For that, we use a setup comprising four compute nodes and one storage node, scaling up to 512 workers. We use a key-value store workload with uniform read-write accesses to 140,000 tuples and store the versioning information on the switch. This setup does not involve transactions but includes latching for writes and verification of reads to ensure consistency. Both the switch-enhanced and the baseline latching mechanisms use the same workload code.

The experiment was conducted with four different read-write ratios to explore the impact of different mixes between read and writes. Additionally, we varied the tuple sizes between 64B and 1024B to examine the effects of data granularity on performance. The single storage node's bandwidth and message rate were inherently bound by its 100G connection, which became a limiting factor, especially in read-only workloads, as illustrated in Figure 26. Oversubscribing CPU cores, as highlighted in the grey-marked areas of the results, leads especially for write-heavy workloads

to poor scaling and a notable drop in throughput due to scheduling effects. Overall, the results demonstrate that the switch approach consistently outperforms the baseline method in handling various workloads.

Discussion. Overall, we believe that our switch-based protocol for optimistic concurrency is highly promising. It is important is that the presented approach of fixing optimistic reads for RDMA does not require alterations in the RDMA protocol, since the switch executes all operations that would be normally executed on the storage nodes seamlessly on the switch. An interesting case is, however, if the switch memory is insufficient to store all tuples' versions. In that case, we can use the switch to only cache the most frequently accessed tuples as skew in workloads is common, falling back to a more traditional handling method (e.g., atomics) for cold tuples. However, this point will be improved in the future, as newer switch generations already have more memory (e.g., the Tofino2), and big cloud vendors commonly employ their own hardware and are not limited by the specifications of consumer products. Such switches could employ a similar architecture as in Ethernet-based programmable switches. Finally, as a last interesting trend, the lock table size on ConnectX-based RDMA NICs may not increase in the future because we have seen the same performance over the last four generations. Therefore, it is necessary to integrate new modern hardware into the RDMA landscape to improve the systems that use RDMA further.

6 Discussion of Other Approaches

This section discusses other synchronization techniques with relaxed guarantees.

Consistent Optimistic Read. The synchronization schemes that we discussed before captured latch semantics. That is, they include an additional validation step after retrieving a data value and operating on it to make sure that the data did not change in the meantime. However, these strong semantics are not required for all use cases [39, 40, 74]. A classic example is a key-value store that is only concerned about returning consistent data to clients and does not consider the clients' operations on the data. Our presented optimistic schemes can be easily adapted to support such relaxed semantics by removing the additional validation step at the end.

Other Incorrect Schemes. Another common technique is called bookend versioning which we briefly showed in Figure 1(a). While incorrect, the intuition of this approach is similar to versioning, where the second version (embedded in the data) validates that the data was not concurrently updated during the RDMA read. This approach suffers from the ordering problem, and its use in recent literature (e.g., [58]) is a testament to the subtleties involved in one-sided synchronization.

Out-of-Place Updates. An alternative design to the data consistency techniques described in Section 4.3 is to leverage out-of-place updates [7, 48, 61], eliminating concurrent data modifications using a copy-on-write strategy. This approach has the same communication overhead as the (broken) single RDMA read versioning technique but comes at the obvious cost of additional storage overhead. Out-of-place updates also enable the use of a technique we call *marking*. Marking is a simple detection technique that can alert readers by relying on a logical flag to indicate that a concurrent write is taking place. This approach manifests as logical insertion or deletion and a busy or pending update flag. However, when combined with in-place updates, marking suffers from the same re-ordering problem as versioning and can fail to detect an inconsistency because of the PCIe bus. Unfortunately, this incorrect use of marking has found its way into existing literature [17, 69].

7 Lessons Learned and Conclusion

As the first analytical study of RDMA synchronization primitives, we aim to leave the reader with a distilled perspective on the lessons learned from our work. These lessons are cast as *anti-patterns*,

which reflect common mistakes that lead to incorrect designs, and important design *considerations*, which do not impact correctness but can be detrimental to performance and system complexity.

Anti-pattern #1.
Reliance on cache line order within a single RDMA read.

