Title

HybridSkipList+: Rethinking Distributed Skiplist With Hybrid RDMA and Caching

Journal

IEEE Transactions on Computers, 75(9)

ISSN

0018-9340

Authors

Lu, Yilei

Ma, Teng

He, Dongbiao

et al.

Publication Date

2026-09-01

DOI

10.1109/tc.2026.3701257

HybridSkipList+: Rethinking Distributed SkipList with Hybrid RDMA and Caching

Yilei Lu, Teng Ma, Dongbiao He, *Member, IEEE*, Xian Yu, Zhe Wang, *Member, IEEE*, Cedric Westphal, *Senior Member, IEEE*, Linghe Kong, *Senior Member, IEEE*

Abstract—RDMA (Remote Direct Memory Access) delivers both high performance and OS kernel bypass. It has thus significantly impacted the modern data center. By exploiting memory semantics operations, RDMA-based data structures have shown great potential for high scalability and a large reduction in CPU utilization when compared with the traditional Ethernet. *Skiplist* is an elegant sorted index data structure. However, it exhibits low throughput upon RDMA memory semantics due to heavy remote access overhead. In this work, we revisit the current design paradigm of RDMA such as operation type, transport type, caching, and memory management. Accordingly, we propose a one-sided/two-sided hybrid paradigm to gain high scalability and at the same time achieve high throughput based on lock-based concurrent *Skiplist*. With this hybrid RDMA paradigm, we design *HybridSkipList+*, which is a distributed *Skiplist*. To reduce the number of expensive network round trips, a client-sided coherent cache with pull-based synchronization is presented. Furthermore, we design a semi-continuous memory allocator to improve the locality of each RDMA access. With an in-depth analysis of the design choices, we implement *HybridSkipList+* and deploy our method on an eight-machine RDMA cluster. The evaluations show it outperforms two baselines by 4.41× and 3.45× respectively in typical utilization conditions.

Index Terms—RDMA, SkipList, Hybrid Communication Paradigm

1 Introduction

Distributed applications (e.g., distributed storage [2], [3], [4], [5]) require to communicate between machines to access remote data with high throughput, low latency, and high reliability. With the growing popularity of Remote Direct Memory Access (RDMA) in modern data centers [6], RDMA-based data structures are touted as being a potential solution to meet the high-performance requirements of current distributed applications.

RDMA traditionally involves both sides, namely the local and remote sides (*two-sided* RDMA). This paradigm can harm performance and shows poor scalability on the server side because of the heavy CPU consumption and massive out-bound RDMA operations [7] which are issued by the sender's CPU. Chen et al. [8] present that the aggregated throughput of RPC decreases sharply by nearly 50% if the number of clients increases from 40 to 120. Unlike the TCP/IP stack, RDMA also provides a *one-sided* memory semantics paradigm. The one-sided paradigm allows a machine to directly access remote memory regions without the involvement of the remote side's CPU and thus has good performance and low CPU consumption.

Several existing in-memory distributed applications explore utilizing the power of RDMA, and many of them are *unsorted key-value stores*. For example, Pilaf [9], C-hint [10] and Nessie [11] allow clients to directly read key-value entries from the server's memory via one-sided operation. DrTM [12] combines one-sided operations and HTM (Hardware Transactional Memory) to access remote key-value structures for fast in-memory transaction processing.

For sorted data structures, their operations always perform within $O(logN)$ time complexity to locate the key. Thus, compared with two network round trips when using a two-sided paradigm, at least $logN$ network round trips are required under the one-sided paradigm, which is expensive. On the other hand, upon the UD (Unreliable Datagram) network, the two-sided operation can be exploited to implement an RPC-based key-value store [13], [14], [15] due to the high performance of UD mode.

As of now, only a small portion of existing works have tried a hybrid paradigm, which enjoys the benefits from both one-sided and two-sided approaches. Based on OCC (Optimistic concurrency control), DrTM-H [16] evaluates various phases (execution, validation, logging, commit) of a transaction with hybrid implementations. However, these works lack considerations of how to squeeze out the performance of the hybrid paradigm in many aspects, such as the client-side cache and the characteristics of the data structure.

Compared to other sorted data structures, such as B-trees, *Skiplist* is an elegant sorted data structure without complex split and combine operations, which means fewer write contentions. *Skiplist* is easy to parallel access, so as to be friendly for read operation [17]. Log-structure merge-tree (LSMT) such as RocksDB and LevelDB uses *Skiplist* as a basic component. However, there is still no specific work on how to design *Skiplist* specifically for RDMA networks.

In this paper, we investigate the design choices to implement an RDMA-based *Skiplist*. These design choices include: *1) Lock-Free/Lock-Based; 2) Reliable-Connection/Unreliable-Datagram; 3) One-sided/Two-Sided; 4) Caching/No-Caching; 5) Snooping/Directory-based for cache updates, and* 6) Continuous/Non-continuous Memory. As a result, we demonstrate that a RDMA-based *Skiplist* should adopt a one-sided and two-sided *hybrid* communication

• *An earlier version of this work [1] appeared in IEEE Computer Society Signature Conference on Computers, Software and Applications (COMSAC 2021).*

strategy with *directory-based* coherent protocol to *cache part of index structures*. Additionally, we contend that: while using RDMA as the underlying transport, *choosing a suitable lock such as retry-based lock* and *use moderate-granularity memory regions* to make a trade-off between continuous and non-continuous memory, can deliver more performance gain.

Based on six design choices, we present *HybridSkipList+*, which releases the true power of the "hybrid is better" [16] concept. It combines one-sided read operations and two-sided write operations, as there is heavy contention on write operations. Another challenge is that range query will incur long latency, so *HybridSkipList+* uses one-sided operation to locate the left and right node and then uses two-sided based RPC to traverse target nodes with server side involvement. *HybridSkipList+* adopts lock-based concurrency control, and the server side is responsible for handling the concurrency to reduce expensive network round trips. By doing so, this only constitutes a very small fraction of the server's CPU time since using one-sided operations avoid the involvement of the server's CPU.

To reduce the communication fan-out brought by indexing, *HybridSkipList+* caches high-degree data structure nodes at the client side to accelerate traversal.

Accordingly, a pull-based cache coherence protocol is proposed, and multiple clients in the same machine can share the cache to reduce synchronization overhead. Consequently, it provides eventual consistency for the read operation. To guarantee most memory regions are continuous and at the same time increase the memory utilization, we propose a semi-continuous memory allocator for *HybridSkipList+*. It uses moderate size hugepage for each memory region, and each allocation request will be assigned to an appropriate memory region to minimize random access.

We implement *HybridSkipList+* and deploy in an eight-machine RDMA cluster. When ten clients concurrently access *HybridSkipList+* under 10% write operation workload, the evaluations show it has good scalability with only 38.5% throughput reduction compared to one-to-one scenario. It outperforms two baselines (pure one-sided *Skiplist* and pure two-sided *Skiplist*) by 4.41× and 3.45× respectively.

To the best of our knowledge, there is still no quantitative study on distributed *Skiplist* that discusses the hybrid usage of one/two-sided RDMA operations. Further, most design principles in this paper are general and can be applied to distributed data structures such as B-Tree and LSM Tree.

2 Background

2.1 RDMA Preliminaries

In this section, we offer a background of RDMA and RDMA Network Interface Card (RNIC). We name the server taking the initiative to access the data in the remote side as the *active side*, and the remote side as the *passive side*.

Programming Paradigm. The communication paradigms of RDMA can be subdivided into two types: 1) One-sided operations: active side can access the memory in the target node directly without the need for the passive side's CPU to initiate. These operations include RDMA Write, Read, and Atomic (*compared and swap/fetch and add*), which is *Memory Semantics* and 2) Two-sided operations: RDMA Send/Recv, which can be seen as traditional Ethernet (*Channel Semantics*) to some extent.

While using memory semantics operations, the CPU in the passive side does not participate in the network communication. The memory is directly manipulated by the RNIC without involving the OS kernel in the passive side nor involving the CPU. This bypass speeds up the process on the passive side and therefore delivers a higher throughput for certain operations [18], [19], [20]. However, it comes at a cost in the sense that the passive side is not aware of the memory access results and cannot assist if needed.

