A Memory-Disaggregated Radix Tree

XUCHUAN LUO, School of Computer Science, Fudan University, Shanghai, China
PENGFEI ZUO, Huawei Cloud, Huawei Technologies Co Ltd, Shenzhen, China
JIACHENG SHEN, The Chinese University of Hong Kong, Hong Kong, China
JIAZHEN GU, The Chinese University of Hong Kong, Hong Kong, China
XIN WANG, School of Computer Science, Fudan University, Shanghai, China and Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China
MICHAEL LYU, The Chinese University of Hong Kong, Hong Kong, China
YANGFAN ZHOU, School of Computer Science, Fudan University, Shanghai, China and Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China

Disaggregated memory (DM) is an increasingly prevalent architecture with high resource utilization. It separates computing and memory resources into two pools and interconnects them with fast networks. Existing range indexes on DM are based on B+ trees, which suffer from large inherent read and write amplifications. The read and write amplifications rapidly saturate the network bandwidth, resulting in low request throughput and high access latency of B+ trees on DM.

In this article, we propose that the radix tree is more suitable for DM than the B+ tree due to smaller read and write amplifications. However, constructing a radix tree on DM is challenging due to the costly lock-based concurrency control, the bounded memory-side IOPS, and the complicated computing-side cache validation. To address these challenges, we design **SMART**, the first radix tree for disaggregated memory with high performance. Specifically, we leverage (1) a *hybrid concurrency control* scheme including lock-free internal nodes and fine-grained lock-based leaf nodes to reduce lock overhead, (2) a computing-side *read-delegation and write-combining* technique to break through the IOPS upper bound by reducing redundant I/Os, and (3) a simple yet effective *reverse check* mechanism for computing-side cache validation. Experimental results show that SMART achieves 6.1× higher throughput under typical write-intensive workloads and 2.8× higher throughput under read-only workloads in YCSB benchmarks, compared with state-of-the-art B+ trees on DM.

CCS Concepts: • **Information systems** → **Distributed storage**; **Data structures;**

Additional Key Words and Phrases: Disaggregated memory, remote direct memory access, radix tree

The preliminary version appears in *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23)*, pages 553-571, as "SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory."
This work is supported by the National Natural Science Foundation of China (Project No. 623B2026) and the Open Fund of PDL (Project No.WDZC20245250106).
Authors' Contact Information: Xuchuan Luo, School of Computer Science, Fudan University, Shanghai, China; e-mail: xcluo23@m.fudan.edu.cn; Pengfei Zuo, Huawei Cloud, Huawei Technologies Co Ltd, Shenzhen, China; e-mail: pfzuo.cs@gmail.com; Jiacheng Shen, The Chinese University of Hong Kong, Hong Kong, China; e-mail: jcshen@cse.cuhk.edu.hk; Jiazhen Gu, The Chinese University of Hong Kong, Hong Kong, China; e-mail: jiazhengu@cuhk.edu.hk; Xin Wang, School of Computer Science, Fudan University, Shanghai, China and Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China; e-mail: xinw@fudan.edu.cn; Michael Lyu, The Chinese University of Hong Kong, Hong Kong, China; e-mail: lyu@cse.cuhk.edu.hk; Yangfan Zhou, School of Computer Science, Fudan University, Shanghai, China and Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China; e-mail: zyf@fudan.edu.cn.

ACM 1553-3077/2024/06-ART15
https://doi.org/10.1145/3664289

ACM Reference Format:
Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael Lyu, and Yangfan Zhou. 2024. A Memory-Disaggregated Radix Tree. *ACM Trans. Storage* 20, 3, Article 15 (June 2024), 41 pages. https://doi.org/10.1145/3664289

1 INTRODUCTION

Distributed range indexes are fundamental building blocks of many applications, e.g., databases and key-value stores, to conduct range queries [1, 26, 65, 69, 72]. To improve resource utilization, many new proposals adopt the **disaggregated memory (DM)** architecture [65, 72]. DM can decouple computing and memory resources into two elastic resource pools (i.e., computing pool and memory pool) interconnected with high-speed networks, e.g., **remote direct memory access (RDMA)** connections [3, 9, 19, 22, 25, 32, 57]. In this way, a DM range indexing system can utilize resources more efficiently.

Current DM index systems [65, 72] use B+ tree to build range indexes, following the idea generally adopted in the monolithic server solutions. However, B+ trees can bring severe read and write amplification issues on DM. Specifically, when reading or writing a key-value item in a B+ tree, one should search the tree by traversing many nodes that contain many irrelevant pivot keys and pointers. This inevitably amplifies the network bandwidth consumption. As such network bandwidth is generally the bottleneck of the DM architecture [28], the amplified bandwidth consumption incurred by B+ trees exacerbates the bottleneck. This issue will lead to low overall throughput and high access latency. Our experimental study shows that it can dramatically degrade the throughput of Sherman [65], the state-of-the-art B+ tree index on DM. The throughput is 10.8× lower than the theoretical bound of **RDMA network interface cards (RNICs)** under the YCSB workloads [10].

In this article, we propose that radix tree is a more suitable tree index structure for DM. Compared with B+ trees, radix trees have smaller read and write amplifications, since they do not store the entire keys in internal nodes. Moreover, the state-of-the-art radix tree design, i.e., ART [38], further reduces read and write amplifications with an adaptive internal node design. However, several challenges should be addressed before radix trees become a high-performance, practical indexing solution for DM.

(1) Lock-based concurrency control is expensive. Remote lock operations are expensive on DM. However, the existing ART design adopts a lock-based algorithm for concurrency control [39], which contains many remote lock operations, worsening the write performance. In addition, computing-side caches are required on DM to reduce operation latency. The traditional **read-copy-update (RCU)** scheme for radix trees causes frequent changes in the addresses of cached nodes, leading to cache thrashing.

(2) Redundant I/Os deteriorate the throughput. RNICs in the memory pool of DM have bounded IOPS (I/O per second) [62]. However, radix trees have multiple small-sized read and write operations when traversing and modifying the tree index. Many of these read and write operations are redundant when multiple clients on the same compute node concurrently traverse the tree. These redundant I/Os on DM waste the limited IOPS of RNICs and thus decrease the peak throughput of radix trees.

(3) The complicated computing-side cache validation. Tree indexes on DM typically adopt computing-side caches to reduce access latency [68]. However, the structural features of radix trees (e.g., path compression) incur many address changes and metadata changes in radix tree nodes. These changes add more cache invalidation situations and thus complicate the cache design.

To address the above challenges, we propose **SMART**, a di**S**aggregated-me**M**ory-friendly **A**daptive **R**adix **T**ree. First, for better concurrency control, we present a *hybrid ART concurrency*

control scheme with a lock-free internal node design and a lock-based leaf node design. The lock-free internal node design avoids the expensive coarse-grained lock-based concurrency control of the traditional ART. The lock-based leaf node adopts an in-place update scheme, which avoids frequently changing leaf node addresses and prevents computing-side cache thrashing. Second, for an IOPS breakthrough, we propose a ***read-delegation and write-combining (RDWC)*** technique to reduce computing-side redundant I/Os. Third, for cache validation, we co-design SMART with an *ART cache*, including a reverse check mechanism to handle new cache invalidation situations of ART.

We implement SMART from scratch and evaluate it using the YCSB benchmark [10]. Compared with Sherman [65], the state-of-the-art B+-tree-based range index on DM, SMART achieves up to 6.1× higher throughput and 1.4× lower latency for typical write-intensive workloads and 2.8× higher throughput with similar latency for read-only workloads. The code of SMART is available at https://github.com/dmemsys/SMART.

In summary, this article makes the following contributions:

— We propose that ART is a better tree index on DM based on theoretical analysis and experimental results.
— We present the first memory-disaggregated radix tree, SMART, with three key designs for high performance, including a hybrid ART concurrency control scheme, a read-delegation and write-combining technique, and a reverse check mechanism for cache validation.
— We implement SMART and evaluate it using YCSB workloads [10]. The evaluation results demonstrate the efficacy and efficiency of SMART.

2 BACKGROUND

2.1 Disaggregated Memory Architecture

Traditional data centers are built on top of distributed monolithic servers, each of which physically packs CPU and memory resources together. Such an architecture causes low memory utilization (<60%) in today's cloud data centers [44, 52, 63]. First, the CPU and memory resources an application requires are allocated from the same monolithic server, which could lead to *memory stranding*, a situation in which all CPUs of the server are allocated, and the remaining unallocated memory on the server cannot be utilized. The experiences in Azure reveal that up to 25% of memory becomes stranded as more CPUs are allocated to applications [40]. Second, each virtual server (e.g., virtual machine, container) used in many cloud data centers is configured to accommodate the peak memory usage, resulting in *untouched memory*, the allocated but unused memory. It is reported that only around 50% of the allocated memory is used in Google [52]. Some scheduling-based methods (e.g., memory harvesting [18]) have been proposed to attack the above issues. However, the accurate estimation of memory usage of heterogeneous applications is a long-standing challenge.

DM is an increasingly prevalent architecture proposed to address the above resource utilization issue at the architectural level [21, 35, 51, 53, 56, 66], the idea of which can work both on special servers and commodity servers, i.e., physical disaggregation and logical disaggregation. As shown in Figure 1, it physically separates computing (e.g., CPUs) and memory (e.g., DRAM) resources into two independent resource pools, i.e., a computing pool and a memory pool. **Compute nodes (CNs)** in the computing pool own powerful computing resources but only have a small piece of memory serving as local caches. In contrast, **memory nodes (MNs)** in the memory pool are equipped with masses of memory but only own a few wimpy computing cores for simple tasks such as establishing network connections and allocating memory spaces. Since all the CNs can access and fully use the whole shared memory on MNs, DM can achieve great resource utilization.

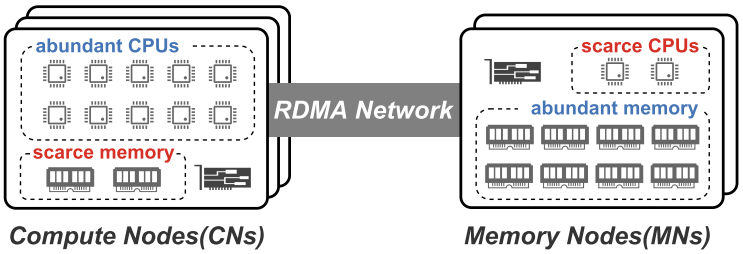


Fig. 1. The architecture of disaggregated memory.

A high-speed network with high bandwidth and low latency, e.g., RDMA network, is a crucial component in the DM architecture that interconnects CNs and MNs [13, 20]. RNICs allow CNs and MNs to communicate with each other using *one-side verbs* (e.g., RDMA_READ, RDMA_WRITE, RDMA_CAS) or *two-side verbs* (e.g., RDMA_SEND, RDMA_RECV). One-side verbs are preferred on the DM architecture to enable computing-side clients to operate directly on the disaggregated memory without involving the weak CPUs on MNs.

2.2 B+ Trees on Disaggregated Memory

Tree indexes are critical for many applications requiring range queries. All previously proposed tree indexes on DM are variants of the B+ tree, including FG [72] and Sherman [65]. FG is the first RDMA-based index supporting DM. It uses a B-link tree structure and completely leverages one-sided verbs to perform index operations, with RDMA-based spin locks for concurrency control. Since FG directly ports the spin-lock-based concurrency control and B-link tree node designs on monolithic servers to DM, its performance suffers from severe network contention on lock retries and write amplification on B-link tree nodes. Sherman [65] is the state-of-the-art B+ tree on DM that addresses the network contention and write amplification issues of FG. First, it addresses the network contention on lock-fail retires with a ***hierarchical on-chip lock (HOCL)*** scheme. The network requests on lock-fail retries are reduced with a local lock table shared among clients on the same CN. The on-chip memory of RNICs is leveraged to reduce PCIe transmissions further. Second, it mitigates the write amplification by allowing fine-grained modification to B+ tree nodes with a *two-level version mechanism*. Therefore, Sherman achieves much better performance than FG. However, Sherman still suffers from the natural performance bottleneck of B+ trees, i.e., inherent read amplification, which will be analyzed in Section 3.

2.3 Radix Tree

The radix tree is another popular tree index structure. It stores the segmented key in the top-down search path over the tree rather than storing the whole key in the internal node. Specifically, each internal node in the radix tree consists of an array of child pointers. Each pointer is associated with a segment of bits of the whole key, called *partial key*, as shown in Figure 2.

Path compression. Path compression is an optimization method for the radix tree to reduce tree height by removing one-child internal nodes and can be implemented in three ways [38]: *(1) The optimistic method* simply abandons the partial keys in the removed nodes and instead stores a depth value to ensure the subsequent traversal process. *(2) The pessimistic method* stores all the partial keys of the removed nodes in the header of the subsequent node. *(3) The hybrid method* integrates the two methods above by storing partial keys into the fixed-sized header of the subsequent node, together with a depth value to ensure the subsequent traversal if some partial keys overflow from the header.

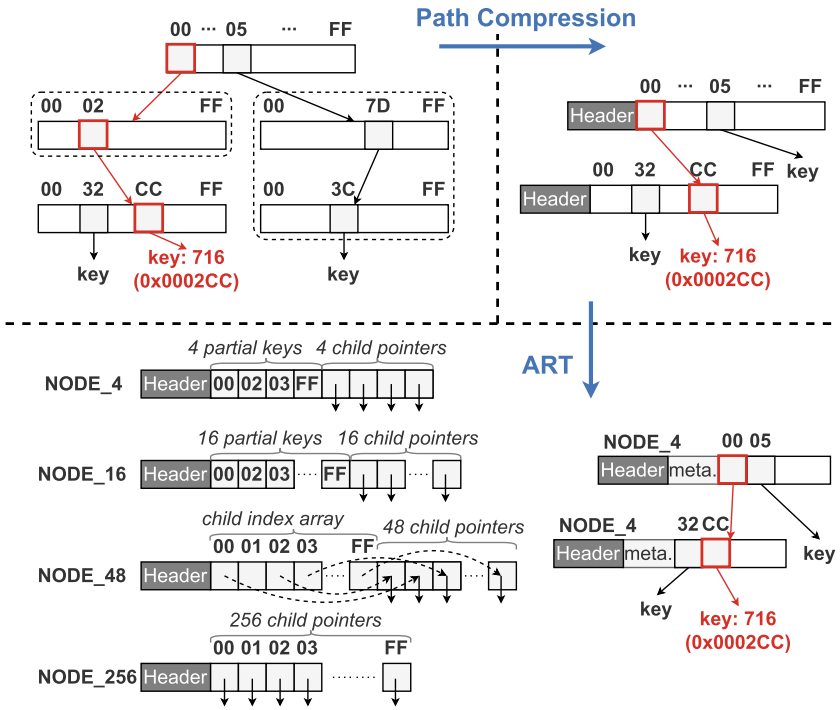


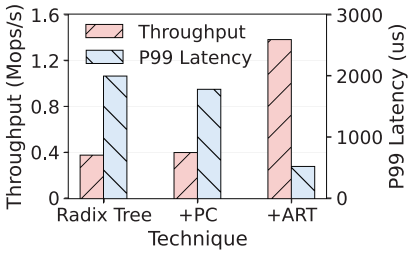
Fig. 2. The optimization process from the basic radix tree to ART. For clarity, hexadecimal partial keys are shown.

Adaptive radix tree (ART). ART [38] is the state-of-the-art variant of the 8-bit-span radix tree, designed to optimize the memory utilization of traditional radix trees. Traditionally, an internal node of a radix tree has all 256 pointers representing all possible partial keys. Many pointers are empty due to the sparse key distribution [38], wasting memory space in these internal nodes. ART addresses the issue by proposing four well-designed internal node structures with different numbers of pointers, i.e., 4, 16, 48, and 256. It dynamically chooses the best-fit internal node structure to save memory space. As for concurrency control, ART is synchronized using a lock-based algorithm, i.e., the ***read-optimized write exclusion (ROWEX)*** protocol [39]. There are some proposed ART-based indexes designed on monolithic servers [31, 36, 37, 45], while none of them is designed for DM.