The first anti-pattern stems directly from the lack of ordering in the PCIe bus, which was first introduced in Section 1 and discussed in more detail in Section 4.2. Designs fall prey to this anti-pattern and thus incorrectly assume that an RDMA read observes the same order in which writes to different cache lines are made. Examples that relied on this anti-pattern include bookend versioning [40, 58] and marking with in-place updates [17, 69]. For instance, in [69], a writer first marks data as invisible to readers, then updates the value. As we have already established (cf. Section 4.2), a reader may observe these two steps out-of-order and, therefore, could accidentally assume a corrupted read is consistent. Worse, the version retrieved during this initial read could later be (incorrectly) validated. This behavior is identical to the broken versioning scheme discussed in Section 4.2.

Anti-pattern #2.
Reliance on the ordering of overlapping reads.

One solution to address the first anti-pattern is to manually enforce an order by issuing multiple reads. However, there are also pitfalls to this approach because distinct RDMA read operations also have subtle ordering guarantees. The second anti-pattern captures the lack of ordering between overlapping reads from the same connection. While RDMA operations can be fenced to enforce ordering in specific cases, this does not pertain to RDMA reads. In other words, there is currently no mechanism to enforce the order in which a remote machine processes RDMA reads without waiting for completion. Optimistic techniques like versioning hence require two sequential reads to ensure that the version is read before the data, as explained in Section 4.3. One-shot optimistic approaches, such as FaRM, do not suffer from this because they detect inconsistency at the granularity of cache lines.

Anti-pattern #3.
Reliance on the atomicity of RDMA writes and RDMA atomics.

The third anti-pattern results from the lack of atomicity guarantees by RDMA write and RDMA atomic operations in the RDMA specification. As Section 3.3 describes, various optimizations to improve pessimistic synchronization techniques exist. Recall that the write-unlatch optimization is only permissible when the locking primitive exclusively utilizes CAS operations. Again, this is because there is a store buffer in the RNIC. CAS operations are a special case because the write-back to memory is conditional. While it is possible to correctly utilize the write-unlatch optimization [56], as shown in Section 4.5, it requires a careful understanding of the underlying hardware. Therefore, we generally caution against designs that combine RDMA writes with RDMA atomic operations but acknowledge that in some instances where the lack of atomicity does not break semantics, it can yield better performance.

Anti-pattern #4.
Reliance on future hardware generations to solve existing problems.

The fourth anti-pattern stems from the assumption that iterative advancements in network hardware will automatically resolve inefficiencies, such as those observed with RDMA atomics. Despite newer hardware generations being designed to handle higher data throughput and reduce latency, the throughput of contented RDMA atomics has not improved across the last two generations and continues to be a bottleneck, as detailed in Section 5.1. Consequently, it is essential to explore alternative methods for scaling database workloads rather than solely relying on future generations of hardware NICs.

Consideration #1.
Data alignment impacts the RDMA atomic scalability.

The first consideration is described in detail in Section 3.2, but we review it here. Because of the lock table present in current RNICs, physical contention between independent latches can arise. We demonstrate that this is not only easily demonstrated by microbenchmarks that highlight the behavior but also have an important role in the scalability of a real system (i.e., NAM-DB). Hence, we advocate that system designers pay close attention to their data alignment when leveraging RDMA atomic operations to avoid this physical contention.

Consideration #2.
Optimistic synchronization performs best under high contention.

Conventional wisdom suggests that pessimistic synchronization pays off in high-contention write-heavy workloads because it eliminates excessive retries. However, our analysis demonstrates that this perspective does not hold for RDMA-based synchronization. Notably, in Section 4.5, we show that in write-heavy workloads, the optimistic approaches can match and even surpass pessimistic ones. While readers often retry in optimistic schemes, pessimistic approaches are subject to RNIC contention. Our results suggest that pessimistic approaches are beneficial when there is less skew, and the number of concurrent workers is small. Interestingly, this also applies in the read-only case since the sequential overhead is low compared to the optimistic strategies, which require validation. Therefore, optimistic synchronization should be preferred for highly skewed workloads to avoid RDMA atomic bottlenecks. As we established with the B-tree performance in Figure 11, this is particularly beneficial when there is a single point of contention, e.g., the root node of a B-tree.

Consideration #3.
Optimistic synchronization has non-negligible overheads.