Transport Type. RDMA supports three transport types: Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD). One-sided requires strong constraints of protocol and reliability. Thus, `RDMA Write` is supported by both RC and UC. `RDMA Read` and `RDMA Atomic` are only supported by RC. All transport types support two-sided (`RDMA Send/Recv`) operations. In RC, the RNIC is responsible for the reliable delivery of each packet. As UC and UD ignore packet drops, they can process more operations than RC [21].

Processing. Before establishing a connection between the two sides, each side should create a Queue Pair (QP), which is a memory-mapped structure. Each QP contains two Completion Queues (CQs): one for sending requests (Sending Queue, SQ), and the other for receiving requests (Receiving Queue, RQ). To control access to the memory, Memory Region are registered with the RDMA device and a remote protection key (rkey) is generated. Memory that can be accessed by both sides is known as RDMA-enabled memory [22]. To access remote memory with a one-sided operation, each RDMA operation requires posting a work request (WR) to the correct QP on the active side. A completion queue entry (CQE) is selectively produced to notify the end of the access and pushed to the SQ on the active side.

Hardware Specification. There is on-chip memory (SRAM) on the RNIC for caching the address translation tables (e.g., MR information), the data for the QPs and the CQE cache [8], [23], [24]. Due to considerations of cost and energy, the capacity of such SRAM is currently limited to only the MB scale. Unfortunately, in RC mode, establishing a new connection creates a new QP and a new address translation tables on each side, so SRAM is not able to store all the meta-data. It is inevitable to incur cache swap in/out. Thus, the performance is degraded when issuing a massive number of concurrent out-bound RDMA operations.

Note that using UD mode does not require to construct a QP for each connection and thus incurs no performance degradation when increasing the concurrent connections. Previous works [25], [26] show that next-generation RNICs may address this, but recent works [27], [28] indicate this issue must still be considered.mitigate this issue.

2.2 *Skiplist*

Skiplist [29] is a popular in-memory index structure that has been widely adopted by databases [30], [31]. It is a sorted data structure containing a set of elements ($\{a_i \| i \in [1, n]\}$, in which $a_i < a_j$ if $i < j$). It supports commonly-used data structure operations such as Insert, Contain, Remove

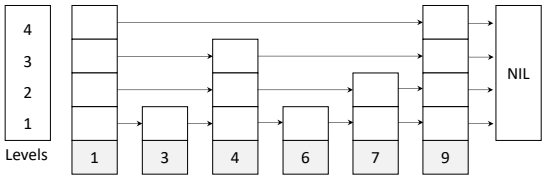

Fig. 1: The Structure of *Skiplist*.

methods and even range queries [1]. These operations can be performed in a *Skiplist* within $O(logN)$ (on average, assuming there are N nodes in the *Skiplist*).

From another perspective, a *Skiplist* consists of a hierarchy of linked lists, which behave as a probabilistic data structure (see Figure 1). Each node is created with a random level and exists in all lists up to its level. The length of each linked list will decrease with a factor p (e.g., 0.5) from the bottom layer to the top layer. p is the probability that a node at a given layer is also inserted at the layer above. The $layer_0$ list contains all the nodes, and each $layer_i$ list is a sub-list of the $layer_{i-1}$ list. This is an important feature of a *Skiplist*: *the higher-level lists are always contained in the lower-level lists*. A key characteristic is that nodes are linked to predecessors and successors at each level, but these adjacent nodes may be scattered in memory, potentially causing costly random memory accesses during traversal.

A *Skiplist* can be implemented as a lock-based data structure. It can also be implemented as a lock-free one to support concurrent access [32], [33], [34]. We will discuss the differences in Section 3.

As the demand for supporting large-scale datasets, distributed data structures have emerged as a necessary component. Typically, distributed hash tables [35] and distributed in-memory b-trees [36] are widely applied to systems such as database and key-value store. To our knowledge, there is little work to discuss distributed *Skiplist*.

Optimality bound of *Skiplist*. In a distributed *Skiplist*, the main overheads come from the network (i.e., RDMA) and the server-side CPU (i.e., data structure operations) separately. Table 1 compares the throughput between RDMA one-sided Read/Write access and two *Skiplist* methods: Contain and Insert ($log(N)$). We use a state-of-the-art implementation [37] of *Skiplist*. The results show a significant performance gap (2.4∼34×) between *Skiplist* and each RDMA operation, and it is evidence that the bottleneck of distributed *Skiplist* is CPU, not the network.

TABLE 1: Comparisons between RDMA and *Skiplist*.

Type (CX:ConnectX)	Complexity	Read/Contain	Write/Insert
RDMA(CX-3,40Gbps)	log(1)	6.2MOPS	6.5MOPS
RDMA(CX-5,100Gbps)	log(1)	14.7MOPS	14.5MOPS
SkipList(1 thread)	log(n)	0.39MOPS	0.50MOPS
SkipList (20 threads)	log(n)	2.6MOPS	2.4MOPS

1. In this paper, read operation consists of Contain and write operation consists of Insert/Update/Remove

3 Design Choices

In this section, we discuss some of the trade-offs that influenced our design decisions, namely: one-sided vs two-sided RDMA operations; reliable vs unreliable data connections; lock-based vs lock-free; caching vs non-caching; snooping vs directory approaches to update caches; and continuous vs non-continuous memory allocation.

3.1 One-sided vs. Two-Sided

One-sided operation benefits from low CPU consumption, while two-sided operation requires fewer network round trips.

In most cases, with the same round trip time, one-sided operations have better performance than two-sided operation [7], [9], as it bypasses the operating system kernel on the remote side. Yet, executing a data structure operation (e.g., an Insert operation for *Skiplist*) still requires multiple data modifications (namely, $log(n)$ operations for *Skiplist*, where n is the node number).

Thus, using one-sided RDMA operations to implement data structures leads to the same number (say, $log(n)$) of network round trips. With two-sided operation, most data structure methods can be regarded as RPC operations, and require only *one* network round trips. If the data structure has a high complexity and is hard to index (e.g., Binary Search Tree, *Skiplist*), using two-sided operation always requires fewer network round trips (one in-bound and one out-bound operations in most cases) than using one-sided operation (on average $log(n)$ in-bound operations). However, two-sided operations require CPU involvement. As we mentioned in Section 2.1, this will incur a performance degradation due to massive out-bound RDMA operations.

Besides, using one-sided operation to implement data structure methods can lead to lower (nearly zero) CPU consumption on the passive side. The root causes are: 1) one-sided operations bypass the kernel on the passive side and thereby avoid the involvement of the passive-side CPU; and 2) unlike two-sided operations, which cope with data structure concurrency on the passive side, the concurrency functionality is moved to active side by using one-sided operation, thus avoiding the passive side CPU involvement in heavy concurrency control.

To enjoy the benefits from both one/two-sided operations, we adopt a hybrid strategy which combines both one-sided for no-locking operations (Contain) and two-sided for operations with heavy contention (Insert/Remove). With one-sided operation, other optimizations such as *unsignal*, *inline* [23] and *Doorbell* [38] can be applied.

3.2 Reliable Connection (RC) vs. Unreliable Datagram (UD)

RC supports all RDMA operations but requires more connections.

As we discussed in Section 2.1, increasing connections (i.e., QPs) will lead to a performance drop in the RC mode. Because the UD mode is based on unconnected transport, using the UD mode requires fewer QPs.

In distributed data structures, the client usually performs one-sided operations. Therefore, the performance of the system will not be significantly impacted by large amounts of incoming operations on the server side. Since clients

typically outnumber servers and perform fewer operations each, any minor performance impact is negligible. We also have demonstrated that the RC mode has comparable scalability to the UD mode. Additionally, even several works [14] present that packet loss occurs in congested scenarios and that using the UD mode cannot guarantee reliable delivery. It is important to note that only the RC mode can support both one-sided and two-sided RDMA operations when using a hybrid strategy. As a result, we have chosen the RC mode for our proposal.

3.3 Lock-Free vs. Lock-Based

Lock-based is friendly to read operations and lock-free is friendly to write.

As we mentioned in Section 2, the principal advantage of a lock-based implementation is that it is much simpler, and much easier to reason about. On the contrary, lock-free implementation is complex but non-blocking.