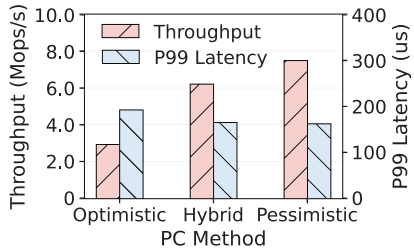
3 ANALYSIS OF TREE INDEXES BUILT ON DM

Since there is no radix tree designed for DM, in this section, we first discuss the design choices for porting a radix tree to DM (Section 3.1). We then theoretically and experimentally compare B+ trees with the radix tree (specifically, ART) that we port to DM (Section 3.2). At last, we present the challenges of designing a high-performance ART on DM (Section 3.3).

Unless otherwise stated, all the experiments in this section are conducted with eight CNs and one MN, each equipped with a 100 Gbps Mellanox ConnectX-6 RNIC. Each CN launches 32 clients with one shared 600 MB cache. We use YCSB workloads [10] (including 60 million entries, using the default Zipfian distribution of skewness 0.99) with 32-byte string keys and 64-byte values, which is typical in real-world workloads [4, 70].



(a) The efficacy of path compression and ART to the radix tree on DM under the YCSB C workload (100% read) with no cache.



(b) The performances of three path compression methods on DM under YCSB insert workload (100% insert) with sufficient caches.

Fig. 3. Analysis of traditional optimizations of the radix tree on DM. "PC" means path compression.

3.1 Radix Tree on DM

As mentioned in Section 2.3, path compression (including optimistic, hybrid, and pessimistic methods) and ART are optimizations for a local radix tree to reduce tree height and improve memory utilization, respectively. In this section, we experimentally show the efficacy and efficiency of these optimizations in the DM architecture, based on which we summarize the design choices for porting a radix tree to DM.

Design choice 1: ***Path compression and ART are both necessary for the radix tree on DM.*** In the DM architecture, the remote tree should be traversed layer by layer via *one-side RDMA verbs* due to the weak memory-side computing power. Therefore, the tree height determines the RTTs required for a traversal. Since path compression can structurally reduce the tree height, it can save RTTs and thus reduce request latency.

The memory-side network bandwidth is generally the performance bottleneck in the DM architecture [28]. As the clients usually need to read the whole tree nodes (including metadata and child pointers) during traversal, the node size determines the network bandwidth consumption. Since ART can reduce the sizes of internal nodes, it can save network bandwidth consumption and thus improve overall throughput.

We evaluate these two optimizations in a radix tree we port to DM with the YCSB C workload, as shown in Figure 3(a). Path compression reduces the P99 latency by 11%, since it can reduce the tree height. The ART design can bring 3.5× improvement in throughput, since it has much smaller internal nodes and thus consumes less network bandwidth during each request.

Design choice 2: ***The pessimistic method is the best choice for path compression on DM.*** The insertion process of a radix tree is different if the tree adopts a different path compression method. Specifically, the optimistic method needs two tree traversals to insert a nonexistent key. One entire tree traversal is required to search for the nonexistent key. The existence of the key is unknown until a different key stored in the leaf node is found. The other traversal executes the actual insertion, where the insertion position is determined by both the target key and the key found in the previous search.

In contrast, the pessimistic method can insert the nonexistent key through one traversal. The existence of the key can be determined in advance, since all the compressed partial keys are stored in the headers. The insertion position is where a mismatch between the target key and the partial keys along the search path occurs.

The hybrid method integrates the two insertion process above. If all the partial keys of the target key are stored in the limited fixed-sized headers in the hybrid method, then the insertion process

Table 1. Read and Write Amplification Factors of Different Trees

	ART	B+ Tree	Sherman
Read	$\frac{M_1+E}{E}$	$\frac{M_2+S\cdot E}{E}$	$\frac{M_2+S\cdot(M_3+E)}{E}$
Write	$\frac{M_1+E}{E}$	$\frac{M_2+S\cdot E}{E}$	$\frac{M_3+E}{E}$

Table 2. An Example of Calculated Read and Write Amplification Factors

	ART	B+ Tree	Sherman
Read	1.10	32.7	33.0
Write	1.10	32.7	1.01

is the same as that of the pessimistic method. Otherwise, the hybrid method needs two traversals to insert the key, like the optimistic one.

To verify this, we compare the insert performances of these three path compression methods, as shown in Figure 3(b). As the pessimistic method can conduct each insertion through only one traversal, it shows the best performance on DM.

3.2 Motivations: B+ Tree vs. ART on DM

The main problem of B+ trees on DM is their severe read and write amplifications. Generally, in internal nodes, the B+ tree stores the whole keys. In leaf nodes, the B+ tree stores multiple keys together. Without optimizations, the B+ tree needs to read and write the entire nodes during each index operation, causing serious read and write amplifications. In the following, we first theoretically compare the read and write amplifications of ART with the B+ tree and the write-optimized B+ tree (i.e., Sherman [65]). We then experimentally show the performance impacts due to the read amplification.

3.2.1 Theoretical Analysis. The read and write amplification factors of different tree structures are shown in Table 1, respectively. We assume the internal nodes are cached and no node split occurs for brevity. M_1 and M_2 denote the metadata size of the leaf node of the radix tree and B+ tree, respectively. M_3 denotes the size of the additional metadata (i.e., entry-level versions) that Sherman applied to each key-value item. S denotes the span size (i.e., the number of items stored in a node) of the B+ tree node. E denotes the key-value item size. Thus, the node size of the B+ tree is $S \cdot E$.

The amplification factor is defined as the ratio of bandwidth consumption from the server and bandwidth returned to the application. Without optimizations, when a client reads or writes a single key-value item in a tree index, the whole leaf node should be read or written. We use 96-byte items, with 32-byte keys and 64-byte values, for all trees as an example.

The leaf node of the ART contains one item with its metadata. In our implementation, 10 bytes of metadata is enough for each item in ART. The read and write amplification factors are $\frac{M_1+E}{E} = \frac{10B+96B}{96B} = 1.10$. The leaf node of the B+ tree contains S items together with the metadata. The metadata at least includes two fence keys ($2 \cdot 32$ bytes), a valid bit, a lock bit, a 1-byte level field, and two 7-bit versions [65], i.e., 67 bytes in total. We use the default span size in Sherman, which is 32. The read and write amplification factors are $\frac{M_2+S\cdot E}{E} = \frac{67B+32\cdot 96B}{96B} = 32.7$. For Sherman, each key-value item in the leaf node is surrounded by a pair of 4-bit entry-level versions. Thus, the read amplification factor is $\frac{M_2+S\cdot(M_3+E)}{E} = \frac{67B+32\cdot(1B+96B)}{96B} = 33.0$. When writing an item without node splitting, the client only requires to write back the modified item with its associated entry-level versions. Thus, the write amplification factor is $\frac{M_3+E}{E} = \frac{1B+96B}{96B} = 1.01$. As shown in Table 2, ART has the most minor overall amplifications according to the above quantitative calculation.

In addition to the amplification factors, tree height is a potential factor affecting the overall performance of tree indexes on DM. When traversing a tree on DM without caches, the higher the tree is, the more network round trips it requires. The tree height of ART is $\alpha \cdot \frac{K}{P}$, where α, K, and P are the path compression ratio, key size, and the size of each partial key, respectively. The path

compression ratio, ranging from 0 to 1, represents the effect of path compression in ART, which is determined by the key distribution. The tree height of the B+ tree is $\lceil log_S N \rceil$, where S and N are the span size of the node and the total number of key-value items in the tree, respectively. With 60 million key-value items loaded, the tree height of Sherman, whose span size is 32, is $\lceil log_S N \rceil = \lceil log_{32} 6e^7 \rceil = 6$.

Note that there is a tradeoff between the amplification factor and the tree height of the B+ tree. A smaller node size results in a higher B+ tree with smaller amplifications and vice versa. However, with long items (e.g., 32-byte keys and 64-byte values), limiting the node size of the B+ tree to a small value (e.g., 256 bytes) is unacceptable. In this case, the span size of the B+ tree is only $\lfloor \frac{256B}{32B+64B} \rfloor = 2$. This results in huge cache consumption, since the smaller the span size, the more pivot keys are stored in internal nodes, and the more computing-side memory is consumed to cache the internal nodes. Besides, the tree height in this case is $\lceil log_S N \rceil = \lceil log_2 6e^7 \rceil = 26$, which leads to high latency when a cache miss occurs.

3.2.2 Experimental Results. To show the impact of read amplification on the performance, we compare the performances of Sherman and ART under *read-only workloads*. The impact of write amplification is similar. We observe that the amplification leads to low throughput and high latency of B+ trees on DM.

Observation 1: *The throughput of the B+ tree is bounded by network bandwidth.* The memory-side network bandwidth is generally the performance bottleneck in the DM architecture [28]. The read and write amplifications of B+ trees cause more bandwidth consumption for each request, exacerbating the network bottleneck and resulting in low throughput.

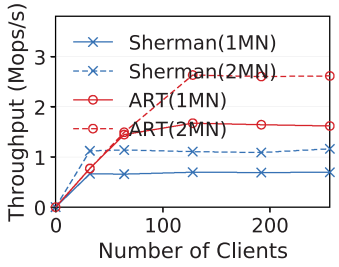
As shown in Figure 4(a), with an increasing number of clients, the limited bandwidth prevents the throughput of Sherman and ART from continually rising. With the same RNIC bandwidth, Sherman has a lower peak throughput than ART due to the severe read amplification. As shown in Figure 4(b), the larger the key size or the span size is, the larger the read amplification is, which decreases the peak throughput of Sherman.

A computing-side cache is usually used for caching the internal nodes of the B+ tree on DM. As shown in Figure 4(c), with the increasing size of the cache, the throughput of Sherman keeps bounded by the bandwidth bottleneck and finally saturates at 4.17 Mops/s. The bandwidth consumption from the server equals the maximum network bandwidth of 100 Gbps (12.5 *GBps*), and the bandwidth returned to the application is $4.17\ Mops/s \cdot 96B = 0.39\ GBps$. Thus, the measured read amplification factor of Sherman is $12.5\ GBps\ /\ 0.39\ GBps = 32.1$, which is close to our theoretical analysis in Section 3.2.1.

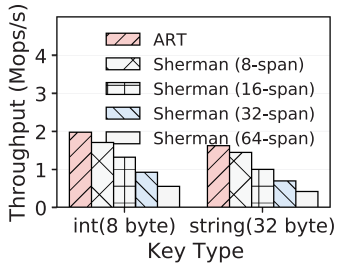
In contrast, without the read amplification from leaf nodes, the throughput of ART reaches about 45 Mops/s, which is the IOPS upper bound of the RNIC we use. This indicates that ART can make full use of the RNIC capacity and achieve the best resource efficiency as DM desires.

Observation 2: *The latency of the B+ tree is worsened by early network congestion.* Network congestion occurs when computing-side requests saturate the bandwidth or IOPS upper bound of RNICs. As the number of clients keeps growing, excess client requests need to queue up across the network, which results in latency deterioration. The read and write amplifications make B+ trees consume the bandwidth rapidly, expediting the process of network congestion.

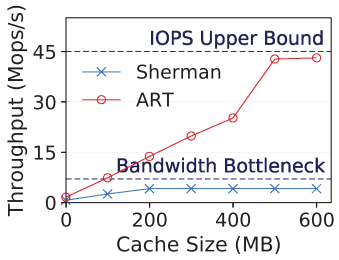
As shown in Figure 4(d), with the increase of throughput, the latency of Sherman and ART is stable in the beginning and then experiences a sudden surge due to the network congestion. Moreover, with the same memory-side RNIC bandwidth, Sherman has a much smaller inflection point (i.e., the throughput threshold that triggers network congestion) than ART. As a result, Sherman shows an extremely high latency with relatively few clients. By contrast, ART has a high tolerance to this latency deterioration, thanks to its small amplifications.



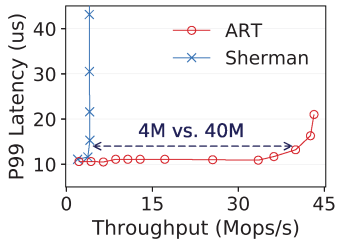
(a) The throughput bottleneck with no cache.



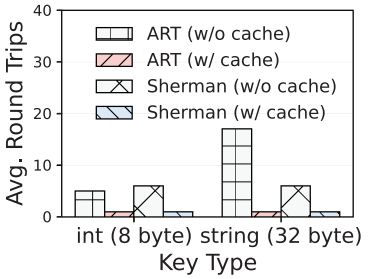
(b) The impact of key size and span size with no cache.



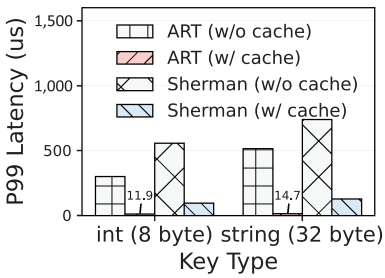
(c) The peak throughput with various sizes of caches.



(d) The latency deterioration with excess requests.



(e) The average number of round trips with different key sizes.

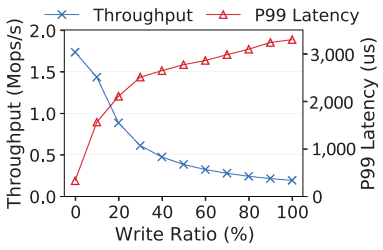


(f) The latency with different key sizes.

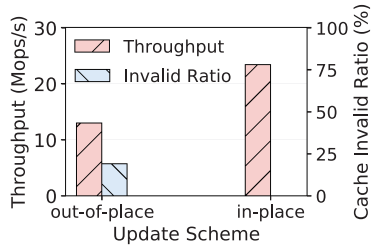
Fig. 4. The read performances of Sherman and ART on DM under the YCSB C workload (100% read).

In addition to the read amplification, the tree height, which determines the number of network round trips, can also affect the latency of tree indexes on DM. The tree height of the B+ tree is determined by the span size and the total number of loaded key-value items, while that of ART is related to the key size.

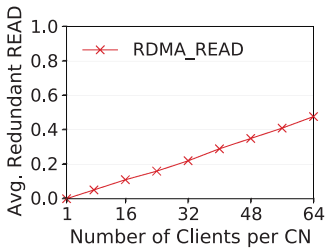
As shown in Figure 4(e), without caches, the average number of network round trips of ART is slightly smaller than that of Sherman with 8-byte keys. However, ART requires more network round trips than Sherman with 32-byte keys. Both ART and Sherman require only one network round trip with caches, since the internal nodes are cached. Figure 4(f) shows the corresponding latency. Even though more network round trips are required, the latency of ART is still lower than that of Sherman. This is because the read amplification of Sherman causes severe latency deterioration.



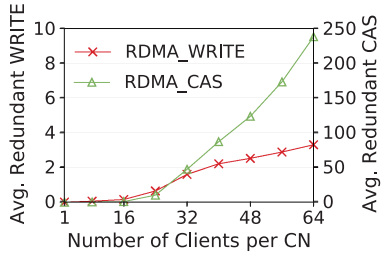
(a) The write performance of ART under the YCSB insert workload (100% insert) with no cache.



(b) The performance degradation caused by cache thrashing under the YCSB A workload (50% read + 50% update) with sufficient caches.



(c) The inter-client redundant I/Os on DM in terms of reads.



(d) The inter-client redundant I/Os on DM in terms of writes.

Fig. 5. The challenge of building high-performance ART on DM under hybrid read-write workloads.

3.3 Challenges: ART on DM

Even though ART has superiority under *read-only workloads*, it suffers from significant challenges on DM under *hybrid read-write workloads*.

Challenge 1: *Lock-based concurrency control of ART causes poor write performance.* Existing ART adopts lock-based algorithms to perform synchronization [39]. However, lock operations are expensive on DM and lead to poor write performance, as shown in Figure 5(a). Specifically, unlike local memory, each lock operation on DM requires additional network transmission (e.g., RDMA_CAS). Furthermore, the lock conflict mechanism (i.e., busy waiting) causes frequent RDMA retries when failing to acquire a lock, which wastes the limited IOPS of RNICs and reduces the throughput.