Although they outperform pessimistic synchronization in many scenarios, there are computational and storage trade-offs among the various optimistic approaches, which we discussed in Section 4.5. Versioning requires an additional RDMA read compared to the other techniques, which increases operation latency and is most evident for small data sizes. On the other hand, CRC is computationally expensive, becoming untenable at large data sizes ($\geq$ 4 KB). Cache line versioning generally performs well but has an increased storage cost, not to mention that software must either handle the embedded versions or copy the data out. While optimistic synchronization is beneficial in many scenarios, it is not a silver bullet as they often have more complex designs compared to pessimistic schemes.

Consideration #4.
Pessimistic synchronization is more "future proof".

As demonstrated, subtle hardware characteristics can have a profound impact on correctness. We highlight this using a widespread deployment but point out that disaggregated memory technologies are in active development. For example, advancements like CXL [50]—a protocol built on PCI-e to support memory coherence across devices—are poised to disrupt the status quo. Changes to intermediate components of RDMA communication may alter the behavior of concurrent RDMA reads and writes, and therefore optimistic synchronization schemes, in unpredictable ways. In contrast, RDMA Atomic operations implement a well-established higher-level abstraction independent of hardware implementation. Hence, system designers should lean toward simple pessimistic synchronization when prioritizing production stability until the community has successfully converged on well-defined memory semantics for RDMA reads and writes.

Consideration #4.
Leverage programmable networking hardware for synchronization is highly promising.

As demonstrated in Section 5, integrating compute resources throughout the entire data path can significantly improve performance and consistency, especially for concurrent RDMA reads and writes. Modern networking hardware, such as programmable switches, offers a novel architecture that enables users to tailor RDMA operations to their specific use cases. In our prototype, we illustrate how a programmable switch can be used to implement correct optimistic RDMA reads, a task that would require careful synchronization and considerably more round-trips with traditional NICs. Consequently, we encourage system designers to consider the computing capabilities across the entire network and leverage all of its new functionalities.

Conclusion. This article is the first paper that holistically (1) highlights the subtleties of RDMA-based synchronization regarding the correctness, (2) provides a robust performance analysis of existing synchronization techniques and optimizations, (3) demonstrates pitfalls of existing designs, and (4) offers a set of anti-patterns and design considerations for enabling developers to design correct and high-performance RDMA-enabled system.

Acknowledgments

We also thank hessian.AI, DFKI, Mellanox, and 3AI for their support. We also like to thank Torsten Hoefler for an insightful discussion about the memory model of RDMA and RDMA's ordering guarantees.

References

[1] ARM. 2018. Arm CoreLink CCI-550 Cache Coherent Interconnect Technical Reference Manual. https://developer.arm.com/documentation/100282/0100/?lang=en. (2018). https://developer.arm.com/documentation/100282/0100/?lang=en

[2] ARM. 2021. Introducing the AMBA Coherent Hub Interface. (2021). https://developer.arm.com/documentation/102407/0100

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming protocol-independent packet processors. *Comput. Commun. Rev.* 44, 3 (2014), 87–95. https://doi.org/10.1145/2656877.2656890

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review* 43, 4 (2013), 99–110.

[5] Mihai Budiu and Chris Dodd. 2017. The p416 programming language. *ACM SIGOPS Operating Systems Review* 51, 1 (2017), 5–14.

[6] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB serverless. In *Proceedings of the 2021 International Conference on Management of Data*. ACM. https://doi.org/10.1145/3448016.3457560

[7] Benjamin Cassell, Tyler Szepesi, Bernard Wong, Tim Brecht, Jonathan Ma, and Xiaoyi Liu. 2017. Nessie: A decoupled, client-driven key-value store using RDMA. *IEEE Trans. Parallel Distributed Syst.* 28, 12 (2017), 3537–3552. https://doi.org/10.1109/TPDS.2017.2729545

[8] Yeounoh Chung and Erfan Zamanian. 2015. Using RDMA for lock management. *CoRR* abs/1507.03274 (2015). arXiv:1507.03274 http://arxiv.org/abs/1507.03274

[9] NVIDIA Coporation. 2021. NVIDIA InfiniBand Adaptive Routing Technology. Whitepaper WP-10326-001_v01. (2021).