In the lock-based implementation, the `Contain` method is wait-free, which means it is not influenced by the `Insert` or the `Remove` method and the `Insert` and `Remove` method is deadlock-free. The *Skiplist* property is maintained by locks to avoid structural changes during `Insert` and `Remove` methods. Only the predecessors of a node that is being added or removed are locked, so this implementation still has a high degree of concurrency. In the lock-free implementation, all methods are wait-free. It avoids the usage of locks and hence cannot maintain the original structure. To tackle this, a special constraint is that a key is considered in the *Skiplist*, if a node with that key can be found linked in the bottom layer without the `Removed` flag.

If the node has been linked in the bottom layer, the `Insert` method is considered successful. The `Remove` method is implemented in a similar way.

The completion of a write operation [2] depends on weather the pointer in the bottom layer is linked or not. After a write operation is completed, other read operations assist it to finish the remaining modifications, including linking or unlinking higher layer pointers of the write operation, when traversing the *Skiplist*. In some cases, the pointer chasing higher layer pointers will retry more than once because of concurrent write. At the same time, the execution of read operation will fail, so it will harm the performance of read operations.

For real-world workloads, the proportion of read operations is usually high. Overall, to optimize the majority operations (read operation), the lock-based implementation should be selected with priority.

Meanwhile, to support the hybrid paradigm, both server side and client side require to access *Skiplist* concurrently. Instead of using locks, we implement access to nodes in an atomic manner. It is easy for the server side to use CPU synchronization primitives and for the client side to use RDMA atomic primitives, such as compare and swap (CAS). However, there is still an atomicity issue between local CAS and `RDMA CAS` operations [12]. To avoid it, we choose a lock-based implementation, because lock-free implementation must use RDMA CAS on the client side during read operation to help the writer link or unlink pointers. We can use `RDMA Read` atomically in a lock-based implementation. This is because the size of most attributes (e.g., pointers) except for the value is equal to or smaller than a cacheline (64 bytes), which can be executed atomically in x86 architecture by utilizing `RDMA_Atomic`.

2. In this paper, read operations consist of lookup/find and write operations consist of insert/update/delete.

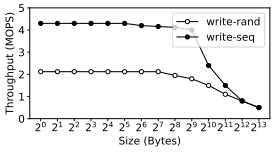

Fig. 2: Comparisons between remote memory sequential and random access.

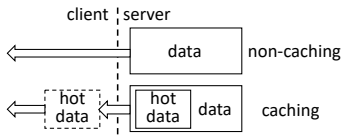

Fig. 3: The process of caching.

3.4 Caching vs. No-caching

Caching has high overhead to maintain coherence; no-caching is less efficient for one-sided data structure operations.

In a client-server model, caching is a technique used to temporarily store data in a local cache on the client side for faster access and reduced network traffic. The cached data is retrieved from the server only when it is needed (see Figure 3). Caching can improve performance and reduce network latency, especially in applications that require frequent access to the same data. However, it also introduces the risk of stale data if the cached data is not updated frequently enough.

Tree-like data structures (e.g., Binary Search Tree, *Skiplist*) always have a hierarchical architecture. Typically, the higher layers are accessed more frequently than the lower layers. Suppose there is a data structure with L layers, and a number N is given such that $L > N$. To speed up the operations on the data structure, the active side can use a cache to temporarily store data from the higher layers that are beyond level N on the passive side. Additionally, there will be more than one active side accessing the data structure concurrently, and each active side will cache a portion of the data structure. To guarantee consistency of the data in the cache in the concurrent access scenario, current works [39] usually adopt synchronization primitives to propagate dirty writes and clear cached read. This approach will bring extra overhead when synchronizing data.

Let us analyze the performance difference between caching and no-caching. After applying caching, for one-sided operations, the average network round trips is $log(n)+1-(L-N)$ compared with the original $log(n)+1$ ($log(n)$ round trips for indexing and one round trip for reading data). On the other hand, each client only needs to cache $\sum_{i=0}^{L-N}\lceil 1/p^i \rceil$ nodes, which is only a fraction $\sum_{i=0}^{L-N}\lceil 1/p^i \rceil / \sum_{i=0}^{L}\lceil 1/p^i \rceil$ of the raw data structure. We conclude that caching is a powerful component to reduce the network round trips, and that caching part of the data appropriately can maximize the performance.

This analysis does not include the overhead to maintain the cache coherency, which we discuss next.

3.5 Snooping vs. Directory-based

Snooping-like protocols exhibit poor scalability and directory-based protocols have high latency.

To maintain cache coherency, there are two commonly-used mechanisms: "snooping" and "directory-based". In the snooping-like protocols, each modification is broadcast to all clients by the server, and the client updates its own cache to a new status. There is a significant drawback: the communication overhead grows linearly with the number of clients. Therefore snooping scales poorly. Directory-based protocols always perform with a higher latency [40] than snooping. The reason is that the client requires at least three network round trips to fetch the new data.

In this paper, we understand snooping as push-based protocol and directory-based as pull-based protocol. The push-based protocol monitors the updates to data and sends messages to invalidate or update cache copies, ensuring that all clients have a consistent view. In the pull-based protocol, when the client modifies the data, other clients update the status of the cache. Because 1) we assume the number of clients is not very large; and 2) network round trips are still expensive compared to local access, we choose pull-based as the essential guideline to design RDMA-based cache coherence protocol.

3.6 Continuous Memory v.s. Non-continuous Memory

Continuous Memory has better performance and Non-continuous Memory has high resource utilization.

Several works [23], [41] use hugepage [42] in memory management to diminish the overhead from virtual address translation and protection metadata. They show that using hugepage in key-value stores can gain a 30% performance improvement. However, the usage of hugepage easily increases fragmentation and hence causes low memory utilization compared with non-continuous memory.

Another observation shows that remote sequential access has a better performance to surpass remote random access, which is similar to the random/sequential memory access asymmetry in local DRAM. Figure 2 displays the performance of remote memory access in both sequential and random pattern with the payload size growing from 1 Byte to 8192 Bytes [3]. We see that the sequential access pattern outperforms random memory access by a factor 2. The frequent cache misses in RNIC cache are the major factor contributing to the performance loss.

To address the trade-off between non-continuous and continuous memory allocation, we have implemented a solution that involves provisioning multiple hugepage sizes, as opposed to relying on a single huge size or no hugepage approach. We also redesigned the native memory allocator to minimize random access overhead.

4 *HybridSkipList+* DESIGN

In this section, we give an overview of *HybridSkipList+* (Section 4.1) and introduce the data structure operation algorithm with lock-based concurrent control (Section 4.2), our cache protocol (Section 4.3), a *Skiplist* friendly allocator (Section 4.4) and range operations (Section 4.5).

3. To assess random write, we choose a random address in the hugepage (1 GB) during testing.

4.1 HybridSkipList+ Overview

HybridSkipList+ is a skiplist that contains modified nodes. Each node contains the basic skiplist parameters: level, pointer to next node, value, as described in Section 2.2. But it also contains other parameters specific to our implementation. Figure 4 presents the format of the node in *HybridSkipList+*. The node holds a version number, used for concurrency control, and four auxiliary flags.

To handle concurrency control, we use retry lock [43]. Instead of preventing concurrent modifications of a data structure node, the core idea is to optimistically assume that the node has not been modified. After the completion of read operation, if there exists a modification (by checking the version number) in the data structure node, this read operation should be retried.

In the format of *HybridSkipList+*, a *version number* is in the tail of the node, to support the `update` method. This number can help to check the modification of its value during a read operation. It is set at an even value to start with. When executing the `update` method, the node is locked, and then it is modified immediately. After that, it increases the version number to an odd value, performs the update, and again increases version number back to an even value. The increment operation is atomic, and modification of value is followed by a memory barrier to flush the cache to the main memory. If the `RDMA Read` is interleaved with the local modification, it will see the odd value, indicating that this read operation must be retried.

Only when the version number is observed as even or checked to be the same as with the first observed value, then the value can be read successfully. Otherwise, it means a failed attempt. Each update to the value of a node will increment this number by two. This number can assist the read operation to identify the status of node: 1) a write operation is modifying the node; or 2) a modification has already been applied to this node while doing the read operation. More details about the version number are in the update operation section (4.2).