One feasible solution is to design lock-free algorithms. However, lock-free design is not the best choice for ART as well. Specifically, an out-of-place update scheme is required for lock-free algorithms to update items larger than 8 bytes. It atomically compares and swaps the corresponding 8-byte addresses instead of modifying the items in place, as the latter cannot be realized atomically. However, in high-concurrency scenarios, a mass of out-of-place updates lead to frequent changes in the addresses of items. This brings about the severe cache coherence issue, since the old addresses of the items have been cached in other CNs. Even worse, in skewed workloads, the addresses of hot items are changed continuously and repeatedly, resulting in cache trashing.

To verify this, we evaluate the two update schemes in ART with the YCSB A workload,[1] as shown in Figure 5(b). The out-of-place scheme brings about an average of 19.1% invalid cached

[1]To eliminate the impact of concurrency conflicts, we scatter the update part of workloads among clients without intersection.

addresses of leaf nodes and thus results in a 44.5% throughput decline compared with the in-place scheme.

Challenge 2: ***Inter-client redundant I/Os on DM waste the limited IOPS of RNICs.*** As mentioned in **Observation 1**, B+ trees suffer from bandwidth bottleneck, while ART can break through the bottleneck and achieve the IOPS upper bound of RNICs, with small read and write amplifications.

However, we find that there are redundant I/Os that waste the limited IOPS of RNICs in the DM architecture, hindering ART from continually breaking through the IOPS upper bound. Specifically, taking read operations as an example, when several clients on the same CN read the same key-value item concurrently, they send identical RDMA_READs across the network. This is superfluous duplication of effort, since all these requests do the same transmission work.

To measure the extent of underlying inter-client redundant reads, we launch various numbers of clients on the same CN. Each client continuously issues 1 KB RDMA_READs, with their destination addresses following a Zipfian distribution of skewness 0.99 (i.e., the same as YCSB's). As shown in Figure 5(c), during each read time window, the average number of redundant RDMA_READs increases with the number of clients and achieves up to 0.48 with 64 clients, implying 48% read performance improvement potential.

As for inter-client redundant writes, we issue constant RDMA_WRITEs with lock-based concurrency control via RDMA_CASes from each client. As shown in Figure 5(d), during each write time window (including lock acquirement and release), the average number of redundant RDMA_WRITEs grows and reaches up to 3.3, indicating around 330% write performance improvement space with 64 clients. Interestingly, the number of redundant writes is more than the read one, since redundant writes inevitably exacerbate the concurrency conflicts, leading to a longer write time window and thus more redundant writes in return. The nearly exponential growth of the redundant number of RDMA_CASes saturates the IOPS upper bound rapidly and causes poor write performance.

Challenge 3: ***Structural features of ART deteriorate the problem of computing-side cache invalidation.*** As presented in Section 2.3, *path compression* and *adaptive nodes* are two important structural features that reduce memory consumption by reducing the tree height and the node size, respectively. However, these two features introduce new cache validation problems. For instance, adjustments on the parent-child relationship of nodes may happen during insertion into compressed nodes. The caches on other CNs still store the old content of the parent node. If a client on those CNs does not conduct a cache verification, then it incorrectly reads the old child node according to the outdated cache and thus fails to access the newly inserted node. Similarly, node type changes are invisible by the computing-side cache either, which may lead to incomplete node fetching.

4 SMART DESIGN

We propose SMART, a high-performance ART for DM. Figure 6 shows the overview of SMART. To improve the efficiency of concurrency control (**Challenge 1**), we present a *hybrid ART concurrency control* scheme. The scheme contains a lock-free internal node design and a lock-based leaf node design to achieve high write performance without cache thrashing (Section 4.1). To save the limited IOPS of RNICs (**Challenge 2**), we propose an RDWC technique to eliminate inter-client redundant I/Os (Section 4.2). To handle the cache validation (**Challenge 3**), we co-design SMART with an *ART cache* (Section 4.3), including a reverse check mechanism. We then introduce how we use coroutines further to improve the computing-side request throughput of SMART (Section 4.4). Last, we summarize the operations (i.e., insert, search, update, delete, scan) that SMART supports (Section 4.5).

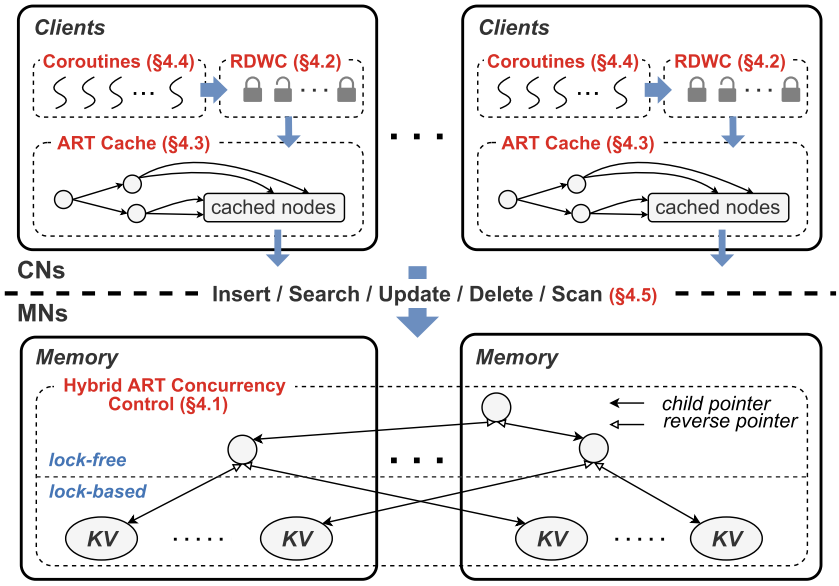


Fig. 6. The overview of SMART.

4.1 Hybrid ART Concurrency Control

In this section, we first describe the data structures and concurrent operations of the hybrid concurrency control scheme in SMART. We then introduce RDMA-related optimizations.

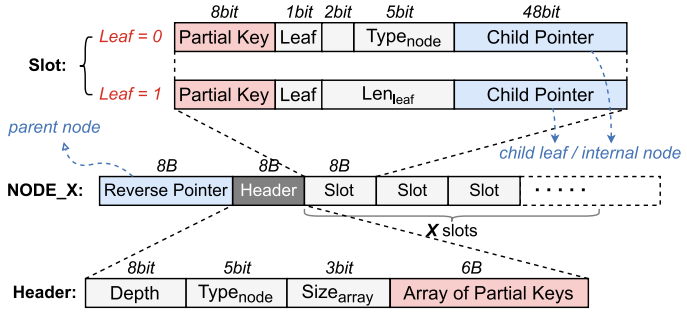
4.1.1 *Data Structures.*

Lock-free internal node. As the addresses of internal nodes change more infrequently, internal nodes do not cause cache thrashing like leaf nodes. Hence, it is feasible for lock-free internal nodes to achieve high performance. We modify the internal nodes of ART as follows:

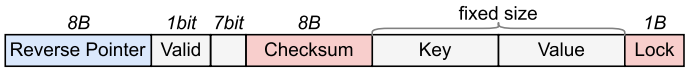
(1) Homogeneous adaptive internal node. As illustrated in Figure 2, a naive ART stores partial keys and child pointers separately. Such a heterogeneous design makes it hard to design a lock-free algorithm, since the separated partial key and child pointer should be modified atomically. Besides, it incurs additional read amplification due to the inflexible fixed-sized internal nodes.

We come up with a homogeneous internal node design that embeds the partial keys into slots. First, this enables a child pointer to be modified together with its corresponding partial key atomically, laying the foundation for lock-free algorithms. Second, the read amplification can be reduced, since internal nodes can have an arbitrary number of slots.

As shown in Figure 7(a), an internal node of SMART consists of an 8-byte reverse pointer, several 8-byte slots, and an 8-byte header. The reverse pointer is used for cache validation, which will be presented in Section 4.3. As for each slot, apart from the embedded 8-bit partial key and the 48-bit child pointer, we add a 1-bit $Leaf$ field to indicate whether the pointer is pointing to a leaf node. When $Leaf$ is set, a Len_{leaf} field is provided, which is used to support variable-sized keys (Section 4.6). When $Leaf$ is unset, there is a 5-bit $Type_{node}$ field to indicate the type of the following internal node. Note that SMART mainly uses the $Type_{node}$ to reduce the network bandwidth consumption rather than memory consumption. When fetching an internal node, SMART can RDMA_READ only the required number of slots according to the $Type_{node}$ field, reducing the read amplification and thus saving the network bandwidth.



(a) The homogeneous adaptive internal node with the pessimistic 8-byte header.



(b) The update-in-place leaf node with the rear embedded optimistic lock.

Fig. 7. The structure of the internal node and the leaf node in SMART. The reverse pointer and the in-header $Type_{node}$ field are used for cache validation.

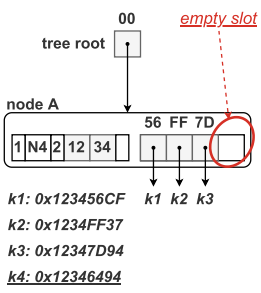
(2) Pessimistic 8-byte header of the internal node. We choose the pessimistic method for path compression, since it can insert a nonexistent key through one traversal, as mentioned in Section 3.1. Besides, following previous designs [36, 39, 45], we fix the header size to 8 bytes, which can be changed atomically. If some partial keys overflow from the header, then we store them in an empty following node. Although this may increase the tree height, we mitigate this with the help of cache (Section 4.3).

As shown in Figure 7(a), a header consists of an 8-bit $Depth$ field, a 5-bit $Type_{node}$ field, a 3-bit $Size_{array}$ field, and a 6-byte array of partial keys. The $Depth$ field indicates the start position for matching the target key. The $Type_{node}$ field is used for cache validation, which will be illustrated in Section 4.3. The $Size_{array}$ field records the length of the partial key array, where at most six partial keys can be stored.

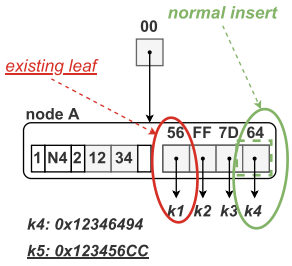
Lock-based leaf node. In-place update schemes are preferred, as it does not cause cache thrashing. To adopt the in-place update, lock-based concurrency control for the leaf node is required. This is acceptable, since locks are fine-grained, as each leaf node in the radix tree only contains one key-value item. We design the leaf node structure as follows for concurrency control:

(1) Checksum-based update-in-place leaf node. The in-place update scheme overwrites the leaf node at the same address, causing conflicts among readers and writers. To avoid conflicts, we adopt an optimistic lock in each leaf node with a checksum-based consistency check mechanism [48, 65], where the fixed-sized key-value item in the leaf node is protected by a checksum. For write-write conflicts, an exclusive lock is used to synchronize the writers. As for read-write conflicts, when a writer modifies the leaf node, the checksum is re-calculated based on the new content of the leaf node and written with the new content. The readers verify the checksum after reading the leaf node. If the checksum verification fails, then the reader conducts a re-read.

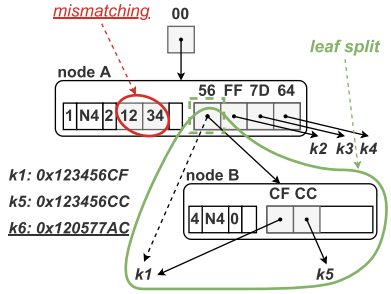
Apart from the checksum-based method, version-based methods [13, 33, 49, 65] are also proposed to detect the read-write conflicts. However, when applied to DM, existing version-based



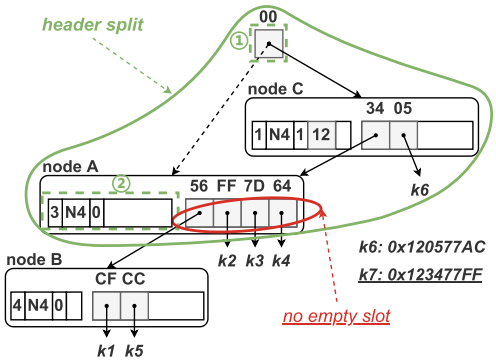
(a) Initial state. Before inserting ***k4***.



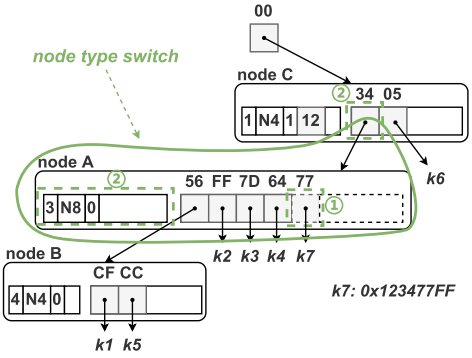
(b) ***Normal insert*** of ***k4***. Before inserting ***k5***.



(c) ***Leaf split*** when inserting ***k5***. Before inserting ***k6***.



(d) ***Header split*** when inserting ***k6***. Before inserting ***k7***.



(e) ***Node type switch*** when inserting ***k7***.

Fig. 8. A step-by-step example of inserting several new keys into SMART with 8-bit partial keys. For clarity, hexadecimal partial keys are shown and reverse pointers are omitted. Each thick dotted box indicates an atomic CAS.

methods will be inefficient, incorrect, or complicated. We will discuss their feasibility for SMART in Section 4.6.2.

(2) Rear embedded lock. To further reduce the overhead of locks, we combine the lock release with the writing back of the updated leaf node by embedding the lock into each leaf node. Therefore, the two operations can be done via one single RDMA_WRITE. Particularly, to avoid premature lock release, we ensure that the lock release is always triggered after the completion of writing back. We achieve this by placing the lock at the rear of a leaf node, which leverages the in-order delivery property of RNICs [13].

As shown in Figure 7(b), a leaf node of SMART consists of an 8-byte reverse pointer, a *Valid* bit, an 8-byte checksum, a 1-byte rear lock, and a fixed-sized key-value item. The reverse pointer is used for cache validation, which will be illustrated in Section 4.3. The *Valid* bit is used to indicate the deleted state.

4.1.2 Concurrent Operations. Based on the above structural modifications, we demonstrate essential write-related sub-operations with a step-by-step example, as shown in Figure 8. Except for the in-place leaf update, all the sub-operations are lock-free. The complete operation process will be described in Section 4.5.

Normal insert. During an insert, the target partial key may not be in the internal node yet. As shown in Figure 8(b), after the WRITE of the new leaf node (k_4), the client CASes the first

empty slot in the node, together with the new partial key. If the CAS fails, then the client checks whether the return value (i.e., a new value of the slot written by a concurrent client) contains the target partial key. If yes, then the client continues to traverse the tree following the return pointer. Otherwise, the client tries the insert again with the next empty slot.

Leaf split. If an existing leaf node is found during an insert, then a leaf split is needed as shown in Figure 8(c). Specifically, the client first calculates the rest of the longest common key prefix of the two leaf nodes (k_5 and k_1). Then, it allocates sufficient sequentially-connected internal nodes to store the common key prefix in their headers. The last internal node will contain two child pointers pointing to the old and new leaf nodes. All internal nodes and the new leaf node can be written in parallel, after which the client CASes the parent slot to point to the first new internal node. If the CAS fails, then the client continues to traverse following the return pointer.

Header split. If a mismatching for in-header partial keys is found, then a header split is required as shown in Figure 8(d). Specifically, the client allocates a new NODE_4 pointing to the split internal node and new leaf node (k_6), with its header storing the matched part of partial keys. The new internal and leaf node can be written in parallel. Then, the client CASes the parent slot to make it point to the new internal node (①). If CAS succeeds, then the redundant in-header old partial keys are removed via an additional CAS (②). Otherwise, the client continues to traverse following the return pointer.

Note that the correctness of concurrent searches can be guaranteed by the in-header *Depth* value, which indicates the start position for matching the current key. A concurrent search READs the parent node and then the child node. Therefore, there are two situations of read-write conflicts. First, the READ of the parent node occurs after the CAS of the parent slot (①), while the READ of the child node occurs before the CAS of the split header (②). In this situation, redundant in-header partial keys are read, which does not affect the correctness. Second, the former READ occurs before the former CAS (①), while the latter READ occurs after the latter CAS (②). In this case, the reader re-reads the parent slot if finding partial keys missing according to the *Depth* value.