[10] Andrei Marian Dan, Patrick Lam, Torsten Hoefler, and Martin T. Vechev. 2016. Modeling and analysis of remote memory access programming. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30–November 4, 2016)*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 129–144. https://doi.org/10.1145/2983990.2984033

[11] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *NSDI*.

[12] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: Distributed transactions with consistency, availability, and performance. In *SOSP*.

[13] Jingwen Du, Fang Wang, Dan Feng, Changchen Gan, Yuchao Cao, Xiaomin Zou, and Fan Li. 2023. Fast one-sided RDMA-based state machine replication for disaggregated memory. *ACM Trans. Archit. Code Optim.* (Mar 2023). https://doi.org/10.1145/3587096 Just Accepted.

[14] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Low-latency communication for fast DBMS using RDMA and shared memory. In *ICDE*.

[15] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, Sujata Banerjee and Srinivasan Seshan (Eds.). USENIX Association, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[16] Torsten Hoefler, Duncan Roweth, Keith Underwood, Bob Alverson, Mark Griswold, Vahid Tabatabaee, Mohan Kalkunte, Surendra Anubolu, Siyan Shen, Abdul Kabbani, Moray McLaren, and Steve Scott. 2023. Datacenter Ethernet and RDMA: Issues at Hyperscale. arXiv:2302.03337 [cs.NI].

[17] Chenchen Huang, Huiqi Hu, Xuecheng Qi, Xuan Zhou, and Aoying Zhou. 2021. RS-store: RDMA-enabled skiplist-based key-value store for efficient range query. *Frontiers of Computer Science* 15, 6 (Sept. 2021). https://doi.org/10.1007/s11704-020-0126-6

[18] Ram Huggahalli, Ravi R. Iyer, and Scott Tetrick. 2005. Direct cache access for high bandwidth network I/O. In *32st International Symposium on Computer Architecture (ISCA 2005),* (4–8 June 2005, Madison, Wisconsin). IEEE Computer Society, 50–59. https://doi.org/10.1109/ISCA.2005.23

[19] InfiniBand Trade Association 2007. *InfiniBand Architecture Specification Volume 1*. InfiniBand Trade Association. Release 1.2.1.

[20] InfiniBand Trade Association. 2010. RDMA Over Converged Ethernet (RoCE). https://cw.infinibandta.org/document/dl/7148. (2010).

[21] Intel. 2012. Intel Data Direct I/O Technology (Intel DDIO): A Primer. (Feb 2012). https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf

[22] Matthias Jasny, Lasse Thostrup, Tobias Ziegler, and Carsten Binnig. 2022. P4DB - The case for in-network OLTP. In *SIGMOD '22: International Conference on Management of Data, (Philadelphia, PA, June 12–17, 2022),* Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1375–1389. https://doi.org/10.1145/3514221.3517825

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

[24] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. *login Usenix Mag.* 41, 3 (2016).

[25] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*.

[26] Tejas Karmarkar. 2015. Availability of Linux RDMA on Microsoft Azure. Online. (July 2015). https://azure.microsoft.com/en-us/blog/azure-linux-rdma-hpc-available/

[27] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated memory with operator off-loading for database engines. In *12th Conference on Innovative Data Systems Research (CIDR 2022) (Chaminade, CA, , January 9–12, 2022)*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p11-korolija.pdf

[28] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2014. Software-defined networking: A comprehensive survey. *Proc. IEEE* 103, 1 (2014), 14–76.

[29] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull.* 42 (2019), 73–84.

[30] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *12th International Workshop on Data Management on New Hardware (DaMoN 2016)* (San Francisco, CA, June 27, 2016). ACM, 3:1–3:8. https://doi.org/10.1145/2933349.2933352

[31] Edgar A. León, Kurt B. Ferreira, and Arthur B. Maccabe. 2007. Reducing the impact of the MemoryWall for I/O using cache injection. In *15th Annual IEEE Symposium on High-Performance Interconnects,(HOTI 2007, Stanford, CA, USA,*

August 22–24, 2007), John W. Lockwood, Fabrizio Petrini, Ron Brightwell, and Dhabaleswar K. Panda (Eds.). IEEE Computer Society, 143–150. https://doi.org/10.1109/HOTI.2007.8

[32] Alberto Lerner, Carsten Binnig, Philippe Cudré-Mauroux, Rana Hussein, Matthias Jasny, Theo Jepsen, Dan R. K. Ports, Lasse Thostrup, and Tobias Ziegler. 2023. Databases on modern networks: A decade of research that now comes into practice. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3894–3897.