In addition to the necessary version number, level number, key, data, and pointers which points to successor nodes, each node contains of four auxiliary flags in the tail of the node. These flags can maintain the properties of *Skiplist*. `Lock` is used to prevent structural changes close to a node while it is being inserted or removed, and by blocking any access to a node until it has been inserted into or removed from all levels of the list. To prevent changes to the node's predecessors while modifying the current node, insert/remove operation will set `Lock` flag to indicate that the predecessors, validates that the locked predecessors still refer to their successors. `FullyLinked` shows whether a node has been linked in all its levels or not. Any read or write operation to a node is not allowed until it is fully linked. `Removed` shows whether a node can be deleted. When a node is marked, it is deleted logically.

These four attributes (version number, pointers, level number, and auxiliary fields) are all less than 8 Bytes and hence client can fetch them back via an `RDMA Read` operation. To improve performance, we apply selective signal [23], [44] to RDMA operations. Even without signal, the client can be aware that a read operation is completed by

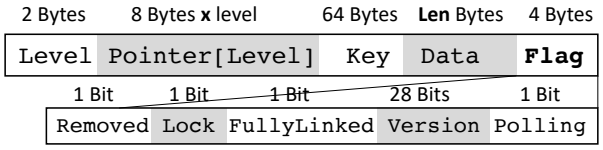

Fig. 4: The Node Structure in *HybridSkipList+*.

detecting that the tail `Polling` flag is changed to one (the `RDMA Write` operation is guaranteed to write the data in a sequential order [41], [45].)

Hybrid Communication Paradigm. As outlined in Section 3, *HybridSkipList+* employs a hybrid strategy for its operations, using a one-sided approach for reads and a two-sided approach for writes, ensuring data consistency with *Skiplist*'s inherent features. Further details on operation algorithms are in Section 4.2. To optimize server CPU usage and minimize client overload, some read operations can be performed using the two-sided approach based on current conditions. We introduce a dynamic tuning strategy for adjusting the proportion of two-sided read operations in Section 4.2.

Multiple Servers. To distribute *Skiplist* to multiple servers, we use the key-hashing partition approach to split *Skiplist* and assign partitions to the indicated server. Specifically, each server can hold multiple partitions. To keep load balancing, each partition can be migrated to another server dynamically if the workload is skewed [46]. Compared to the tree-like data structure, which requires complex node split/merge and spin, *Skiplist* is suitable for dividing into multiple partitions. Each partition will be an individual *Skiplist* and store the elements whose key is in a specific range. The mappings between partition and server are well known by both the server and client side. To maintain load balancing across servers with limited partitions, portions of data can be transferred between partitions by moving their head or tail segments [47].

4.2 Skiplist Basic Operations

We outline the implementation of five basic methods: Find, Contain, Insert, Update, and Remove. Find and Contain are designed for both one-sided and two-sided paradigms and can be executed by either client or server. The two-sided paradigm mirrors the traditional client-server RPC with three steps: client request, server execution, and client response. Initially, the client sends Insert/Update/Remove requests using a two-sided RDMA Send. Upon request arrival, the server performs local *Skiplist* operations and, if needed, sends a response back to the client via RDMA Send.

Find. The `Find` method is the core of all operations. As shown in Figure 5, setting the header node as the root, the client traverses the nodes in *Skiplist* via iterative `RDMA Reads` until the target node is found, or the loop is ended.

We aim to exploit the "Doorbell" technique [38] to batch multiple `RDMA Read` operations. Doorbell can reduce multiple CPU-generated MMIOs (from local memory to RNIC) to only one. However, these `RDMA Read` operations in the same batch should not rely on the value of each other. The client only needs to poll the last CQE in the batch once to determine the status of all requests. According to this

```
1  int find(Key key_, Node[] pre, Node[] suc){
2    int key=key_.hashCode(), lFound = -1;
3    Node pre=rdma_read(head);
4    for(int level=MAX_LEVEL to 0){
5      Node cur=rdma_read(pre.next[level]);
6      cur_key,suc=rdma_batch_read({cur->key,cur->
           next[level]});
7      while(cur_key<key) {
8        pre=cur; cur=suc;
9        cur_key,suc=rdma_batch_read({cur->key,cur->
             next[level]});
10     }
11     if(lFound==-1 && key==cur_key)
12       lFound=level;
13     pre[level]=pre; suc[level]=cur;
14   }
15   return lFound;
16 }
17 KeyValue *contain(uint64_t key) {
18   Node preds[MAX_LEVEL], succs[MAX_LEVEL];
19   int lFound=find(key, preds, succs);
20   if (lFound==-1) return nullptr;
21   Node node_found=succs[lFound];
22   is_removed, is_fully=rdma_batch_read({
         node_found->is_removed, node_found->
         is_fully});
23   if (!is_fully || is_removed)
24     return nullptr;
25   while(true){
26     int version=rdma_read(node_found);
27     if(version % 2 == 1) continue;
28     KeyValue *kv=rdma_read_node(node_found);
29     if(version == rdma_read(node_found))
30       return kv;
31   }
32 }
```

Fig. 5: Find and Contain methods.

rule, we use Doorbell batch at lines 6, 9 and 22. As an example, on line 6, the client can fetch back the key and successor' pointer in the same batch. `RDMA Read` is an atomical operation because all attributes (except value) are the size of a cacheline (64 bits), which is executed atomically in x86 architecture by exploiting `RDMA Atomic`. **Contain.** The `Contain` method reads the target key-value pair from *Skiplist*; if not found, it returns `null`. This method is wait-free. It calls the `Find` method to verify the existence of the key. If it finds a node with this key, it then checks the `FullyLinked` and `Removed` flags of the node via `RDMA Read`. If `FullyLinked` is true and `Removed` is false, the node is considered to have existed and the value can be returned to client. Before returning the value, it checks the version number of such a node. If the value of the version number cannot satisfy these two conditions: 1) even and 2) equal to the first observed value, the `Contain` method should retry from the beginning until it is valid.

Additionally, at line 26 in this algorithm, we primarily determine the status of the node via `RDMA Read` and then fetch the data as the result. However, if the node has a small data size, a better strategy is fetching the whole node first (line 28) and then check its status (line 29). When `update` operations are rare, this optimization can reduce network round trips significantly.

The `Insert`, `Update`, and `Remove` methods are two-sided implementations. They are only invoked by servers. The server side uses *gcc builtin atomic* library to guarantee

the critical data access is atomic.

Insert. The `Insert` method first calls the `Find` method to gain the predecessor nodes and successor nodes. In this case, the server side is responsible for processing the `Find` method, rather than relying on a one-sided operation to locate the node. This is because after applying the cache, the combination of one-sided and two-sided operation in the same method will give rise to consistency issues. In order to prevent any modifications to predecessors during the insertion of a node, this method sequentially locks the predecessors in ascending order. It then verifies that the predecessors still maintain their links to the successors. If the validation fails, the server releases the locks and retries from the beginning of the `Find` method. After acquiring the locks, the server initiates a new node and completes the pointer chasing by setting the `FullyLinked` flag as true.

Update. The `Update` method is implemented based on the `Find` method. If it locates the target node, it first locks the node. Then it increases the version number to an odd value, performs the update, and increases the version number to an even value. The key difference is that the update method only requires locking the found node because the update method does not change the structure of *Skiplist*.

Remove. Similar to `Insert` method, the server should first call the `Find` method to locate the "victim" node. If the node is found, it checks whether the node is ready to be removed, namely if the `FullyLinked` flag of the victim is true and the `Removed` flag of the victim is false. If the node can be removed, this method will remove it logically by setting the `Removed` flag from false to true. Then the physical deletion proceeds. The server locks the predecessors of the victim node (at all layers), validates that the predecessors' `Removed` flags are still false and link to the victim node. After validation, the server can splice out the victim node from the top layer to the bottom, maintaining the correctness property of *Skiplist*.