Node type switch. To avoid **copy-on-write (COW)** overhead and additional cache coherence introduced by out-of-place updates (**Challenge 1**), we conduct an in-place node type switch. This is feasible, thanks to the homogeneous adaptive internal node design (Section 4.1.1). To be specific, we pre-allocate the contiguous space of NODE_256 on MNs for each internal node. This consumes a little additional memory but enables lock-free operations during the node type switch. When neither a matching partial key nor an empty slot is found in the current internal node, the client can try to CAS the following empty slots one-by-one, whose addresses are behind the node (①) as shown in Figure 8(e). After a successful CAS, the current best-fit node type can be determined by the index of the newly inserted slot. The client then tries to update the two old $Type_{node}$ values (on the header and the parent slot) with the new one via two concurrent CASes (②), making the newly inserted leaf visible by subsequent search. If both CASes succeed or fail with return values containing $Type_{node}$ values larger than/equal to the expected one, then the node type switch is finished. Otherwise, the client retries the CASes.

In-place leaf update. To update a leaf node, the client first acquires the rear embedded lock in the leaf node. It then WRITEs back the updated leaf node with the re-calculated checksum and the unset lock, after which the in-place leaf update is finished with the lock properly released.

4.1.3 RDMA-related Optimizations. To further optimize performance on DM, SMART adopts the following RDMA-related optimizations [28].

Inline write. For small-sized WRITE (e.g., writing internal nodes smaller than NODE_16 or leaf nodes), the INLINE flag is set, enabling the RNIC to encapsulate payload into the **work queue entry (WQE)** and thus reducing PCIe overhead.

Unsignaled verbs. As for writing commands allowing asynchronous execution (e.g., CAS of the header during ***header split***), SMART unsets the SIGNALED flag to reduce the overhead of polling RDMA completion queues.

Doorbell batching. If a client issues multiple WQEs to the same queue pair (e.g., to the same MN), then a doorbell batching is conducted to reduce PCIe overhead.

4.2 Read Delegation and Write Combining

SMART proposes the RDWC technique on DM to eliminate inter-client redundant I/Os in terms of reads and writes, respectively, to break through the IOPS upper bound. The high-level idea of RDWC is similar to FLOCK [50], a communication framework that enables connection sharing among threads for RDMA networks. In FLOCK, the client side coalesces a set of requests in one message, and then the server processes all the requests in the message. However, FLOCK is unsuitable for DM due to the weak computing power of MNs. Unlike FLOCK, RDWC only focuses on eliminating redundant requests and does not involve MNs. In this section, we first introduce *hash-based local locks*, a critical component to implement RDWC. We then present the processes of the *read delegation* and the *write combining*, respectively.

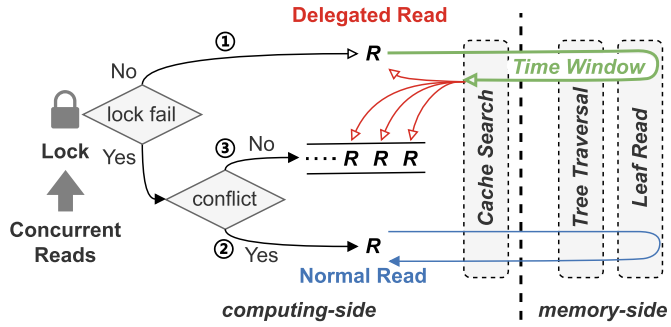
4.2.1 Hash-based Local Locks. The inter-client redundant I/Os on each CN occur among the concurrent read and write operations on the same key or address. Therefore, computing-side local locks are needed to collect the concurrent operations.

We maintain the local locks in each CN as a table, similar to the local lock table of HOCL in Sherman [65]. However, unlike Sherman, which maintains each local lock for a coarse-grained global lock, SMART maintains each local lock for a key (i.e., fine-grained leaf node). It is challenging to store all such locks in each limited computing-side memory. To address this, we use hash-based local locks, where a lock corresponds to a set of keys with the same hash value.

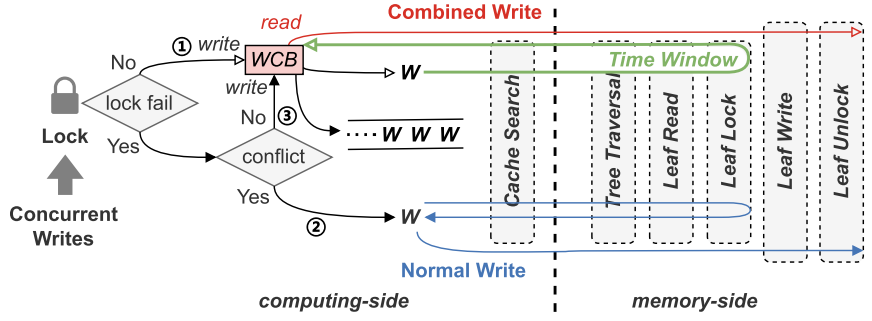
We dynamically maintain a *unique key* in each local lock to solve the hash-conflict problem of our hash-based scheme. Specifically, the first client who acquires a local lock successfully will record its target key as the unique key of this local lock. The subsequent clients who fail to acquire this local lock will conduct a hash-conflict check by comparing their target key with the unique key. If the target key is exactly the same as the unique key, then the client can be involved in the read delegation or write combining. Otherwise, a hash conflict is found, and the client should execute a normal remote read or write on its own for correctness. The unique key is freed when the first client releases the local lock.

4.2.2 Read Delegation. To reduce inter-client redundant I/Os for reads, a *delegation client* can be elected on each CN to execute the same read and then share its RDMA_READ result with other waiting clients. The first client who acquires the local lock successfully is the delegation client, and the subsequent clients who fail to acquire the lock are the waiting clients. The relationship between the delegation client and the waiting clients is similar to that between the first cache miss and the subsequent delayed cache hits in the cache system [5].

We implement this as shown in Figure 9(a). After acquiring the corresponding local lock successfully, the delegation client records its target key as the unique key and then conducts the remote tree search (i.e., including cache search, tree traversal, and leaf node read), which is the time window of read delegation (①). During the time window, the subsequent clients failing to acquire the local lock first execute the hash-conflict check by comparing their target key with the unique key. If a hash conflict is found, then the client executes a normal tree search by itself (②). Otherwise, it pushes itself into a read-waiting queue and waits for the search result from the first client (③). Finally, the delegation client shares its search result with the waiting clients and releases the local lock.



(a) Read delegation.



(b) Write combining.

Fig. 9. The processes of the read delegation and the write combining on SMART, respectively.

4.2.3 Write Combining. **Write combining (WC)** is a normal technology in modern processors [11]. When a processor intends to issue multiple writes to the same memory region in a small time window, it combines the writes into a single burst write to save the system bus bandwidth. This idea, also known as write coalescing, is applied to many storage systems [27, 34, 61]. Inspired by this, we find it feasible to conduct a WC on each CN. When clients intend to make several concurrent key-value writes to the same memory-side key or address, they can combine the writes into a single consensus write to save the network bandwidth and the limited IOPS of RNICs.

We implement WC on DM as shown in Figure 9(b). A client that succeeds in acquiring the corresponding local lock first records its target key as the unique key and writes its new value into the **write combining buffer (WCB)** and then conducts the remote tree insert or update (①). Differently, the time window of write combining is the former partial period of tree insert or update (i.e., cache search, tree traversal, and lock acquirement on leaf node). After that, the client reads the combined consensus result from WCB and then makes an RDMA_WRITE to write back the result and release the remote lock. Finally, the client releases the local lock. During the write-combining time window, the subsequent clients first perform the same hash-conflict check. If a hash conflict is found, then the client performs a normal tree insert or update on its own (②). Otherwise, it first writes its expected value into the WCB (with local lock-based concurrency control), making the value visible to the first client. Then, the client pushes itself into a write-waiting queue to wait for the completion of the remote write (③).

4.2.4 Put Both Together. Naively putting read-delegation and write-combining together may introduce incorrect read results when a client reads a key-value item after writing it. Specifically,

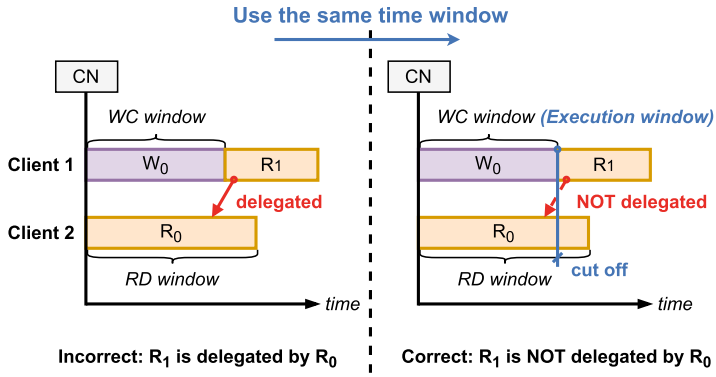


Fig. 10. The issue (the left part) and the solution (the right part) of putting the read-delegation (RD) time window and the write-combining (WC) time window together, respectively.

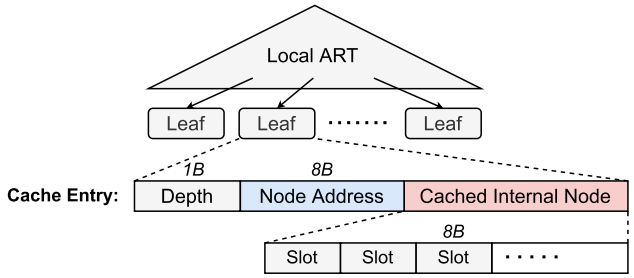
the latter read may be delegated by a client whose read happens before the write operation. In this case, the old value (i.e., the value of the item before the client's write) is returned to the read operation that happens after the write, breaking the causality of the read and write. As shown in Figure 10, we use the same time window for read-delegation and write-combining to address this issue. For clarity, we call the same time window as the *execution window*, define the write and read with causality as W_0 and R_1, and define the read happening before W_0 as R_0. Without loss of generality, we assume that R_0 and W_0 are operations succeeding in acquiring the local locks in the read-delegation time window and the write-combining time window, respectively. The execution window is one of the two time windows that has an earlier end time. It will cut off the other window to make the two time windows end at the same time, thus separating R_0 and R_1 into two different time windows.

Specifically, if R_0 ends earlier than W_0, then the execution window is the read-delegation time window. R_1 has yet to happen during this window, since it should be conducted after W_0 is completed due to their causality. Thus, it is not delegated by R_0. If W_0 ends earlier than R_0, then the execution window is the write-combining time window. Since the execution window will cut off the read-delegation time window, R_1 is also not delegated by R_0 in this case. Consequently, by determining the execution window, the write and read operations with causal relations are included in two non-overlapped time windows, and thus, the above issue can be avoided. To achieve this, we let R_0 and W_0 fairly acquire another local lock when they complete their remote processes, where the winner decides the execution window. The winner cuts off the other time window atomically without blocking the corresponding process, and thus, the read delegation and the write combining can be conducted exclusively and concurrently.[2]

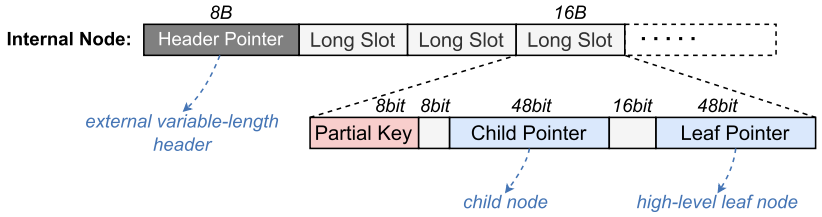
4.3 ART Cache

To reduce remote access during tree traversal, a memory-efficient ART-indexed cache is designed on each CN to store partial internal nodes of SMART. To be specific, utilizing the feature that each radix tree node (excluding header) can be uniquely identified by a key prefix, we adopt a local ART on each CN to index the cached internal nodes. Each CN maintains its local ART without interfering with each other. All operations of SMART first search in the local ART for the deepest internal node matching the prefix of the target key. The operations then start the remote traversal

[2]Note that with N clients in each CN, there are at most N dynamically allocated WCBs and unique keys at the same time, whose memory consumption (i.e., size of N key-value items) is negligible.



(a) The leaf node (*i.e.*, cache entry) of the local ART.



(b) The internal node of the local ART.

Fig. 11. The structure of the local ART in the ART cache.

from the cached internal node to save network round trips. If a client reads a remote internal node, then it will insert the node into the local ART or replace the outdated cached node with the new read one. If a client is steered to the wrong remote node due to the cache invalidation, then it will invalidate and remove the outdated cache node from the local ART. In this section, we first present the structure of the *local ART*. We then introduce the *cache invalidation situations* in SMART and the *reverse check mechanism* to handle the cache invalidation.

4.3.1 Local ART. Figure 11 shows the structure of the local ART in the ART cache. As shown in Figure 11(a), each leaf node (i.e., cache entry) of the local ART contains the snapshot of a traversal context (i.e., the content of an internal node being read from MNs, the *Depth* value, and the address of the node). As shown in Figure 11(b), the internal node of local ART is structurally similar to the remote one (Section 4.1.1) without the reverse pointer, except for the following two designs:

(1) High-level leaf node. For clarity, we define internal nodes adjacent to the leaf nodes as lowest-level internal nodes and others as high-level internal nodes. In the remote ART, key-value items can be inserted into the lowest-level internal nodes, since all the keys can be zero-padded to the same length. However, as a unique identifier for a remote internal node, a key prefix is variable-length and cannot be padded by zero. For instance, "\01" and "\0\01" represent the key prefixes of two remote internal nodes, whose depth is 2 and 3, respectively. With zero padding, these two key prefixes will be indistinguishable. Since the local ART uses key prefixes to index the cache entries, the cache entries should be inserted into high-level internal nodes of the local ART. We call leaf nodes (i.e., cache entries) inserted into the high-level internal nodes as high-level leaf nodes.

Since local locks are much less expensive than remote locks, the lock-free design (Section 4.1.1) is no longer necessary for the local ART. Therefore, we extend the 8-byte slot into the 16-byte *Long Slot*, as shown in Figure 11(b), to realize the high-level leaf node described above. For each *Long Slot*, apart from the 8-bit partial key and the 48-bit child pointer, we add a 48-bit leaf pointer to point to a high-level leaf node, i.e., a cache entry.

(2) External variable-length header. In the remote ART, we fix the header size to 8 bytes for the lock-free algorithm. This is unnecessary for the local ART. We extend the fixed-sized header to a variable-length header in the local ART to store all the corresponding compressed partial keys in each header. Specifically, instead of embedding the header in each internal node, we use an 8-byte header pointer to point to the variable-length header, as shown in Figure 11(b). This is acceptable in the local ART, since local memory access is much less expensive than remote access, which consumes one RTT.

4.3.2 Cache Invalidation Situations. Since we cache the slots of the internal nodes in clients, changing the slots in the disaggregated memory leads to cache invalidation. We analyze all operations that change the slots (i.e., slot insert, update, and delete) and find there are only three types of cache invalidation in the current SMART design, i.e., Type 1: *adjustments on the parent-child relationship*, Type 2: *node type changes*, and Type 3: *deleted nodes*. Specifically:

For slot insert, inserting a new slot does not affect the client cache, since the new slot is not in the client cache.

For slot update, it contains four situations according to the structure of slots in Figure 7(a) (note that the $Partial\ Key$ field keeps unchanged until deleted):

- Updating the $Child\ Pointer$ field. This type of cache invalidation corresponds to Type 1.
- Updating the $Type_{node}$ field. This type of cache invalidation corresponds to Type 2.
- Updating the $Leaf$ field. Since leaf nodes have different addresses from internal nodes, the Leaf field update should be combined with a $Child\ Pointer$ update. Thus, this type of cache invalidation corresponds to Type 1.
- Updating the Len_{leaf} field. This field keeps unchanged, since SMART is currently designed for fixed-sized leaf nodes. The support for variable-sized leaf nodes will be discussed in Section 4.6.

For slot delete, this type of cache invalidation corresponds to Type 3.