[33] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. 2003. High performance RDMA-based MPI implementation over InfiniBand. In *ICS*.

[34] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto smartNICs using iPipe. In *ACM Special Interest Group on Data Communication*. 318–333.

[35] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. 2015. On the design and scalability of distributed shared-data databases. In *SIGMOD*.

[36] Teng Ma, Kang Chen, Shaonan Ma, Zhuo Song, and Yongwei Wu. 2021. Thinking more about RDMA memory semantics. In *IEEE International Conference on Cluster Computing, (CLUSTER 2021*) (Portland, OR, September 7–10, 2021). IEEE, 456–467. https://doi.org/10.1109/Cluster48925.2021.00033

[37] Teng Ma, Dongbiao He, and Gordon Ning Liu. 2021. HybridSkipList: A case study of designing distributed data structure with hybrid RDMA. In *IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC 2021)* (Madrid, Spain, July 12–16, 2021). IEEE, 68–73. https://doi.org/10.1109/COMPSAC51774.2021.00021

[38] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference* (San Jose, CA, June 26-28, 2013), Andrew Birrell and Emin Gün Sirer (Eds.). USENIX Association, 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[39] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX ATC*.

[40] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. 2016. Balancing CPU and network in the cell distributed B-Tree store. In *USENIX ATC*.

[41] Sundeep Narravula, A. Marnidala, Abhinav Vishnu, Karthikeyan Vaidyanathan, and Dhabaleswar K. Panda. 2007. High performance distributed lock management services using network-based remote atomic operations. In *7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007)* (14–17 May 2007, Rio de Janeiro, Brazil). IEEE Computer Society, 583–590. https://doi.org/10.1109/CCGRID.2007.58

[42] Jacob Nelson and Roberto Palmieri. 2020. Performance evaluation of the impact of NUMA on one-sided RDMA interactions. In *SRDS*.

[43] Nvidia. 2023. ConnectX-7 Ethernet Datasheet. (2023). https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf

[44] PCI-SIG. 2014. PCI express base specification revision 4.0. (2014).

[45] Dan R. K. Ports and Jacob Nelson. 2019. When should the network be the computer?. In *Workshop on Hot Topics in Operating Systems*. 209–215.

[46] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. 2007. *A Remote Direct Memory Access Protocol Specification*. Technical Report. https://doi.org/10.17487/rfc5040

[47] Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, and Thomas G. Robertazzi. 2013. Design and performance evaluation of NUMA-aware RDMA-based end-to-end data transfer systems. In *HiPC*.

[48] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4–6, 2020*. USENIX Association, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[49] H. Shah, F. Marti, W. Noureddine, A. Eiriksson, and R. Sharp. 2014. *Remote Direct Memory Access (RDMA) Protocol Extensions*. Technical Report. https://doi.org/10.17487/rfc7306

[50] Debendra Das Sharma. 2019. *Compute Express Link*. Technical Report. Compute Express Link.

[51] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. 2021. CliqueMap: Productionizing an RMA-based distributed caching system . In *ACM SIGCOMM 2021 Conference*. ACM. https://doi.org/10.1145/3452296.3472934

[52] Dan Tang, Yungang Bao, Weiwu Hu, and Mingyu Chen. 2010. DMA cache: Using on-chip storage to architecturally separate I/O data from CPU data for improving I/O performance. In *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010)* (9–14 January 2010, Bangalore, India), Matthew T. Jacob, Chita R. Das, and Pradip Bose (Eds.). IEEE Computer Society, 1–12. https://doi.org/10.1109/HPCA.2010.5416638

[53] Konstantin Taranov, Fabian Fischer, and Torsten Hoefler. 2022. Efficient RDMA Communication Protocols. (2022). arXiv:cs.NI/2212.09134