Dynamic Switch. One-sided Find/Contain (Read) methods require more RDMA operations than two-sided Insert/Update/Remove (Write) methods. In some cases, the Read method is dominant, thus the bandwidth of RDMA is saturated. To tackle this, we propose a mechanism for Read methods to switch between the two communication paradigms dynamically. The server side monitors the inbound network traffic by reading several hardware counters in the **/proc** pseudo-file system for RDMA operations. If it exceeds the *maximum throughput* threshold (obtained by using profiling tools), the server informs several randomly selected clients to turn their Read methods to the two-sided RPC mode. It is accomplished by piggybacking the notifications (one-bit flag) in the response of the two-sided Write methods. This is inspired by the implementation of FaRM [41].

Correctness of Concurrency. *Skiplist* are naturally lock-free, and the only work to be done is to carefully choose the order of operations [32]. Here we prove it is correct under RDMA one-sided operations. Two-sided operations are similar for correctness.

As shown in Figure 6, we assume a Find method and an Insert method are processed at the same time. The server side first creates the newly allocated node and sets the pointers in the new node accordingly (①). After that, the

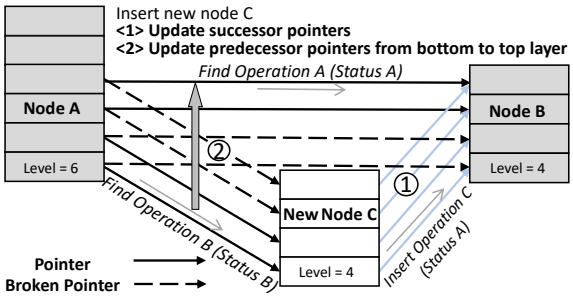

Fig. 6: Concurrency Control in Skiplist with one-sided RDMA operations.

previous pointers will be updated from the bottom to the top (②). Meanwhile, if the client indexes Node B via `RDMA Read`, it can still get the consistent views of *Skiplist* in such a scenario through different views for different readers, thereby no lock is required [48].

Similarly, if the Delete method is traversing the pointers from bottom to top, the Find method can still find a consistent view, which is before the linearizable point of the completion of the Deletion method. Finally, *HybridSkipList+* is eventually shown to be consistent.

4.3 Client-sided Cache

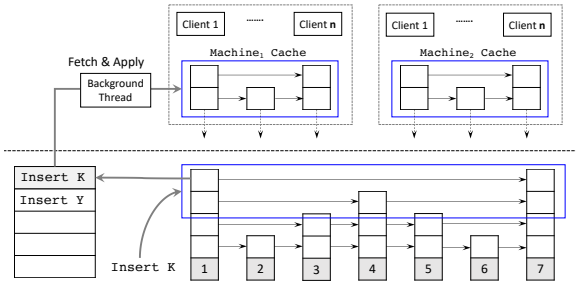

Fig. 7: The architecture of client-sided cache in *HybridSkipList+*.

Using one-sided operation to execute read operations incurs a large number of network round-trips. To remedy this, as shown in Figure 7, the client employs a cache to buffer data of the high layer whose level is larger than a given level N. These layers will be organized as a "sketch" which is a subset of a raw *Skiplist*. Each read operation first locates the correct node by accessing the nodes in the client-side cache. After the pointer is located at the level N, it changes to read data from the server via `RDMA Read` directly. Clients in the same machine share their cache to reduce the overhead of maintaining cache coherence.

To mitigate the overhead of synchronizing data modification, inspired by directory-based protocol, we present a pull-based synchronization primitive. The server appends the record of each write operation whose level is larger than N to the shared log area, as shown in in Figure 7. Each record is organized as a tuple {operation type, parameters}. This procedure is processed immediately after the end of each write operation (or remove or update), which means the data is transparent to clients. This shared log will ensure

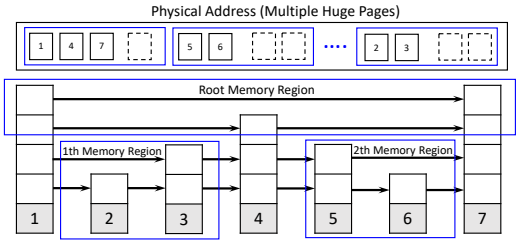

Fig. 8: Layout of Semi-Continuous Memory Allocator.

a deterministic order of this operation at the higher levels. Since the records only include write operations which is a small part of total operations in practical applications, the overhead of recording is tolerable. Meanwhile, the client fetches back the shared modification logs via `RDMA Read`. After that, it updates the data in the cache using the fetched logs, without the intervention of the server. Because the data in the cache is up-to-date, each read operation in client side will always get a consistent view of *Skiplist*.

Overall, *HybridSkipList+* provides eventual consistency, i.e., there is no guarantee that each read operation can immediately access any newly applied write operation, but eventually will. Eventual consistency is common in distributed data structures.

4.4 Semi-Continuous Memory Allocator

As noted in Section 3, continuous memory regions result in low utilization, whereas noncontinuous regions lead to poor performance. Therefore, we balance the trade-off between a single large large-page and massive small-memory regions by adopting moderate granularity for RDMA-enabled memory registration.

The allocator registers several moderate size memory regions and each one is a fixed size hugepage. When a data structure operation acquires memory, the resources are allocated from the indicated memory region *according to the key's value and node's level*. If the node's level is larger than L_r (e.g. key *1, 4, 7* in Figure 8), the node will layout in the "root" memory region. Otherwise, this node exists in the $(key/interval_size)$th memory region ((e.g. *0th MR: key 2, key 3 and 1th MR: key 5, key 6* in Figure 8)). If one of the memory regions is full, the allocator will register additional memory from the operating system After using moderate-granularity memory registration, since adjacent nodes of *Skiplist* are always laid in the memory region with better locality, frequent cache misses in RNIC cache can be mitigated.

As an example, when trying to find the node with key 6, there are four random memory accesses (key 1 $\rightarrow$ key 4 $\rightarrow$ key 5 $\rightarrow$ key 6). However, with the semi-continuous memory allocator, only two random memory accesses are required since key *1, 4* and key *5, 6* are all in the same memory region.

4.5 Skiplist Range Operations

Range query is an important feature in RDBMS. It can get these sorted elements in the range of a_l and a_r. A query consists of computing some functions (e.g., minimum, filter) [49], [50].

```
1  //Client-sided
2  KeyValue **RangeQuery(Key lkey, Key rkey, Func *
       fn){
3    KeyValue* lnode=contain(lkey); // one-sided
4    RightValue* rnode=contain(rkey);
5    KeyValue **res;
6    send_request(lnode, rnode, fn); // two-sided
7    recv_response(&kvs); // one-sided
8    return res;
9  }
10 //Server-sided
11 void handler() {
12   KeyValue **range, *cur, *next;
13   KeyValue *lnode, rnode, Func *fn = recv_request
         (); // gain the request
14   while(cur != rnode) {
15     retry:
16     next = cur->next;
17     if(isLocked()) goto retry;
18     add_to_arrary(range, next);
19     cur = next;
20   }
21   KeyValue *res = execute(range, fn);
22   //  execute the user-specific query
23   write_to_buffer(res);
24 }
```

Fig. 9: Range query method.

B+Tree/B-Tree naturally supports range queries. This is because most keys are stored in the same leaf node, thus scanning these keys in leaf nodes causes fewer pointer chasing. Assume each leaf has T keys on average, the network round trips should be $2 * log(n) + \frac{|a_l, a_r|}{T}$ (where $|a_l, a_r|$ is the distance between left node (a_l) and right node (a_r)).

In *Skiplist*, the address of keys are not continuous, fetching each key triggers a pointer chasing, which means an expensive network round trip. The total network round trips are $2 * log(n) + |a_l, a_r|$. In a case of large range query, pointer chasing is the major part of the overhead.

Similar to the one/two-sided combination approach of `Insert` method, *HybridSkipList+* employs the idea of a hybrid paradigm. Pseudo-code in Figure 9 demonstrates how to leverage the advantages of both one-sided and two-sided operations. The client uses two $Find$ methods to locate the position of left node and right node (line 3, 4). Then, by invoking a RPC, it delegates to the server to traverse the target nodes between left node and right node and at the same time, executes the user-specific query (line 6). After the traversal is completed by the server side, the client can fetch back the results which contain the array of target key-value pairs (line 7). Accordingly, the total network round trips reduce to $2 * log(n) + 2$.