4.3.3 Reverse Check Mechanism. To handle the above three types of cache invalidation situations, we design a reverse check mechanism specifically for SMART, as existing solutions on B+ trees are infeasible for ART. We store the check information in remote internal and leaf nodes. A mismatch between check information and cache content indicates an outdated cache entry, which will be invalidated.

(1) Adjustments on the parent-child relationship. We store a reverse pointer in the front of each node to point to its parent, as shown in Figure 7. If the client reads a remote node according to a cached pointer, then it checks whether the reverse address is equal to the node address in the cache entry. If not, then a mismatch is found, which indicates that a newly inserted node (e.g., caused by ***leaf split*** or ***header split***) is invisible to the client due to the outdated cache entry.

(2) Node type changes. We design a $Type_{node}$ field in the header of each node to indicate the current type of the node, as shown in Figure 7(a). If the client reads a remote node according to a cached pointer, then it checks whether the in-header $Type_{node}$ value being read is the same as that in the cached slot. If not, and the in-header $Type_{node}$ value is larger than the cached one, then read the rest of the remote node.

(3) Deleted nodes. We set the in-header $Type_{node}$ value to zero to indicate the deleted state of an internal node. As for a deleted leaf node, the $Valid$ bit is unset.

4.4 Coroutine-based Throughput Boost

In this section, we present how we use coroutines [12, 24] to improve the computing-side request throughput of SMART. A coroutine is a special function that can suspend and resume during

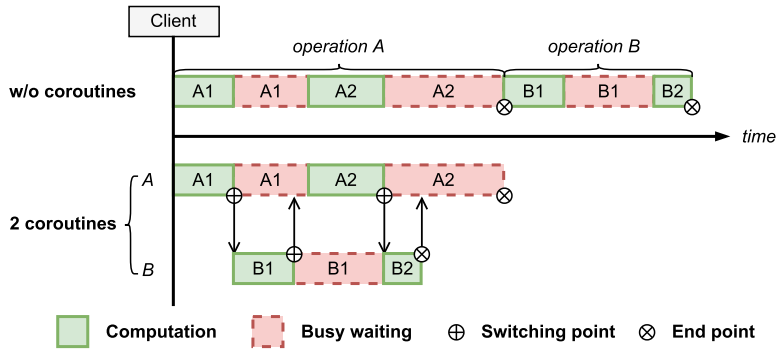


Fig. 12. The timelines of a client thread executing two simultaneous operations with no (the upper part) or two coroutines (the lower part), respectively.

the execution of each thread. It is a lightweight way for each client thread to handle multiple tasks simultaneously. The client thread can schedule the coroutines with some manually defined program points. For brevity, we define these program points as switching points. When a coroutine executes to the switching points, it saves its context and suspends. After that, another coroutine will resume and restore its context.

We implement each operation of SMART as a coroutine function. Thus, each client thread can execute multiple operations simultaneously. Figure 12 shows the timeline of a client thread executing two simultaneous operations with two coroutines, compared with that without coroutines. The client can hide the busy-waiting overhead and achieve higher throughput by scheduling the two coroutines according to the switching points. Specifically, we define the following three types of switching points in the coroutine functions of SMART.

(1) Switching after issuing synchronous RDMA requests. Each operation of SMART contains multiple RDMA requests (e.g., RDMA_READ, RDMA_WRITE, RDMA_CAS). After issuing a synchronous RDMA request, the client should poll the **completion queue (CQ)** to check whether the RDMA request is complete. This wastes the computing-side CPU resources and results in low overall throughput. Therefore, we let the client thread switch to another coroutine after issuing each synchronous RDMA request to hide the RDMA polling overhead.

(2) Switching after failing to acquire remote locks. As mentioned in Section 4.1.1, SMART adopts a lock-based leaf node design. When failing to acquire the lock of the leaf node, the client needs to retry until the lock request is successful. Switching a coroutine that fails to acquire a lock can avoid a dead-lock problem where two coroutines of the same client thread try acquiring the same lock. The coroutine that succeeds in acquiring the lock will never resume if the other one is trapped in lock-fail retries without this switching point. In addition, this type of switching can also hide the busy waiting overhead to some extent, as the lock-fail retries introduce many network I/Os and lead to low overall throughput.

(3) Switching after being pushed into waiting queues. As mentioned in Section 4.2, SMART proposes the RDWC technique to batch concurrent read or write operations. Note that the RDWC technique actually batches the coroutine functions, since we implement each operation as a coroutine function. The coroutines can improve the batch rate of RDWC, since each client thread executes multiple coroutines simultaneously. This switching point can also avoid a dead-lock problem like the previous type. Take read delegation as an example. The elected delegation coroutine will never resume if others of the same thread are trapped in the read-waiting queue without this switching point.

ALGORITHM 1: Search(key, &value).

```
1  // Search a non-empty slot from the cache; READ the root slot if a cache miss occurs
2  slot, cache_entry = CacheSearchOrRootRead(key)
3  RESTART:
4  // If the slot is empty, the key is not found
5  if (slot == NULL)
6      return KEY_NOT_FOUND
7  // READ the leaf or internal node pointed by the slot
8  node = NodeRead(slot.child_pointer, slot.leaf)
9  // If the reverse check fails, invalidate the cache entry and READ the slot
10 if (ReverseCheck(node, cache_entry, &slot) == IS_EXPIRED)
11     goto RESTART
12 if (IsLeaf(node))
13     if (node.key == key) // The key is found
14         value = node.value
15         return SEARCH_FINISH
16     else return KEY_NOT_FOUND
17 else
18     // If the header does not match, the key is not found
19     if (HeaderMatch(key, node.header) == NOT_MATCH)
20         return KEY_NOT_FOUND
21     // Search the node for a slot whose partial key is matched
22     if (SlotSearchOnNode(key, node, &slot) == SUCCESS)
23         goto RESTART // search next level of the tree
24     return KEY_NOT_FOUND
```

4.5 Operations

All operations first search in the cache for the deepest slot that is matched by the prefix of the target key. If none of the cached slots hits, then start the traversal from the tree root slot.

Search. The pseudo-code of the search operation is shown in Algorithm 1. The client first reads the node according to the slot, after which a ***reverse check*** is conducted to check if the cache entry expires. If yes, then invalidate the cache entry and retry this search (lines 7–11). As for a leaf node being read, the target item is found if its key is the same as the target key. Otherwise, it does not exist (lines 13–16). As for an internal node, if all the in-header partial keys are matched, and the next target partial key can be found in a slot, then read the next node along the child pointer in the slot and repeat the process. Otherwise, the target item does not exist (lines 18–24).

Insert/Update. The pseudo-code of the insert and update operations are shown in Algorithm 2. The client first reads the node and conducts a ***reverse check*** like the search (lines 9–11). After that, as for a leaf node, if its key is the same as the target key, then execute an ***in-place leaf update*** (lines 13–16). Otherwise, a ***leaf split*** is needed (lines 17–20). As for an internal node, if a mismatching for the in-header partial keys is found, conduct a ***header split*** (lines 22–26). Otherwise, turn to search among the slots. If the current target partial key can be found in a slot, then read the next node along the corresponding child pointer in the slot and start the process again (lines 27–28). Otherwise, conduct a ***normal insert*** with the first empty pointer slot (lines 29–35). If no empty slot can be found, then a ***node type switch*** is needed (lines 36–39).

Delete. The pseudo-code of the delete operation is shown in Algorithm 3. Delete operations have a similar process as insert operations. A normal delete clears the slot pointing to the target

ALGORITHM 2: Insert/Update(key, value).

```
1  /* For all CAS-related functions below, the slot is updated to the return value of
       the CAS. Other functions are the same as those in Search(). */
2  slot, cache_entry = CacheSearchOrRootRead(key)
3  RESTART:
4  // If the slot is empty, execute a normal insert via a CAS
5  if (slot == NULL)
6      if (NormalInsert(key, value, &slot) == CAS_SUCCESS)
7          return INSERT_FINISH
8      else goto RESTART
9  node = NodeRead(slot.child_pointer, slot.leaf)
10 if (ReverseCheck(node, cache_entry, &slot) == IS_EXPIRED)
11     goto RESTART
12 if (IsLeaf(node))
13     // If the key is found, execute an in-place leaf update
14     if (node.key == key)
15         InPlaceLeafUpdate(key, value, node)
16         return UPDATE_FINISH
17     // Otherwise, execute a leaf split via a CAS
18     if (LeafSplit(key, value, &slot, node) == CAS_SUCCESS)
19         return INSERT_FINISH
20     else goto RESTART
21 else
22     // If the header does not match, execute a header split via CASes
23     if (HeaderMatch(key, node.header) == NOT_MATCH)
24         if (HeaderSplit(key, value, &slot, node.header) == CAS_SUCCESS)
25             return INSERT_FINISH
26         else goto RESTART
27     if (SlotSearchOnNode(key, node, &slot) == SUCCESS)
28         goto RESTART // search next level of the tree
29     // If no matched slot is found, execute a normal insert on the first empty slot
30     for slot in EmptySlots(node)
31         ret = NormalInsert(key, value, &slot)
32         if (ret == CAS_SUCCESS)
33             return INSERT_FINISH
34         if (ret == SAME_PARTIAL_KEY_INSERTED)
35             goto RESTART
36     // If the node is full, conduct a node type switch via CASes
37     if (NodeTypeSwitch(key, value, &slot, node) == SAME_PARTIAL_KEY_INSERTED)
38         goto RESTART
39     return INSERT_FINISH
```

leaf node via RDMA_CAS and unsets the *Valid* bit of the deleted leaf node (lines 6–10). Opposite operations of ***leaf split*** and ***header split*** are conducted for path compression (lines 11–25).

Scan. The pseudo-code of the scan operation is shown in Algorithm 4. At each level of traversal, the client conducts parallel RDMA_READs to fetch all nodes inside the target key range (lines 7–8). For each RDMA_READ, the client processes the node being read in the same way as the search operation, with an additional comparison between partial keys and target key range to exclude unwanted concurrent search paths (lines 14–20). Like many other existing tree indexes [65, 72] on DM, SMART does not guarantee the scan is atomic with concurrent insert or update operations.

ALGORITHM 3: Delete(key).

```
1  /* The implementation of the function Search_() is similar to the function Search(),
       while it additionally returns the slot that points to the target node, the
      node that contains the slot, and the parent slot that points to the node. */
2  RESTART:
3  ret, slot, node, parent_slot = Search_(key)
4  if (ret == KEY_NOT_FOUND)
5      return KEY_NOT_FOUND
6  // Set the slot to NULL via CAS
7  if (NormalDelete(&slot) != CAS_SUCCESS)
8      goto RESTART
9  // Unset the valid bit of the deleted leaf
10 InvalidateNode(slot.child_pointer)
11 // Path compression is available
12 if (CurrentNonEmptySlotNum(node) <= 1)
13     remain_slot = RemainSlot(node, slot)
14     // If the node is empty, recursively remove empty nodes
15     if (remain_slot == NULL)
16         return Delete(KeyPrefix(node))
17     // If the remaining slot points to a leaf node, execute a leaf merge
18     if (remain_slot.leaf && LeafMerge(remain_slot, &parent_slot) == CAS_SUCCESS)
19         InvalidateNode(parent_slot.child_pointer)
20         return DELETE_FINISH
21     // If the remaining slot points to an internal node, execute a header merge
22     if (!remain_slot.leaf && HeaderMerge(remain_slot, &parent_slot) == CAS_SUCCESS)
23         InvalidateNode(parent_slot.child_pointer)
24         return DELETE_FINISH
25     goto RESTART
26 return DELETE_FINISH
```

4.6 Discussion

4.6.1 Support for Variable-sized Keys and Values. SMART currently supports fixed-sized keys and values. For variable-sized keys and values, the optimizations of *update-in-place leaf node* and *rear embedded lock* in SMART are no longer applicable. Instead, SMART can use the RCU scheme to out-of-place update the leaf node to support variable-sized keys and values. The search, insert, and delete operations on variable-sized key-value items are the same as those on fixed-sized ones.

To apply the RCU scheme, when updating a key-value item, SMART first allocates a new leaf node and writes the new item into it via an RDMA_WRITE. SMART then updates the parent slot of the old leaf node via an RDMA_CAS, with the Len_{leaf} field storing the new leaf node size and the child pointer pointing to the new leaf node. Like PACTree [31], SMART can use the **epoch-based memory reclamation (EBMR)** technique [17, 23] to detect the state in which no thread is reading the old leaf node, and the node can be safely freed.

As for the leaf node structure, SMART can follow the design in RACE [73]. As shown in Figure 13, the leaf node structure includes a Len_{key} field and a Len_{val} field, which indicate the sizes of the following Key and $Value$ fields, respectively. SMART can use the 7-bit Len_{leaf} field in the parent slot and a pre-configured $length\_unit$ value to indicate the length of the leaf node. The maximum length of a leaf node is $2^7 \cdot length\_unit$. When a key-value item exceeds the maximum length, SMART can store the remaining content in a second key-value block linked to the leaf node.

ALGORITHM 4: Scan(from, to, &results).

```
/* Most functions are the same as those in Search(). */
slots, cache_entries = CacheScanOrRootRead(from, to)
RESTART:
// If no slot is to be processed, the scan is finished
if (IsEmpty(slots))
    return SCAN_FINISH
// Read all the nodes pointed by the slots via doorbell batching
nodes = NodeBatchRead(slots)
next_slots = EmptySet()
for slot, node, cache_entry in slots, nodes, cache_entries
    if (ReverseCheck(node, cache_entry, &slot) == IS_EXPIRED)
        next_slots.add(slot)
        continue
    // For a leaf node, check if the key is within [from, to)
    if (slot.leaf && node.key >= from && node.key < key)
        results.add(node.key, node.value)
    // For an internal node, filter out slots whose key ranges overlap [from, to)
    if (!slot.leaf)
        for next_slot in FilterSlotsOnNode(node, from, to)
            next_slots.add(next_slot)
slots = next_slots
goto RESTART
```

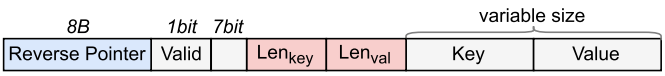


Fig. 13. The structure of the variable-sized leaf node.

Moreover, the cache validation mechanism (Section 4.3.3) can be extended to support variable-sized leaf nodes with a new cache invalidation situation, i.e., Type 4: *leaf node length changes*. When a client reads a remote leaf node according to a cached slot, it checks whether the sum of the Len_{key} and Len_{val} values equals the $Len_{leaf} \cdot length\_unit$ value. If not, then the cached slot is invalid.

4.6.2 Feasibility of Version-based Consistency Check. There are generally four existing version-based approaches for readers to detect concurrent writes on data, i.e., *single version* [71], *Lamport versions* [33], *bookend versions* [49, 65], and *FaRM cache line versions* [13].

The *single version* method requires two dedicated RDMA_READs to read the version number and data, respectively. After reading the data, the version number is read once more via RDMA_READ and compared to the initial value to detect concurrent modifications to the data. This method is inefficient on DM due to multiple RDMA_READs.

The *Lamport versions* method maintains two version numbers for each data, i.e., v_1 and v_2. The writer increments v_1 before writing the data and increments v_2 after writing. The reader reads v_2 before reading the data and v_1 after reading it. The concurrent modifications to the data are detected by comparing the two version numbers. Unfortunately, when implemented on DM, the readers also need to send multiple RDMA_READs, which is expensive.

The *bookend versions* method embeds a pair of version numbers into the data. One is stored at the start of the data, and the other is stored at the end. The data obtained via RDMA_READ is

consistent only when the two versions are identical. Although this method only requires a single RDMA_READ in the best case, it presumes that the RDMA_READ is performed in increasing address order, which is proved to be incorrect [71].

The *FaRM cache line versions* method is a technique proposed by FaRM [13], which leverages the cache-coherent DMA specified by x86. It stores a version number at the beginning of every cache line and detects the concurrent modifications to the data by comparing the cache line versions that the data contains. Like the checksum-based method, this method is correct and efficient on DM with only one RDMA_READ required in the best case, and thus, it is feasible for the consistency check to the leaf node of SMART. However, it is complicated due to the cache-line-based data management. Therefore, we choose to use the checksum-based method in our design (Section 4.1.1).