[54] Konstantin Taranov, Salvatore Di Girolamo, and Torsten Hoefler. 2021. CoRM: Compactable remote memory over RDMA. In *SIGMOD '21: International Conference on Management of Data, Virtual Event* (China, June 20–25, 2021). 1811–1824. https://doi.org/10.1145/3448016.3452817

[55] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 2020)* (July 15–17, 2020), Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 33–48. https://www.usenix.org/conference/atc20/presentation/tsai

[56] Chao Wang and Xuehai Qian. 2021. RDMA-enabled concurrency control protocols for transactions in the cloud era. *IEEE Transactions on Cloud Computing* (2021), 1–1. https://doi.org/10.1109/tcc.2021.3116516

[57] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A rapid prototyping framework for P4. In *Symposium on SDN Research*. 122–135.

[58] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A write-optimized distributed B+Tree index on disaggregated memory. In *2022 International Conference on Management of Data*. ACM. https://doi.org/10.1145/3514221.3517824

[59] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2022. The case for distributed shared-memory databases with RDMA-Enabled memory disaggregation. *CoRR* abs/2207.03027 (2022). https://doi.org/10.48550/arXiv.2207.03027 arXiv:2207.03027

[60] Tinggang Wang, Shuo Yang, Hideaki Kimura, Garret Swart, and Spyros Blanas. 2020. Efficient usage of one-sided RDMA for linear probing. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB 2020)* (Tokyo, Japan, August 31, 2020), Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 1–13. http://www.adms-conf.org/2020-camera-ready/ADMS20_06.pdf

[61] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. 2015. HydraDB: A resilient RDMA-driven key-value middleware for in-memory cluster computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2015)* (Austin, TX, November 15–20, 2015), Jackie Kern and Jeffrey S. Vetter (Eds.). ACM, 22:1–22:11. https://doi.org/10.1145/2807591.2807614

[62] Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. 2021. XStore: Fast RDMA-Based ordered key-value store using remote learned cache. *ACM Trans. Storage* 17, 3 (2021), 18:1–18:32. https://doi.org/10.1145/3468520

[63] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled distributed transactions: Hybrid is better!. In *OSDI*.

[64] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*.

[65] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. Netlock: Fast, centralized lock management using programmable switches. In *Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. 126–138.

[66] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. 2020. NetLock: Fast, centralized lock management using programmable switches. In *SIGCOMM '20: Proceedings of the 2020 Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, August 10–14, 2020) Henning Schulzrinne and Vishal Misra (Eds.). ACM, 126–138. https://doi.org/10.1145/3387514.3405857

[67] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2016. The end of a myth: Distributed transactions can scale. *CoRR* abs/1607.00655 (2016).

[68] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2021. Chiller: Contention-centric transaction execution and data partitioning for modern networks. *SIGMOD Rec.* 50, 1 (2021).

[69] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 2022)* (Santa Clara, CA, February 22–24, 2022), Dean Hildebrand and Donald E. Porter (Eds.). USENIX Association, 51–68. https://www.usenix.org/conference/fast22/presentation/zhang-ming

[70] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proc. VLDB Endow.* 14, 10 (2021), 1900–1912. https://doi.org/10.14778/3467861.3467877

[71] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A fast and cost-efficient storage engine using DRAM, NVMe, and RDMA. In *SIGMOD '22: International Conference on Management of Data* (Philadelphia, PA, , June 12–17, 2022). ACM, 685–699. https://doi.org/10.1145/3514221.3526187

[72] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2023. Design guidelines for correct, efficient, and scalable synchronization using one-sided RDMA. *Proc. ACM Manag. Data* 1, 2 (2023), 131:1–131:26. https://doi.org/10.1145/3589276

[73] 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 *SIGMOD*.

[74] Pengfei Zuo, Qihui Zhou, Jiazhao Sun, Liu Yang, Shuangwu Zhang, Yu Hua, James Cheng, Rongfeng He, and Huabing Yan. 2022. RACE: One-sided RDMA-conscious extendible hashing. *ACM Transactions on Storage* 18, 2 (May 2022), 1–29. https://doi.org/10.1145/3511895

Received 21 May 2024; revised 21 May 2024; accepted 3 February 2025