To handle the consistency issue raised by write operations, the server checks the `Lock` flag and waits until the write operation is finished during the processing of traversal. To mitigate the CPU stress in the server side, we use the approach of RFP RPC [7], [44] to implement the response phase of RPC. The server side provides a circular buffer for client to fetch back the response via `RDMA Read`. Due to the use of one-sided operation, the server side is passive and is not involved in the network processing.

5 EVALUATION

5.1 Evaluation Setting

We use an eight-machine cluster based on InfiniBand protocol for the evaluation. Each machine is equipped with dual 4-core CPUs (Intel $Xeon$ E5-2640 v2, 2.0 GHz), 96 GB memory space, and a Mellanox ConnectX-3 InfiniBand NIC (MT27500, 40 Gbps). The CPU has two sockets and the memory is equally allocated to each socket. An 18-port Mellanox InfiniScale-IV switch connects all of these machines. Because RNIC only binds to socket 1, we use one of the two sockets (i.e., four cores) to simplify our experiments. The machines run MLNX-OFED-LINUX-4.2-1.0.0 driver provided by Mellanox for Ubuntu 14.04 [51].

In the following evaluations, based on RDMA verbs, we implement *HybridSkipList+* with around $5k$ C++11 SLOC to verify the system design choices. The codes were compiled using GCC 4.8.5. We use YCSB [52] to generate 64 million key-value pairs offline for the experiment, allowing the client to create requests with varying write ratios. For simplicity, keys and values are 8 bytes and 64 bytes, respectively. Additionally, we implement *pure one-sided* Skiplist and *pure two-sided* Skiplist as baselines to assess the hybrid approaches. In one-sided *Skiplist*, `RDMA Atomic` operations are used for locking. The Skiplist's probability p is set to 0.5, and results are averaged over ten runs.

5.2 Basic Performance

We set the workload as 10% write operations, cache proportion as $1/32$ and data size in key-value item as 4096 Bytes. The evaluation results are the average of ten runs. We measure the performance of *HybridSkipList+* with all optimizations, and it can perform 1.48×∼4.41× / 0.52×∼3.45× higher throughput than one-sided/two-sided *Skiplist* respectively. *HybridSkipList+* performs a lower throughput than two-sided *Skiplist* when the number of clients is less than six because two-sided *Skiplist* is sensitive to contention. When the client number is 16, *HybridSkipList+* can reach a peak throughput 237 KOPS. With the same setting, the pure `RDMA Write` test performs 780 KOPS (390 KOPS for pingpong). The throughput achieved by *HybridSkipList+* is a significant improvement, as it reaches 61% of the upper bound limit of 390 KOPS for the pingpong scenario of `RDMA Write`. In conclusion, *HybridSkipList+* can make full use of the RDMA bandwidth.

To evaluate the design choices in Section 3, we measure performance gains through step-by-step optimization in Figure 10. After applying the semi-continuous RDMA-enabled memory allocator, *HybridSkipList+* performs at nearly 40% higher throughput than original implementation with one-sided operations. Including caching optimization also brings more throughput improvements (191%). Furthermore, a 52% throughput improvement can be gained with the hybrid communication paradigm.

Figure 11 shows the performance of range query, and we compare the implementation of range query in *HybridSkipList+* to the original one-sided implementations. Here we define the query's range as the number of nodes between left node and right nodes and use the average as the user-specific function. *HybridSkipList+* has a higher throughput (2.4× ∼4.9×) when the range of each query is more than

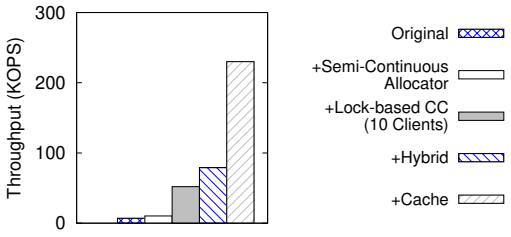

Fig. 10: The cumulative performance gain when applying different optimizations.

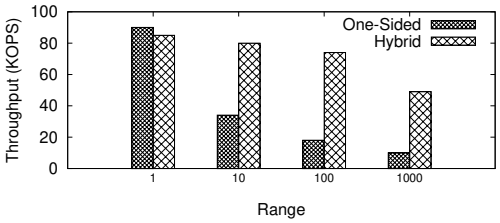

Fig. 11: The performance of range query.

10, due to the usage of hybrid RDMA approach. The outlier is range = 1. The reason is that at least two network round trips are required using RPC, but this number is only one while fetching results via `RDMA Read`.

5.3 Scalability

We measure the scalability of *HybridSkipList+* using multiple clients. The results are displayed in Figure 12. The two-sided strategy exhibits poor scalability: after reaching a client number of 4, its aggregate throughput experiences a significant decline. Specifically, when there are ten clients, the average throughput per client is reduced by 86.2% compared to when there is only one client. Furthermore, as we increase the number of clients beyond 10, the throughput continues to decrease due to the two-sided strategy being constrained by server-side CPU limitations.

However, the one-sided strategy performs the lowest throughput of all four strategies since it requires more network round trips. In the case of *HybridSkipList+* (No cache), the aggregate throughput increases consistently from one client to 24 clients, and the degradation of the average throughput is only about 38.5 For the *HybridSkipList+* scenario, when the number of clients exceeds 16, the server side will experience IOPS binding, resulting in a slight decline in performance. It still performs better than one-sided *Skiplist*/two-sided *Skiplist*. We can conclude that *HybridSkipList+* scales well with the increasing number of clients.

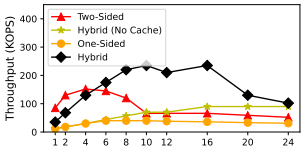

Fig. 12: The aggregate throughput with increasing the number of clients.

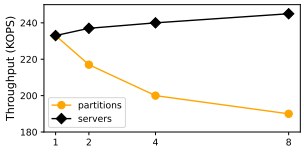

Fig. 13: The aggregate throughput with partitions increasing.

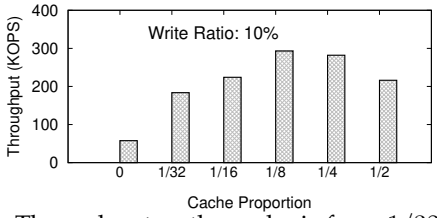

Fig. 14: Throughput as the cache is from $1/32$ to $1/2$.

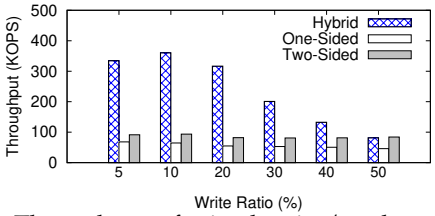

Fig. 15: Throughput of mixed write/read workloads.

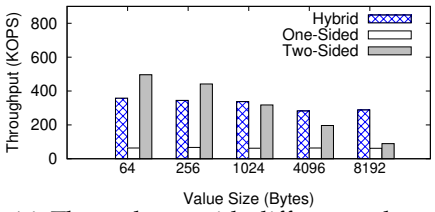

Fig. 16: Throughput with different value sizes.

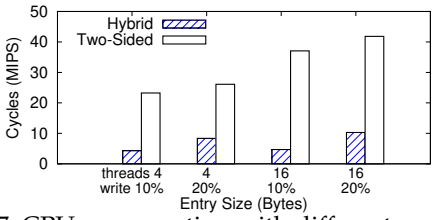

Fig. 17: CPU consumption with different workload.

As shown in Figure 13, we measure the performance of distributing *HybridSkipList+* to multiple partitions or multiple servers (physical machines). We set the client number as 10. There is about a 7% performance improvement when we distribute the same *Skiplist* from one server to eight servers (each server has a partition), which shows good scalability. However, when we split *Skiplist* from one partition to eight partitions but stored on the same server, the performance drop is significant (about 17%) but is still tolerable. This is because each partition should establish connections (QP) with all clients, and thus massive connections incur performance drop in RC mode as we mentioned in Section 2.1.

5.4 The Effect of Cache

We measure the benefit of caching with different cache sizes in the client side and the results are shown in Figure 14. When there are six clients and each one maintains $1/32$ of *Skiplist* as cache in the client machine, *HybridSkipList+* can reach 184 KOPS, which is 3.2 $\times$ higher than the case with no cache. Overall, by increasing cache sizes, the throughput increases accordingly but becomes slower gradually due to the overhead from the synchronization primitives.