4.6.3 Generality of Techniques in SMART. Some techniques in SMART can also be applied to other kinds of indexes. Particularly: (1) The *RDWC* technique can benefit any tree indexes, since it is transparent to the lower-level index structures. When applied to other index structures, it brings about the same performance improvement as applied to ART. (2) The *reverse check mechanism* can benefit any radix-tree-based indexes. It is designed to handle the cache validation problems caused by ART's features. (3) The *rear embedded lock* can be adopted in any lock-based structures on DM to save one RTT.

4.6.4 The First Lock-free ART Design. A pure lock-free ART can be formed with the lock-free node design in Figure 7(a) and a lock-free leaf node design with a traditional RCU scheme. To our knowledge, this is the first lock-free ART design. In our implementation, SMART can degenerate into the pure lock-free ART by disabling the optimizations of *update-in-place leaf node* and *rear embedded lock*.

5 EVALUATION

5.1 Experimental Setup

5.1.1 Testbed. We run all experiments on 16 physical machines (16 CNs and 2 MNs)[3] on the Clemson cluster of CloudLab [14]. Each machine has two 36-core Intel Xeon CPUs, 256 GB of DRAM, and one 100 Gbps Mellanox ConnectX-6 RNIC. Each RNIC is connected to a 100 Gbps Ethernet switch. Each MN owns 64 GB DRAM and one CPU core for network connection and memory allocation. Each CN owns 4 GB DRAM and 64 CPU cores, where each core can serve as a client. The MNs register memory with huge pages to reduce page translation cache misses of RNICs [13].

5.1.2 Workloads. Without explicit mention, we use the index microbench [67] to generate YCSB [10] workloads like previous work [6, 31, 47]. We evaluate SMART with six YCSB core workloads: A (50% read, 50% update), B (95% read, 5% update), C (100% read), D (latest-read, 95% read, 5% insert), E (95% scan accessing up to 100 items, 5% insert), and an additional LOAD (100% insert) workloads, using the default Zipfian distribution for all workloads except for YCSB LOAD and D. We also evaluate SMART with four uniform workloads: A, B, C, and D, which are generated using the uniform distribution for the corresponding YCSB workloads. For most workloads, we test two key types, i.e., integer (8-byte) and string (32-byte). For string workloads, we use 125 million publicly available email addresses [16] and conduct a common pre-processing (i.e., swap username and domain fields of email addresses) like previous work [38, 46, 47, 67]. We use 8-byte values consistent with prior work [6, 29, 46, 49, 65, 68]. For each workload, we populate 60 million keys before conducting 60 million operations, except for the LOAD test.

[3]Like Sherman [65], we make two physical machines act as both CN and MN to save machine resources.

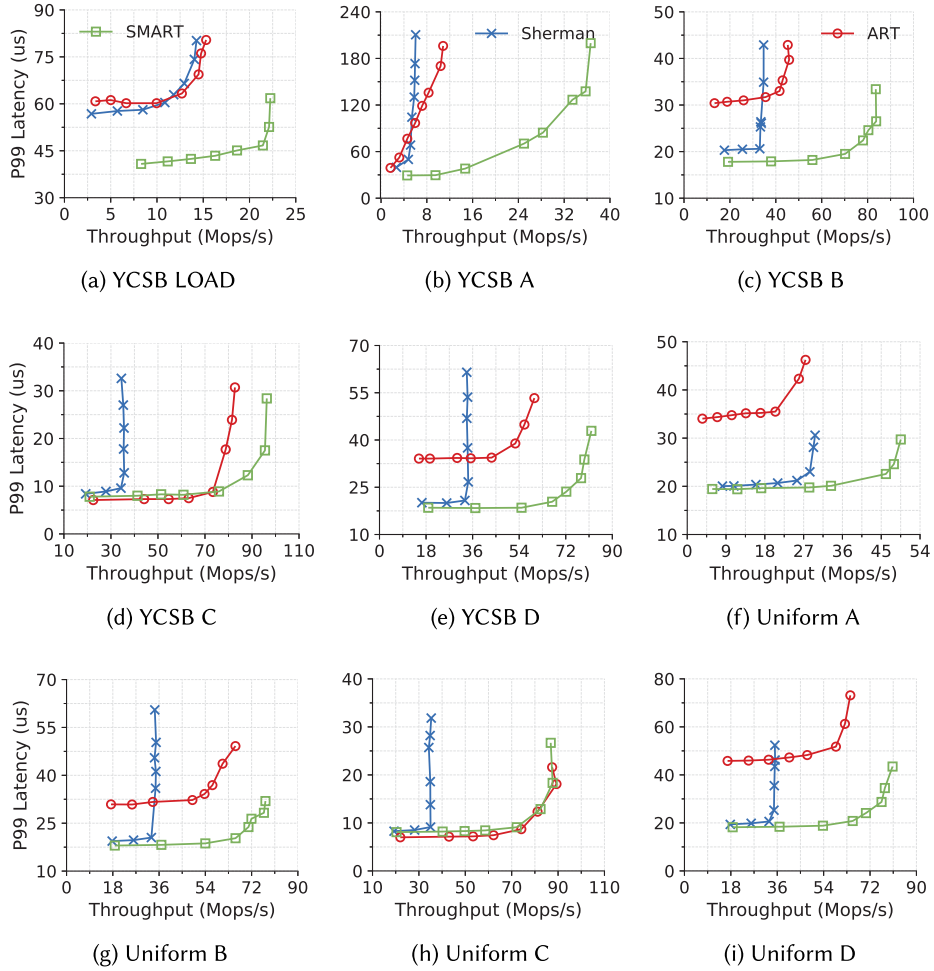


Fig. 14. The performance comparison of tree indexes on DM under YCSB and uniform workloads of integer keys, respectively.

5.1.3 Comparisons. We compare SMART with two state-of-the-art tree indexes, i.e., Sherman [65] and ART [38]. We use the default configuration of Sherman (e.g., a span size of 32 for long key) with all optimizations enabled (e.g., on-chip memory). Since ART is not designed for DM, we port it to DM by re-implementing it from scratch (as mentioned in Section 3), including its synchronization design (i.e., ROWEX [39]). For better baseline performance, we apply the HOCL of Sherman to ART and any other baselines of SMART. Coroutines are used in all three indexes to boost overall throughput.

5.2 Performance Comparison

Figures 14 and 15 present the throughput-latency curves of the three indexes with integer and string keys, respectively, using various numbers of clients (16 at least and 896 at most, evenly distributed across 16 CNs). Without loss of generality, we discuss the performance of integer keys in the following:

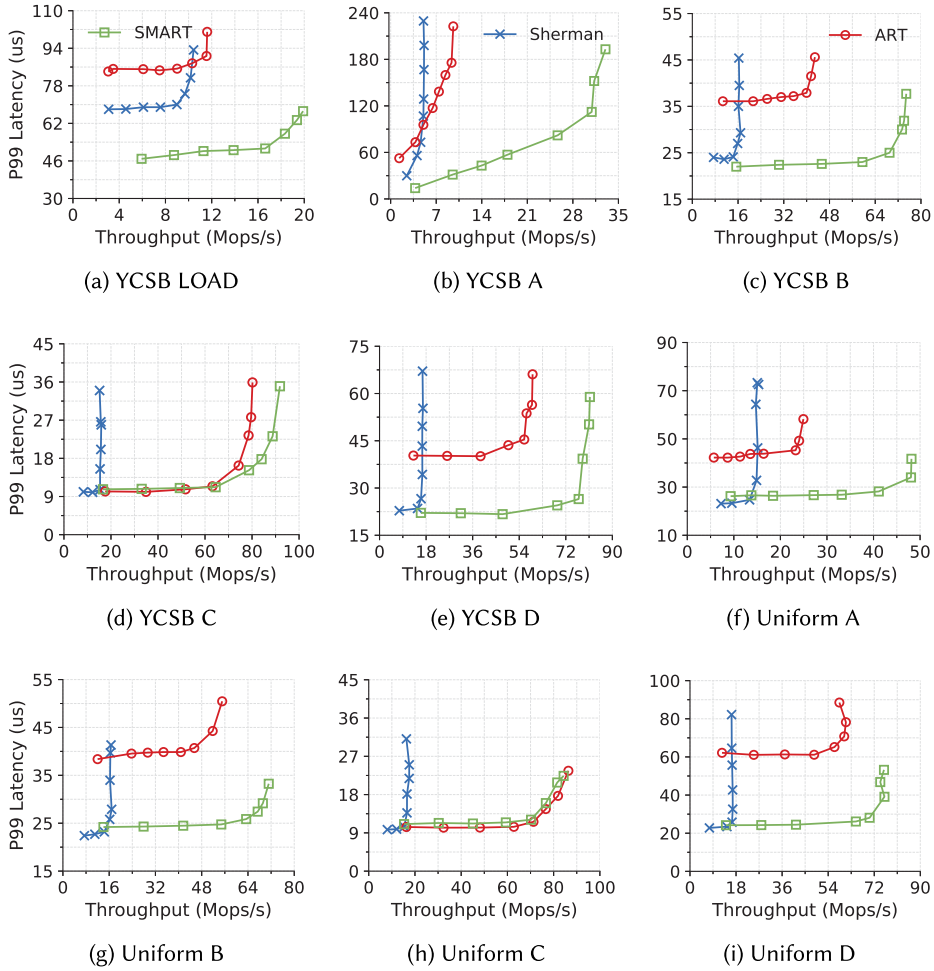


Fig. 15. The performance comparison of tree indexes on DM under YCSB and uniform workloads of string keys, respectively.

5.2.1 YCSB Workloads. The performances of the three tree indexes with the default six YCSB workloads are shown in Figures 14(a)–14(e).

Search-only workload (*YCSB C*). For the YCSB C workload, SMART outperforms Sherman by 2.8× due to no leaf read amplification, as mentioned in Section 3. Moreover, it outperforms ART by 1.2× due to the read delegation mechanism for reducing redundant I/Os. It is worth noting that SMART achieves up to 96 M requests per second, which breaks through the sum of the IOPS upper bounds of memory-side RNICs (about 90 Mops/s in total for two MNs, each equipped with one RNIC capable of 45 Mops/s). This is because the read delegation can perform concurrent duplicated reads with only one delegated read. In this experiment, the read delegation ratio (i.e., the number of concurrent reads being delegated) is 7.8%, with 48 clients on each CN. Besides, the similar P99 latency of SMART and ART shows that the read delegation causes near-zero overhead.

Insert workload (*YCSB LOAD, D*). For the YCSB LOAD workload, SMART outperforms Sherman and ART by 1.6×, 1.5× in throughput and achieves 1.4×, 1.5× lower P99 latency,

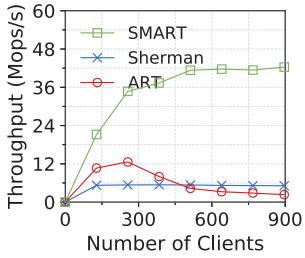


Fig. 16. The scalability of tree indexes under the YCSB A workload of integer keys.

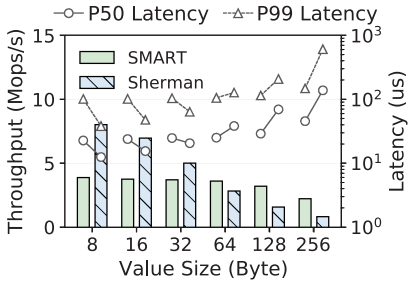


Fig. 17. The performance of scan under the YCSB E workload of integer keys with different value sizes.

respectively. This can be attributed to the design of the lock-free internal nodes. Specifically, both Sherman and ART have low throughput and high latency due to the node-grained locks, which introduce additional RTTs with frequent lock-fail retries, thus wasting the limited IOPS of RNICs in write-intensive scenarios (i.e., 50% insert). Interestingly, with string workloads, the latency of ART becomes much worse, since the smaller set of string partial keys (e.g., alphanumeric characters) aggravates concurrency conflicts.

For the YCSB D workload, SMART achieves 2.4× and 1.4× higher throughput and 1.1× and 1.8× lower P99 latency, compared with Sherman and ART, respectively. With fewer write conflicts (i.e., only 5% insert), read and write amplifications become the main reason for the poor performance of Sherman. ART still has a high tail latency, since concurrent writes cause cache misses, leading to remote tree traversals and thus continuous lock operations on the remote tree.

Update workload (*YCSB A, B*). Compared with Sherman and ART, SMART gains 6.1× and 3.4× improvement in throughput and 1.4× and 1.3× reduction in latency for YCSB A and achieves 2.4× and 1.8× higher throughput and 1.1× and 1.7× lower P99 latency for YCSB B, respectively.

Unlike the insert workload, YCSB A and B follow a Zipfian distribution of skewness 0.99, indicating a high amount of update concurrency conflicts. Consequently, Sherman performs poorly with YCSB A due to its coarse-grained, lock-based concurrency control. ART performs better than Sherman, since update operations do not modify the partial key fields and thus do not need to acquire locks. However, the out-of-place update scheme used by ART causes cache thrashing, resulting in huge cache-miss overhead and thus much higher latency than SMART. Note that the cache thrashing also impacts search performance, leaving a poor performance of ART on YCSB B (with only 5% update). As shown in Figure 16, ART experiences performance collapse with increasing clients due to severe cache thrashing. In contrast, SMART shows excellent scalability due to the cache-friendly in-place leaf node design and fine-grained concurrency control.

Scan workload (*YCSB E*). We evaluate the performance of scan operations with 128 clients using varying value sizes as shown in Figure 17. For a small value size (e.g., 8 bytes), SMART shows poorer performance than Sherman, since the small-sized leaf nodes saturate the memory-side IOPS upper bound, which is an inherent shortcoming of radix trees. However, for a value size larger than 64 bytes, which is common in real-world workload [4, 70], the scan performance of Sherman becomes worse than SMART, since the large-sized leaf nodes rapidly saturate the bandwidth bottleneck.

5.2.2 Uniform Workloads. As shown in Figures 14(f)–14(i), the performances of the three tree indexes with the uniform workloads are basically the same as those with the YCSB workloads, except for the following two workloads: ***(1) Uniform A.*** All three tree indexes show a much better performance with uniform A than YCSB A, since, with the uniform workload, the concurrency

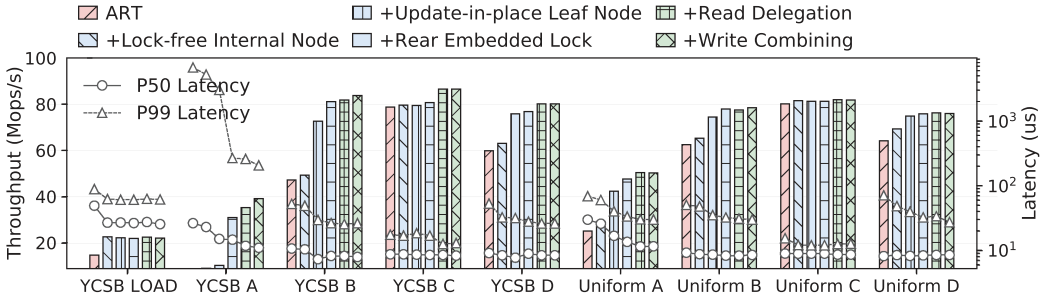


Fig. 18. The factor analysis of overall performance on SMART.

conflicts are significantly reduced. For the uniform A workload, SMART still outperforms Sherman and ART by 1.7× and 1.8× in throughput, respectively, and achieves 1.8× lower P99 latency than ART. ***(2) Uniform C.*** ART and SMART show a similar performance for the uniform C workload. This is because the read delegation in SMART does not benefit the throughput, as the clients on each CN rarely concurrently search for the same key under the uniform workload. Compared with Sherman, SMART still gains 2.5× improvement in throughput.

5.3 Factor Analysis for SMART Design

5.3.1 Main Design. Figure 18 presents the factor analysis on SMART. We start with the naive ART and apply each proposed technique one-by-one. We use 16 CNs (each launches 24 clients) and integer keys for experiments in this section.

+ Lock-free internal node. The lock-free internal nodes mainly contribute to the insert workload. With YCSB LOAD, it brings 1.5× improvement in throughput and 1.8×/1.4× reduction in P50/P99 latency. Unlike ROWEX, lock-free internal nodes eliminate expensive lock overhead during insertion and thus improve performance.

+ Update-in-place leaf node. In-place update scheme mainly contributes to the update workload. It achieves 1.5× improvement in throughput and 1.4×/1.7× reduction in P50/P99 latency with YCSB B. The in-place update scheme alleviates the cache coherence problem, as the addresses of the cached leaf nodes never expire until being deleted.

+ Rear embedded lock. The rear embedded locks further optimize the in-place update scheme. It eliminates the lock-releasing overhead, saving one RTT during each update. With YCSB A, it improves throughput by 3.0× and reduces tail latency by 11.3×.

+ Read delegation. The read delegation mechanism contributes to the search workload. It brings 1.1× throughput improvement and 1.3× tail latency reduction with YCSB C. It eliminates superfluous reads and thus saves network I/O consumption to support more client requests.

+ Write combining. The write combining mechanism mainly contributes to the write-intensive workload. It improves the throughput by 1.1× and reduces tail latency by 1.3× with YCSB A. Both the read delegation and the write combining bring much less improvement under uniform workloads than YCSB workloads, since there are few redundant I/Os with the uniform key distribution.

As the RDWC technique can reduce concurrency conflicts similar to HOCL, we compare their efficiency by applying them on SMART, respectively. As shown in Figure 19, when applying the primitive HOCL design, SMART shows poor performance with an average of 0.76 lock-fail retry count, due to the limited on-chip memory space (128 MB per RNIC in our evaluation) with only 2 MNs, which is insufficient for a large number of fine-grained locks. With E-HOCL (i.e., integrating the rear embedded lock technique into HOCL), SMART achieves much better performance

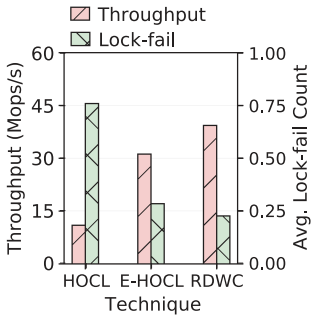


Fig. 19. The efficiency comparison of HOCL, E-HOCL, and RDWC under the YCSB A workload.

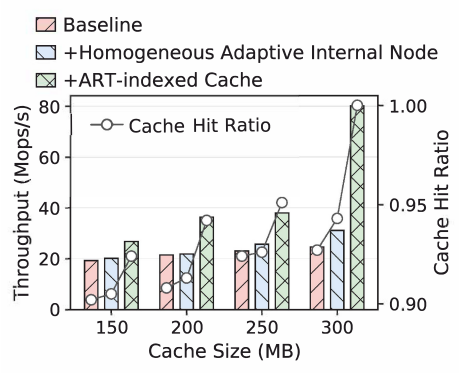


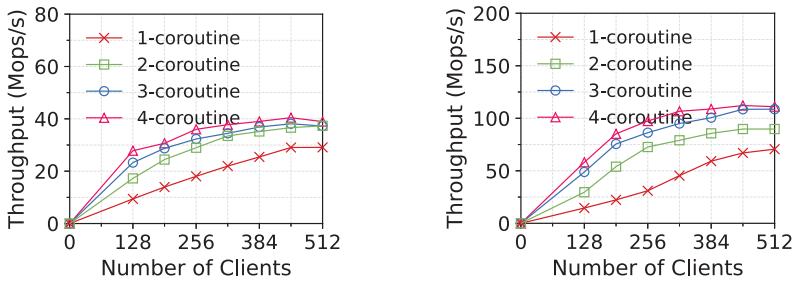
Fig. 20. The factor analysis of cache efficiency on SMART under the YCSB C workload of string keys with different cache sizes.

with an average of 0.29 lock-fail retry count. However, despite the optimization, HOCL still shows lower improvement efficiency than RDWC, which can introduce a 26.2% higher throughput. This is because RDWC saves not only the lock overhead but also the superfluous bandwidth consumption of reads and writes.

As the design of RDWC is transparent to the lower-level index structures, it will lead to the same amount of performance improvements on Sherman, i.e., 1.3× and 1.1× under write-intensive and read-only workloads (Figure 18). Therefore, after applying RDWC to Sherman, SMART can still achieve 4.7× (= 6.1/1.3) higher throughput under write-intensive workloads and 2.5× (= 2.8/1.1) higher throughput under read-only workloads.

5.3.2 Cache-related Techniques. Some cache-related techniques contribute to cache efficiency: ***(1) Homogeneous adaptive internal node.*** Due to the homogeneous adaptive internal node design, more fine-grained and flexible adaptive nodes are available, saving cache space with smaller sizes of cached nodes. ***(2) ART-indexed cache.*** Compared with a normal hash-based cache index, ART-indexed cache can efficiently save memory consumption of index keys without redundant key prefixes stored. As shown in Figure 20, after applying the above two techniques one-by-one, SMART achieves an increasing cache hit ratio and overall throughput under each specific limited cache size.

5.3.3 Coroutine-based Throughput Boost. We make each client thread execute multiple coroutine functions simultaneously to improve the throughput of SMART. To demonstrate the efficacy of coroutines, we evaluate SMART with each client executing different numbers of coroutine functions under the YCSB A and YCSB C workloads, as shown in Figures 21(a) and 21(b), respectively. With sufficient coroutines, SMART achieves 1.3× and 1.6× higher throughput under the YCSB A and C workloads, respectively, since the clients can schedule their coroutine functions to hide the busy-waiting overhead (e.g., RDMA polling). The improvement in SMART with YCSB C is higher than that with YCSB A, since the read-only workload consumes less network I/Os and thus leaves more improvement space for the coroutines to boost the throughput. Note that the improvement slows down when the number of coroutines exceeds two with YCSB A and three with YCSB C. This is because such numbers of coroutines are enough for each client thread in SMART to hide the busy-waiting overhead and saturate the limited memory-side IOPS.



(a) YCSB A (b) YCSB C

Fig. 21. The throughput of SMART with different numbers of coroutines for each client thread under the YCSB A and C workloads of integer keys, respectively.

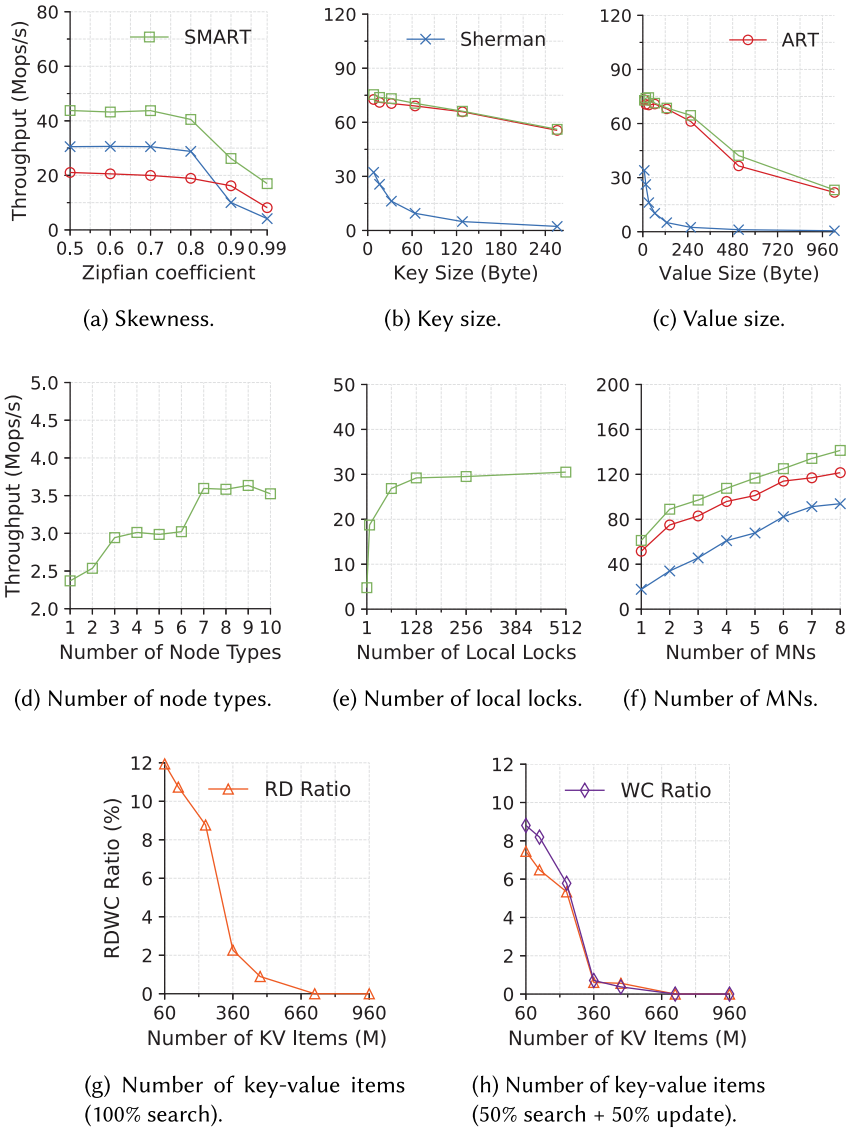
5.4 Sensitivity

In this section, we investigate how the workload skewness, key size, and value size affect the performance of SMART. Without explicit mention, we use 16 CNs with 16 clients each and integer keys for the sensitivity evaluation.

5.4.1 Skew Test. Figure 22(a) shows the performances of different tree indexes on a generated Zipfian workload [42] (50% search + 50% update) with various skewness. SMART performs best under both slightly and highly skewed workloads. Sherman shows a good performance in slightly skewed workloads, while having the poorest performance in highly skewed workloads because of its coarse-grained lock-based concurrency control design. ART performs better than Sherman in highly skewed workloads due to the lock-free RCU scheme but performs worst in slightly skewed workloads due to cache thrashing. Note that the RDWC in SMART does not benefit the overall throughput, since the network bandwidth is unsaturated. As the Zipfian skewness grows from 0.5 to 0.99, the performances of ART and SMART decrease by the same multiple (2.6×), and thus their performance gap is reduced. The performance of Sherman decreases by 7.4×, indicating the poor efficiency of coarse-grained lock-based design.

5.4.2 Impact of Key/Value Size. Figures 22(b) and 22(c) show the impact of key size and value size on the performances of the three tree indexes under YCSB C with sufficient caches. As the key size grows from 8 to 256 bytes, SMART and ART show a slight performance decline (1.3×), while Sherman experiences a rapid drop in performance (14×). As the value size grows from 8 to 1,024 bytes, the performance declines of SMART, ART, and Sherman are 3.1×, 3.4×, and 64×, respectively. This is because, during each search, Sherman needs to fetch the whole leaf node, whose size grows with key and value size, causing the rapidly increasing consumption of network bandwidth. On the contrary, SMART and ART only need to fetch the fine-grained small-sized leaf node. Thus, they are not bounded by the network bandwidth bottleneck, showing a stable performance with varying key sizes and value sizes. The performances of ART and SMART are close, since the read delegation in SMART does not benefit the throughput under the unsaturated network. This is consistent with the results shown in Figure 14(d).

5.4.3 Impact of Number of Node Types. As stated in Section 4.1.1, SMART proposes the *homogeneous adaptive internal node*. Thus, more fine-grained node types are available. Like ART [38], we make the distribution of the sizes of different node types approximately a geometric sequence from 4 to 256. Figure 22(d) shows the performance of SMART under YCSB C with increasing numbers of node types. As the number of node types increases from 1 (i.e., only NODE_256) to 10,



(a) Skewness. (b) Key size. (c) Value size.

(d) Number of node types. (e) Number of local locks. (f) Number of MNs.

(g) Number of key-value items (100% search).

(h) Number of key-value items (50% search + 50% update).

Fig. 22. The sensitivity analysis.

SMART achieves 1.5× higher throughput, since fine-grained node types can save more network bandwidth. Note that the throughput stops increasing when the number of node types exceeds 7. This is because the granularity of seven node types is already fine enough.

5.4.4 Impact of Number of Local Locks. As presented in Section 4.2.1, SMART adopts hash-based local locks on each CN to perform the RDWC technique. Figure 22(e) shows the performance of SMART with various numbers of local locks on each CN. As the number of local locks grows from 1 to 128, SMART achieves 6.1× improvement in throughput, since with more local locks, more keys can benefit from the RDWC technique. When the number of local locks exceeds 128,

the increase in the throughput slows down, since it is sufficient for 128 local locks to distinguish the target keys of the concurrent operations from the 16 clients on each CN.

5.4.5 Impact of Number of MNs. We use 16 CNs with 64 clients each and the YCSB C workload to evaluate the impact of the number of MNs on the performance. As shown in Figure 22(f), the performances of all three tree indexes increase as the number of MNs grows. This verifies that the performances of tree indexes on DM are all bottlenecked by the memory-side RNICs. Both SMART and ART perform better than Sherman, since they have smaller read amplification and thus are bottlenecked by the limited IOPS of RNICs rather than the bandwidth. The performance of SMART is higher than that of ART due to the read delegation technique, which works, since the 1,024 clients are enough to saturate the network.

5.4.6 Impact of Number of Key-value Items. We use 16 CNs with 64 clients each and two generated Zipfian workloads to evaluate the impact of the number of key-value items on the RDWC ratio of SMART. As shown in Figure 22(g), with 60 million key-value items, the read delegation ratio achieves 11.9% under the read-only workloads (100% search). As shown in Figure 22(h), with 60 million key-value items, the read delegation ratio and write combining ratio achieve 7.5% and 8.8%, respectively, under the write-intensive workload (50% search + 50% update). As the number of key-value items grows from 60 million to 960 million, both the read delegation and write combining ratios decrease. This is because, with a huge number of items, e.g., 720 million, there will be only a low chance of conflicting accesses even with a skewed key distribution. In this case, the RDWC technique is no longer needed to optimize the IOPS bottleneck, since there are few redundant I/Os and concurrency conflicts on DM.

6 RELATED WORK

6.1 Disaggregated Memory

The DM architecture is widely discussed in the literature [3, 9, 19, 22, 25, 32, 57], which is proposed to address the problem of a growing imbalance between computing and memory resources. Many recent academic works have been conducted on DM. LegoOS [56] designs a distributed operating system for disaggregated resource management. PolarDB Serverless [8] co-designs the database and DM to achieve better dynamic resource provisioning and faster failure recovery speed. Clover [64] explores an efficient manner to build a key-value store on disaggregated **persistent memory (PM)**, with careful designs between the data plane and the metadata/control plane. FUSEE [59] designs a fully memory-disaggregated key-value store that brings disaggregation to metadata management. Ditto [58] is an elastic and adaptive caching system on DM that can adjust resources in a resource-efficient and agile manner. The above works mainly focus on designing high-performance, resource-efficient storage systems on DM. SMART focuses on building a fast, scalable RDMA-based range index on DM for the above storage systems. In the following, we will introduce the related work on the RDMA-based indexes on DM.

6.2 RDMA-based Indexes

Attracted by the high performance of RDMA, there are increasing studies focusing on RDMA-based indexes [2, 49, 55, 65, 72]. Many studies conduct index operations via **remote procedure calls (RPCs)**, which is unsuitable for DM due to weak memory-side computing power. RACE [73] is an extendible RDMA-based hashing index on DM with lock-free remote concurrency control and efficient remote resizing. As a hashing index, it cannot support range queries like SMART. FG [72], designed as a B-link tree, is the first range index that completely leverages *one-side verbs* for write operations and thus supports DM. Sherman [65] is the state-of-the-art B+ tree index with

several RDMA-friendly software techniques. However, constrained by the structure of the B+ tree, both FG and Sherman suffer from low peak throughput and early latency deterioration due to read and write amplifications. To avoid such issues of B+ trees, SMART proposes that the radix tree is a more suitable tree index structure for DM. ROLEX [41] is a scalable RDMA-oriented learned index on DM. It mainly focuses on improving the cache efficiency of range indexes using learned models. Unlike ROLEX, SMART focuses on reducing the read and write amplifications of range indexes using the radix tree. Besides, extending RDMA interfaces is another approach to design tree indexes on DM, which offloads index write operations into memory-side NICs via SmartNICs or other customized hardware [2, 7, 15, 30, 43, 54, 60]. Different from these works, SMART is designed with commodity RNICs. To our knowledge, SMART is the first radix tree index on DM that achieves high performance with commodity RNICs.