The peak throughput is 293 KOPS when caching size is $1/8$ of the whole *Skiplist*, and the throughput declines after that. When the caching ratio is $1/2$, the throughput decrease to 74% of the peak throughput. The reason is that when caching a larger proportion of *Skiplist*, the overhead of cache coherence protocol is higher than the benefits from caching.

5.5 Different Workloads

To maintain fairness in comparison, we use 8 clients and $1/8$ caching proportion to achieve a peak throughput for one-sided *Skiplist*.

Figure 15 illustrates the performance of different read-/write ratios in all clients. For simplicity, we treat the insert method as a representation of the write operation and the contain method as a representation of read operation. With fewer read operations, the throughput decreases 74% from 5% to 50% write ratio. The reason is that write operations will incur heavy write contention to saturate the server's CPU, and bring more overhead even when using a hybrid strategy. Figure 16 shows the impact of data size. *HybridSkipList+* has a higher throughput than one-sided *Skiplist* when data size is more than 256 bytes. The performance of *HybridSkipList+* decreases slower than two-sided when data size increases from 64 Bytes to 4096 Bytes. The reason is that, in the two-sided approach with increasing value size, the server would respond to clients with larger payload size packets. This leads to the performance being easily limited by the server-side RPC operations.

5.6 CPU Utilization

In Figure 17, we evaluate the CPU utilization of *HybridSkipList+* to show the benefits of the hybrid RDMA paradigm. *HybridSkipList+* performs at a lower CPU utilization compared to a two-sided *Skiplist*. Even with eight clients, *HybridSkipList+* still maintains a very small CPU consumption and each server thread is 4.70 MIPS (million instructions per second) on average. The reason is that some of the data structure operations are executed by the client-side without any involvement of the server side. In contrast, two-sided *Skiplist* can reach 37.06 MIPS (nearly 7.89$\times$ higher than *HybridSkipList+*). When the write ratio is two-fold from 10% to 20%, the CPU consumption also increases linearly due to the heavy contention.

6 Related Works

SkipList. There are several works about lock-free *Skiplist*. Fomitchev et al. [32] present a lock-free *Skiplist* with back-links to visit the deleted node while concurrent executing. Concurrent SkipListMap [33] in Java SE 6 is the most popular *Skiplist* implementation, which is lock-free. Maurice et al. [34] present a lock-based algorithm and adopt an optimistic lock and lazy removing node for the delete operation. There are several works on how to optimize the specific operations of *Skiplist*. For instance, Alam et al. [53] focus on how to design and optimize Range Queries of *Skiplist* in a distributed scenario. *HybridSkipList+* is the first work to redesign distributed *Skiplist* for RDMA operations.

RDMA-based data structure. Many studies discard the traditional RPC, and attempt to leverage the one-sided operations to totally bypass server CPU. Pilaf [9] allows its clients to directly read data from the server's memory through RDMA-read for Get requests. The clients use CRC64 to check for data inconsistencies caused by potential

races with the Put operations on the server. FaRM [41] is a distributed memory platform and implements Hopscotch Hashing which adopts one-sided operation. Based on both one-sided and two-sides RDMA operations, Catfish [54] is a R-tree index data structure which leverages RDMA Read for R-tree traversal, and Cell [36] implements a B-tree index data structure and prefetches one B-Tree node at once. However, they do not have an efficient cache mechanism and there is no work on how to leverage RDMA to implement data structures with the client-side cache.

Hybrid RDMA paradigms. Based on OCC, DrTM-H [16] evaluates various phases (execution, validation, logging, commit) of transactions with both one-sided and two-sided implementations. It shows that neither approach is the best design choice and that a hybrid strategy can leverage the benefits from both one-sided and two-sided. As an orthogonal work, RCC [55] studies six concurrency control protocols using two-sided or one-sided primitives to understand performance under different transaction workloads. *HybridSkipList+* attempts to take advantage of the hybrid strategy for *Skiplist*, and it is orthogonal to these works.

Compared to HybridSkipList [1], *HybridSkipList+* presents a client-sided coherent cache with pull-based synchronization. At the same time, it optimizes range query to better support RDBMS and proposes a semi-continuous memory allocator to improve the raw performance.

7 Conclusion

In this paper, we applied the idea that "Hybrid is Better" to the data structure design principle. Based on six design choices, we presented *HybridSkipList+*, which combines one-sided and two-sided operations dynamically. Our method can support both basic data structure operation and range query with a hybrid RDMA paradigm.

To enable the caching of frequently accessed data, we rethought the cache coherence protocol under the RDMA scenario and proposed a pull-based protocol. In addition, we proposed a semi-continuous memory allocator to improve the locality of each RDMA access. The results show that *HybridSkipList+* easily benefits from these main design choices. While our work is based upon Skiplist, we hope these general methods, protocols and communication paradigms provide a good guidance for developers to implement a library by generalizing the abstraction to other various distributed data structures types.

References

[1] T. Ma, D. He, and N. Liu, "HybridSkipList: A Case Study of Designing Distributed Data Structure with Hybrid RDMA," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2021, pp. 68–73.

[2] Q. Liu and P. Varman, "Telepathy: A lightweight silent data access protocol for nvram+ rdma enabled distributed storage," *IEEE Transactions on Computers*, vol. 72, no. 3, pp. 839–852, 2022.

[3] J. Shu, Y. Chen, Q. Wang, B. Zhu, J. Li, and Y. Lu, "Th-dpms: Design and implementation of an rdma-enabled distributed persistent memory storage system," *ACM Transactions on Storage (TOS)*, vol. 16, no. 4, pp. 1–31, 2020.

[4] W. Bai, S. S. Abdeen, A. Agrawal, K. K. Attre, P. Bahl, A. Bhagat, G. Bhaskara, T. Brokhman, L. Cao, A. Cheema *et al.*, "Empowering azure storage with rdma," in *USENIX NSDI*, 2023, pp. 49–67.

[5] S. Zheng, J. Wang, D. Xue, J. Shu, and L. Huang, "Hydra: A decentralized file system for persistent memory and rdma networks," *IEEE TPDS*, vol. 33, no. 12, pp. 4192–4206, 2022.

[6] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble *et al.*, "Snap: A microkernel approach to host networking," in *ACM SOSP*, 2019, pp. 399–413.

[7] M. Su, M. Zhang, K. Chen, Z. Guo, and Y. Wu, "Rfp: When rpc is faster than server-bypass with rdma," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 1–15.

[8] Y. Chen, Y. Lu, and J. Shu, "Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing," in *14th EuroSys*. ACM, 2019, p. 19.

[9] C. Mitchell, Y. Geng, and J. Li, "Using one-sided RDMA reads to build a fast, CPU-efficient key-value store." in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2013, pp. 103–114.

[10] Y. Wang, X. Meng, L. Zhang, and J. Tan, "C-Hint: An effective and reliable cache management for RDMA-accelerated key-value stores," in *ACM SoCC*. ACM, 2014, pp. 1–13.

[11] B. Cassell, T. Szepesi, B. Wong, T. Brecht, J. Ma, and X. Liu, "Nessie: A decoupled, client-driven key-value store using rdma," *IEEE TPDS*, vol. 28, no. 12, pp. 3537–3552, 2017.

[12] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," in *25th SOSP*. ACM, 2015, pp. 87–104.

[13] P. Li, Y. Hua, P. Zuo, Z. Chen, and J. Sheng, "Rolex: A scalable rdma-oriented learned key-value store for disaggregated memory systems," in *USENIX FAST*, 2023, pp. 99–114.

[14] A. Kalia, M. Kaminsky, and D. G. Andersen, "Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs," in *12th USENIX OSDI*, 2016, pp. 185–201.

[15] S. Sun, R. Zhang, M. Yan, and J. Wu, "Skv: A smartnic-offloaded distributed key-value store," in *IEEE CLUSTER*, 2022, pp. 1–11.

[16] X. Wei, Z. Dong, R. Chen, and H. Chen, "Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!" in *13th USENIX OSDI*, 2018.