7 CONCLUSION

Based on a thorough theoretical and experimental analysis of tree indexes built on DM, this article points out the performance bottleneck of B+ trees on DM due to severe read and write amplifications and then presents SMART, the first radix-tree-based index on DM. SMART addresses the challenges of applying ART on DM, including a hybrid concurrency control scheme to reduce lock overhead and avoid cache thrashing, a read-delegation and write-combining technique to reduce redundant I/Os, and a tailed cache validation mechanism. Our evaluation results show that SMART outperforms the state-of-the-art B+ tree on DM by up to 6.1× under write-intensive workloads and 2.8× under read-only workloads.

APPENDIX

A ARTIFACT APPENDIX

A.1 Abstract

The artifact provides the source code of SMART and automated scripts to reproduce all the experiment results in the article. The experiment results can show the superiority of ART on DM compared with the B+ tree and demonstrate the efficacy and efficiency of SMART we design.

A.2 Scope

A.2.1 Design Choices of the Radix Tree on DM. By reproducing the experiments of Figure 3, the artifact can validate that path compression and ART are both necessary for the radix tree on DM, and the pessimistic method is the best choice for path compression on DM.

A.2.2 Superiority of ART on DM. By reproducing the experiments of Figure 4, the artifact can validate that the radix tree is more suitable for DM than the B+ tree due to smaller read amplification under *read-only workloads*.

A.2.3 Challenges of ART on DM. By reproducing the experiments of Figure 5, the artifact can validate that ART suffers from significant challenges on DM under *hybrid read-write workloads*.

A.2.4 Efficacy and Efficiency of SMART. By reproducing the experiments of Figures 14–22, the artifact can validate that SMART can show better performance under YCSB workloads, compared with the state-of-the-art B+ tree on DM and a naive ART design.

A.3 Contents

A.3.1 Source Codes. The artifact contains source codes of SMART and the compared baselines (e.g., ART). Specifically, the source code of SMART contains the implementation of our three

key designs, i.e., the hybrid ART concurrency control scheme, the read-delegation and write-combining technique, and the reverse check mechanism for cache validation.

A.3.2 Automated Scripts. Except for Figures 14(f)–14(i), 15(f)–15(i), and the results with uniform workloads in Figure 18, which can be reproduced by manually changing the workloads, the artifact provides automated scripts to reproduce all the other experiment results in the article. Each figure has a Python script to automatically reproduce and visualize the experimental results.

A.4 Hosting

The artifact is available at https://github.com/dmemsys/SMART. Please use the *latest* commit version on the *extended-version* branch.

A.5 Requirements

The artifact is developed and tested using the r650 machines on CloudLab. Sixteen r650 machines are needed to reproduce all the results. Each r650 machine has two 36-core Intel Xeon CPUs, 256 GB of DRAM, and one 100 Gbps Mellanox ConnectX-6 RNIC. Each RNIC is connected to a 100 Gbps Ethernet switch.

A.6 Tutorial

A.6.1 Environment Setup. To set up the environment, please clone the source codes to the r650 machines. The necessary dependencies can be installed using our provided shell scripts in the artifact. Listing 1 shows the commands to set up the experiment environment.

```
# Get the source codes
git clone -b extended-version https://github.com/dmemsys/SMART
git clone https://github.com/River861/Sherman
# Set bash as the default shell
sudo su && chsh -s /bin/bash
# Install Mellanox OFED
cd SMART && sh ./script/installMLNX.sh
# Resize disk partition
sh ./script/resizePartition.sh
reboot
sudo su && resize2fs /dev/sda1
# Install libraries and tools
cd SMART && sh ./script/installLibs.sh
# Setup hugepages
echo 36864 > /proc/sys/vm/nr_hugepages
```

Listing 1. Commands to set up the environment.

A.6.2 Workloads Generation. The index microbench is used to generate YCSB workloads, including two key types, i.e., integer and string. Listing 2 shows the commands to generate the YCSB workloads to reproduce the results.

```
# Download YCSB source code
cd SMART/ycsb
sudo su && curl -O --location https://github.com/brianfrankcooper/YCSB
    /releases/download/0.11.0/ycsb-0.11.0.tar.gz
tar xfvz ycsb-0.11.0.tar.gz
mv ycsb-0.11.0 YCSB
# Download the email dataset
gdown --id 1ZJcQOuFI7IpAG6ZBgXwhjEeKO1T7Alzp
# Start to generate all the YCSB workloads
sh generate_full_workloads.sh
```

Listing 2. Commands to generate the YCSB workloads.

A.6.3 Results Reproduced. The artifact provides a single batch script to reproduce all the experiments with the YCSB workloads. This script should be run on a *master node*, which can directly establish SSH connections to other nodes of the r650 cluster. To reproduce the experiments, please set up the *home_dir* and *master_ip* values in *./exp/params/common.json*. Then, the script can be run. Listing 3 shows the commands. The reproduced results will be saved automatically.

```
sudo su && cd SMART/exp
# Run all the YSCB experiments
sh run_all.sh
```

Listing 3. Commands to start the experiments.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their constructive comments and suggestions.

REFERENCES

[1] Marcos Kawazoe Aguilera, Wojciech M. Golab, and Mehul A. Shah. 2008. A practical scalable distributed B-tree. *Proc. VLDB Endow.* 1, 1 (2008), 598–609. Retrieved from http://www.vldb.org/pvldb/vol1/1453922.pdf

[2] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, 120–126. DOI: https://doi.org/10.1145/3317550.3321433

[3] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. 2020. Disaggregation and the application. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX Association. Retrieved from https://www.usenix.org/conference/hotcloud20/presentation/angel

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. ACM, 53–64. DOI: https://doi.org/10.1145/2254756.2254766

[5] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. 2020. Caching with delayed hits. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. ACM, 495–513. DOI: https://doi.org/10.1145/3387514.3405883

[6] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A height optimized trie index for main-memory database systems. In *Proceedings of the International Conference on Management of Data*. ACM, 521–534. DOI: https://doi.org/10.1145/3183713.3196896

[7] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. 2021. PRISM: Rethinking the RDMA interface for distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. ACM, 228–242. DOI: https://doi.org/10.1145/3477132.3483587

[8] 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: A cloud native database for disaggregated data centers. In *Proceedings of the International Conference on Management of Data*. ACM, 2477–2489. DOI: https://doi.org/10.1145/3448016.3457560

[9] Amanda Carbonari and Ivan Beschastnikh. 2017. Tolerating faults in disaggregated datacenters. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM, 164–170. DOI: https://doi.org/10.1145/3152434.3152447

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 143–154. DOI: https://doi.org/10.1145/1807128.1807152

[11] Intel Corporation. (1998). Write Combining Memory Implementation Guidelines. Retrieved from https://download.intel.com/design/PentiumII/applnots/24442201.pdf

[12] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2 (2009), 6:1–6:31.DOI: https://doi.org/10.1145/1462166.1462167

[13] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 401–414. Retrieved from https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87

[14] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 1–14. Retrieved from https://www.usenix.org/conference/atc19/presentation/duplyakin

[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 *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 51–66. Retrieved from https://www.usenix.org/conference/nsdi18/presentation/firestone

[16] Fonxat. 2018. 300 Million Email Database. Retrieved from https://archive.org/details/300MillionEmailDatabase

[17] Keir Fraser. 2004. *Practical Lock-freedom*. Ph. D. Dissertation. University of Cambridge, UK. Retrieved from https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193

[18] Alexander Fuerst, Stanko Novakovic, Iñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. 2022. Memory-harvesting VMs in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 583–594. DOI: https://doi.org/10.1145/3503222.3507725

[19] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 249–264. Retrieved from https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao

[20] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over commodity ethernet at scale. In *Proceedings of the ACM SIGCOMM Conference*. ACM, 202–215. DOI: https://doi.org/10.1145/2934872.2934908

[21] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. 2022. Clio: A hardware-software co-designed disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 417–433. DOI: https://doi.org/10.1145/3503222.3507762

[22] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. 2013. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the 12th ACM Workshop on Hot Topics in Networks*. ACM, 10:1–10:7. DOI: https://doi.org/10.1145/2535771.2535778

[23] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.* 67, 12 (2007), 1270–1285. DOI: https://doi.org/10.1016/j.jpdc.2007.04.010

[24] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: Coroutine-oriented main-memory database engine. *Proc. VLDB Endow.* 14, 3 (2020), 431–444. Retrieved from http://www.vldb.org/pvldb/vol14/p431-he.pdf

[25] Eric Hooper. 2018. Intel Rack Scale Design: Just What Is It? Retrieved from https://www.datacenterdynamics.com/en/opinions/intel-rack-scale-design-just-what-is-it

[26] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. 2005. iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. *ACM Trans. Datab. Syst.* 30, 2 (2005), 364–397. DOI: https://doi.org/10.1145/1071610.1071612

[27] Minwen Ji, Alistair C. Veitch, and John Wilkes. 2003. Seneca: Remote mirroring done write. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference*. USENIX, 253–268. Retrieved from http://www.usenix.org/events/usenix03/tech/ji.html

[28] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design guidelines for high performance RDMA systems. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 437–450. Retrieved from https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[29] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2019. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 1–16. Retrieved from https://www.usenix.org/conference/nsdi19/presentation/kalia

[30] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostic, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. ACM, 756–771. DOI: https://doi.org/10.1145/3477132.3483565

[31] Wook-Hee Kim, Madhava Krishnan Ramanathan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A high performance persistent range index using PAC guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. ACM, 424–439. DOI: https://doi.org/10.1145/3477132.3483589

[32] HP Labs. 2014. The Machine: A New Kind of Computer. Retrieved from https://www.hpl.hp.com/research/systems-research/themachine

[33] Leslie Lamport. 1977. Concurrent reading and writing. *Commun. ACM* 20, 11 (1977), 806–811. DOI: https://doi.org/10.1145/359863.359878

[34] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th International Symposium on Computer Architecture*. ACM, 2–13. DOI: https://doi.org/10.1145/1555754.1555758

[35] Seung-Seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-network memory management for disaggregated data centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. ACM, 488–504. DOI: https://doi.org/10.1145/3477132.3483561

[36] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write optimal radix tree for persistent memory storage systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*. USENIX Association, 257–270. Retrieved from https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon

[37] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, 462–477. DOI: https://doi.org/10.1145/3341301.3359635

[38] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering*. IEEE Computer Society, 38–49. DOI: https://doi.org/10.1109/ICDE.2013.6544812

[39] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*. ACM, 3:1–3:8. DOI: https://doi.org/10.1145/2933349.2933352

[40] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 574–587. DOI: https://doi.org/10.1145/3575693.3578835

[41] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A scalable RDMA-oriented learned key-value store for disaggregated memory systems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*. USENIX Association, 99–114. Retrieved from https://www.usenix.org/conference/fast23/presentation/li-pengfei

[42] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 429–444. Retrieved from https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[43] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading distributed applications onto SmartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*. ACM, 318–333. DOI: https://doi.org/10.1145/3341302.3342079

[44] Qixiao Liu and Zhibin Yu. 2018. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from Alibaba Trace. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 347–360. DOI: https://doi.org/10.1145/3267809.3267830

[45] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query optimized persistent ART. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*. USENIX Association, 1–16. Retrieved from https://www.usenix.org/conference/fast21/presentation/ma

[46] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the European Conference on Computer Systems*. ACM, 183–196. DOI: https://doi.org/10.1145/2168836.2168855

[47] Ajit Mathew and Changwoo Min. 2020. HydraList: A scalable in-memory index using asynchronous updates and partial replication. *Proc. VLDB Endow.* 13, 9 (2020), 1332–1345. Retrieved from http://www.vldb.org/pvldb/vol13/p1332-mathew.pdf

[48] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 103–114. Retrieved from https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[49] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. 2016. Balancing CPU and network in the cell distributed B-tree store. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 451–464. Retrieved from https://www.usenix.org/conference/atc16/technical-sessions/presentation/mitchell

[50] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. 2021. Birds of a feather flock together: Scaling RDMA RPCs with flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. ACM, 212–227. DOI: https://doi.org/10.1145/3477132.3483576

[51] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. 2018. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the 13th EuroSys Conference*. ACM, 16:1–16:12. DOI: https://doi.org/10.1145/3190508.3190537

[52] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 7. DOI: https://doi.org/10.1145/2391229.2391236

[53] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-performance, application-integrated far memory. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 315–332. Retrieved from https://www.usenix.org/conference/osdi20/presentation/ruan

[54] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. 2021. Xenic: SmartNIC-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. ACM, 740–755. DOI: https://doi.org/10.1145/3477132.3483555

[55] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 2019. Fast general distributed transactions with opacity. In *Proceedings of the International Conference on Management of Data*. ACM, 433–448. DOI: https://doi.org/10.1145/3299869.3300069

[56] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 69–87. Retrieved from https://www.usenix.org/conference/osdi18/presentation/shan

[57] Yizhou Shan, Will Lin, Ryan Kosta, Arvind Krishnamurthy, and Yiying Zhang. 2021. Disaggregating and consolidating network functionalities with SuperNIC. *CoRR* abs/2109.07744 (2021).

[58] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R. Lyu. 2023. Ditto: An elastic and adaptive memory-disaggregated caching system. In *Proceedings of the 29th Symposium on Operating Systems Principles*. ACM, 675–691. DOI: https://doi.org/10.1145/3600006.3613144

[59] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A fully memory-disaggregated key-value store. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*. USENIX Association, 81–98. Retrieved from https://www.usenix.org/conference/fast23/presentation/shen

[60] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: Smart remote memory. In *Proceedings of the 15th EuroSys Conference*. ACM, 29:1–29:16. DOI: https://doi.org/10.1145/3342195.3387519

[61] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. 2010. Extending SSD lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*. USENIX, 101–114. Retrieved from http://www.usenix.org/events/fast10/tech/full_papers/soundararajan.pdf

[62] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. RFP: When RPC is faster than server-bypass with RDMA. In *Proceedings of the 12th European Conference on Computer Systems*. ACM, 1–15. DOI: https://doi.org/10.1145/3064176.3064189

[63] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next generation. In *Proceedings of the 15th EuroSys Conference*. ACM, 30:1–30:14. DOI: https://doi.org/10.1145/3342195.3387517

[64] 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 *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 33–48. Retrieved from https://www.usenix.org/conference/atc20/presentation/tsai

[65] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In *Proceedings of the International Conference on Management of Data*. ACM, 1033–1048. DOI: https://doi.org/10.1145/3514221.3517824

[66] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. 2021. Concordia: Distributed shared memory with in-network cache coherence. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*. USENIX Association, 277–292. Retrieved from https://www.usenix.org/conference/fast21/presentation/wang

[67] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a BW-Tree takes more than just buzz words. In *Proceedings of the International Conference on Management of Data*. ACM, 473–488. DOI: https://doi.org/10.1145/3183713.3196895

[68] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based ordered key-value store using remote learned cache. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 117–135. Retrieved from https://www.usenix.org/conference/osdi20/presentation/wei

[69] Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. 2010. Efficient B-tree based indexing for cloud data processing. *Proc. VLDB Endow.* 3, 1 (2010), 1207–1218. Retrieved from http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/R107.pdf

[70] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 191–208. Retrieved from https://www.usenix.org/conference/osdi20/presentation/yang

[71] 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. DOI: https://doi.org/10.1145/3589276

[72] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing distributed tree-based index structures for fast RDMA-capable networks. In *Proceedings of the International Conference on Management of Data*. ACM, 741–758. DOI: https://doi.org/10.1145/3299869.3300081

[73] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-conscious extendible hashing for disaggregated memory. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 15–29. Retrieved from https://www.usenix.org/conference/atc21/presentation/zuo

Received 31 October 2023; revised 26 January 2024; accepted 30 March 2024