[17] Z. Xie, Q. Cai, H. Jagadish, B. C. Ooi, and W.-F. Wong, "Pi: a parallel in-memory skip list based index," *arXiv preprint arXiv:1601.00159*, 2016.

[18] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "RDMA over commodity Ethernet at scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 202–215.

[19] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: distributed transactions with consistency, availability, and performance," in *ACM SOSP*, 2015, pp. 54–70.

[20] T. Ziegler, J. Nelson-Slivon, V. Leis, and C. Binnig, "Design guidelines for correct, efficient, and scalable synchronization using one-sided rdma," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–26, 2023.

[21] S. Ma, T. Ma, K. Chen, and Y. Wu, "A survey of storage systems in the rdma era," *IEEE TPDS*, 2022.

[22] L. Ou, X. He, and J. Han, "An efficient design for fast memory registration in rdma," *Journal of Network and Computer Applications*, vol. 32, no. 3, pp. 642–651, 2009.

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

[24] S.-Y. Tsai and Y. Zhang, "Lite kernel rdma support for datacenter applications," in *ACM SOSP*, 2017, pp. 306–324.

[25] A. Dragojevic, D. Narayanan, and M. Castro, "Rdma reads: To use or not to use?" *IEEE Data Eng. Bull.*, vol. 40, no. 1, pp. 3–14, 2017.

[26] E. Zamanian, C. Binnig, T. Harris, and T. Kraska, "The end of a myth: Distributed transactions can scale," *Proceedings of the VLDB Endowment*, vol. 10, no. 6, pp. 685–696, 2017.

[27] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter rpcs can be general and fast," in *USENIX NSDI*. Boston, MA: USENIX Association, 2019, pp. 1–16.

[28] Z. Wang, T. Ma, L. Kong, Z. Wen, J. Li, Z. Song, Y. Lu, G. Chen, and W. Cao, "Zero overhead monitoring for cloud-native infrastructure using rdma," in *USENIX ATC*, 2022.

[29] W. Pugh, "A skip list cookbook," Tech. Rep., 1998.

[30] J. Zhang, S. Wu, Z. Tan, G. Chen, Z. Cheng, W. Cao, Y. Gao, and X. Feng, "S3: A scalable in-memory skip-list index for key-value store," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2183–2194, 2019.

[31] W. Kim, C. Park, D. Kim, H. Park, Y.-r. Choi, A. Sussman, and B. Nam, "Listdb: Union of write-ahead logs and persistent skiplists for incremental checkpointing on persistent memory," in *16th USENIX Symposium on OSDI*, 2022, pp. 161–177.

[32] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," in *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. ACM, 2004, pp. 50–59.

[33] D. Lea, "Concurrent skip list map," https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html, 2017.

[34] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A simple optimistic skiplist algorithm," in *International Colloquium on Structural Information and Communication Complexity*. Springer, 2007, pp. 124–138.

[35] M. F. Kaashoek and D. R. Karger, "Koorde: A simple degree-optimal distributed hash table," in *International Workshop on Peer-to-Peer Systems*. Springer, 2003, pp. 98–107.

[36] C. Mitchell, K. Montgomery, L. Nelson, S. Sen, and J. Li, "Balancing cpu and network in the cell distributed b-tree store," in *USENIX ATC*, 2016, pp. 451–464.

[37] T. Crain, V. Gramoli, and M. Raynal, "No hot spot non-blocking skip list," in *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 2013, pp. 196–205.

[38] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 295–306, 2015.

[39] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott, "Cashmere-2l: Software coherent shared memory on a clustered remote-write network," in *SOSP*, vol. 97, 1997, pp. 170–183.

[40] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," in *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*. IEEE, 1990, pp. 148–159.

[41] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *11th USENIX NSDI*, 2014, pp. 401–414.

[42] "Huge page concepts in linux." [Online]. Available: https://www.linuxfoundation.org/webinars/huge-page-concepts-in-linux

[43] T. Wang and H. Kimura, "Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores," *Proceedings of the VLDB Endowment*, vol. 10, no. 2, pp. 49–60, 2016.

[44] Y. Wu, T. Ma, M. Su, M. Zhang, K. Chen, and Z. Guo, "RF-RPC: Remote fetching RPC paradigm for RDMA-enabled network," *IEEE TPDS*, vol. 30, no. 7, pp. 1657–1671, 2018.

[45] J. Liu, J. Wu, and D. K. Panda, "High performance RDMA-based MPI implementation over InfiniBand," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[46] A. F. Trajano and M. P. Fernandez, "Two-phase load balancing of in-memory key-value storages using network functions virtualization (nfv)," *Journal of Network and Computer Applications*, vol. 69, pp. 1–13, 2016.

[47] G. Graefe and H. Kuno, "Modern b-tree techniques," in *2011 IEEE 27th International Conference on ICDE*. IEEE, 2011, pp. 1370–1373.

[48] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A provably correct scalable concurrent skip list," in *Conference On Principles of Distributed Systems (OPODIS)*, 2006.

[49] D. Wu, C. H. Cheong, and M. H. Wong, "Supporting asynchronous update for distributed data cubes," *Journal of network and computer applications*, vol. 32, no. 4, pp. 889–900, 2009.

[50] S. Kargar and L. Mohammad-Khanli, "Fractal: An advanced multidimensional range query lookup protocol on nested rings for distributed systems," *Journal of Network and Computer Applications*, vol. 87, pp. 147–168, 2017.

[51] Mellanox. (2016, July) The OFED homepage. https://www.openfabrics.org/.

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

[53] S. Alam, H. Kamal, and A. Wagner, "A scalable distributed skip list for range queries," in *Proceedings of the 23rd international symposium on HPDC*. ACM, 2014, pp. 315–318.

[54] M. Xiao, H. Wang, L. Geng, R. Lee, and X. Zhang, "Catfish: Adaptive RDMA-enabled R-Tree for Low Latency and High Throughput," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 164–175.

[55] C. Wang and X. Qian, "Rdma-enabled concurrency control protocols for transactions in the cloud era," *IEEE TCC*, 2021.

Yilei Lu Yilei Lu is a Ph.D. student in the Department of Computer Science at Tsinghua University. His research focuses on AI infrastructure and Large Language Models with a particular emphasis on industry practices.

Teng Ma received the PhD degree in computer science and technology from Tsinghua University, China in 2021. His research interests include distributed database, Remote Direct Memory Access (RDMA), and key-value store systems. He received his B.E. degree from University of Science and Technology Beijing, China, in 2016. Now he is a posdoc in Alibaba Group.

Dongbiao He received the PhD degree in computer science and technology from Tsinghua University, Beijing, China in 2019. His research interests include networking and edge computing systems. He received his B.E. degree from Jilin University, China, in 2013. He is an associate researcher at the Computer Network Information Center of the Chinese Academy of Sciences.

Xian Yu is currently an associate researcher at the Computer Network Information Center of the Chinese Academy of Sciences. He received the Ph.D. degree in Institute of Computing Technology, Chinese Academy of Sciences, China, 2020. He received the B.E. degree in Hunan University, China, 2014. His research interests include SDN/NFV, satellite internet, and cloud computing networks.

Zhe Wang is currently a Ph.D student in Department of Computer Science in Shanghai Jiao Tong University. He received his bachelor degree in Shanghai University in 2017. His research interests include wireless communication, data center networks and cloud computing.

Cedric Westphal is a principal research architect with Futurewei working on future network architectures for both wired and wireless networks. His current focus is on next generation networks. He is an adjunct assistant professor with the University of California, Santa Cruz. He is the recipient of multiple awards, including best paper at IEEE ICC and of the TCIIN 2018 Technical Achievement Award to *recognize a lifelong set of outstanding technical contributions in the area of information infrastructure and networking.*

Linghe Kong is a research professor in the Department of Computer Science and Engineering at Shanghai Jiao Tong University. Previously, he was a postdoctoral researcher at Columbia University, McGill University, and Singapore University of Technology and Design. He earned his Ph.D. in Computer Science from Shanghai Jiao Tong University in 2012, a Master's in Tele in 2007, and a B. Eng. in Automation from Xidian University in 2005. His research interests include satellite networking, edge and cloud computing, and the Internet of Things.