ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory

Xuchuan Luo[1,*], Pengfei Zuo[2], Jiacheng Shen[3,*], Jiazhen Gu[3], Xin Wang[1,4], Michael R. Lyu[3], and Yangfan Zhou[1,4]

[1]*School of Computer Science, Fudan University*
[2]*Huawei Cloud* [3]*The Chinese University of Hong Kong*
[4]*Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China*

Abstract

Disaggregated memory (DM) is an increasingly prevalent architecture in academia and industry 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 paper, we propose to use the radix tree, which 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, compared with state-of-the-art B+ trees on DM.

1 Introduction

Distributed range indexes are fundamental building blocks of many applications, *e.g.*, databases and key-value stores, to conduct range queries [2, 21, 53, 57, 59]. To improve resource utilization, many new proposals adopt the disaggregated memory (DM) architecture [53, 59]. 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, 16, 19, 20, 27, 47]. In this way, a DM range indexing system can utilize resources more efficiently.

Current DM index systems [53, 59] 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 which contain many useless keys and pointers since only one key is the target. This inevitably amplifies the network bandwidth consumption. As such network bandwidth is generally the bottleneck of the DM architecture [23], 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 [53], the state-of-the-art B+ tree index on DM. The throughput is 10.8× lower than the theoretical bound of RNICs under the YCSB workloads [10].

In this paper, 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 [32], 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 [33], 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

*The work was mainly conducted when Xuchuan and Jiacheng were interns at Huawei Cloud.

in the memory pool of DM have bounded IOPS (I/O per second) [51]. 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 [56]. 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 to achieve high performance without 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 [53], 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 paper 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

As shown in Figure 1, the DM architecture physically separates computing (*e.g.*, CPUs) and memory (*e.g.*, DRAM) resources into two independent resource pools to address the resource utilization issue in traditional data centers with monolithic servers [18, 31, 42, 43, 46, 54]. In the DM architecture, compute nodes (CNs) own powerful computing resources but

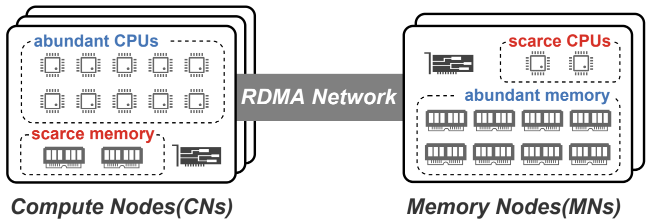


Figure 1: The architecture of disaggregated memory.

only have a small piece of memory serving as local caches. In contrast, memory nodes (MNs) 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.

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 [12, 17]. RDMA network interface cards (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 [59] and Sherman [53]. 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 [53] 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.*, coarse-grained lock-based concurrency control and inherent read amplification, which are analyzed in Section 3.

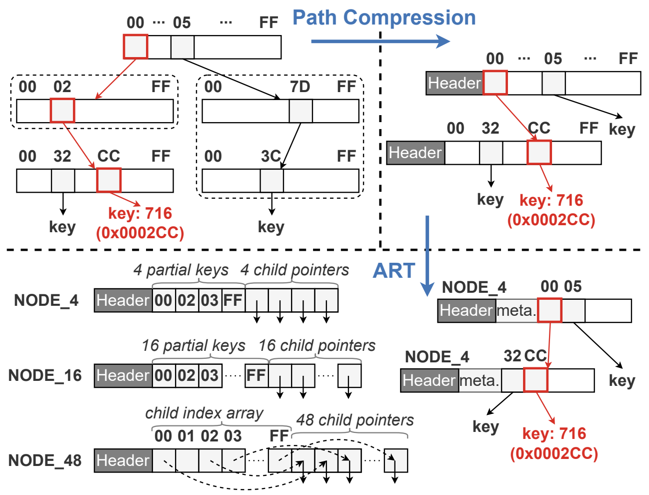


Figure 2: The optimization process from the basic radix tree to ART. For clarity, hexadecimal partial keys are shown. NODE_256 is simply an array of 256 pointers, which is not shown due to space limitation.

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 [32]: ***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.

Adaptive radix tree (ART). ART [32] 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 [32], 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 [33]. There are some proposed ART-based indexes designed on monolithic servers [26, 29, 30, 37], while none of them is designed for DM.

Table 1: Read and write amplification factors of different trees.

	ART	B+ Tree	Sherman
Read	$\frac{M_1+E}{E}=1.10$	$\frac{M_2+S\cdot E}{E}=32.7$	$\frac{M_2+S\cdot(M_3+E)}{E}=33.0$
Write	$\frac{M_1+E}{E}=1.10$	$\frac{M_2+S\cdot E}{E}=32.7$	$\frac{M_3+E}{E}=1.01$

3 Analysis of Tree Indexes Built on DM

In this section, we first theoretically and experimentally compare B+ trees with a vanilla ART (§ 3.1). We then present the challenges of designing ART on DM (§ 3.2).

All the experiments in this section are conducted with 8 CNs and 1 MN, each equipped with a 100Gbps Mellanox ConnectX-6 RNIC. Each CN launches 32 clients with one shared 600MB cache. We use YCSB workloads [10] (including 60 million entries) with 32-byte string keys and 64-byte values, which is typical in real-world workloads [4, 58].

3.1 Motivations: B+ Tree vs. ART on DM

The main problem of B+ trees on DM is their severe read and write amplifications. 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 [53]). We then experimentally show the performance impacts due to the read amplification.

3.1.1 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 of the B+ tree node. E denotes the key-value item size.

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 the same size of the key-value item, *i.e.*, 96 bytes, 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 ($2\cdot 32B$), a valid bit, a lock bit, a 1-byte level field, and two 7-bit versions [53], *i.e.*, 67 bytes in total. We use the

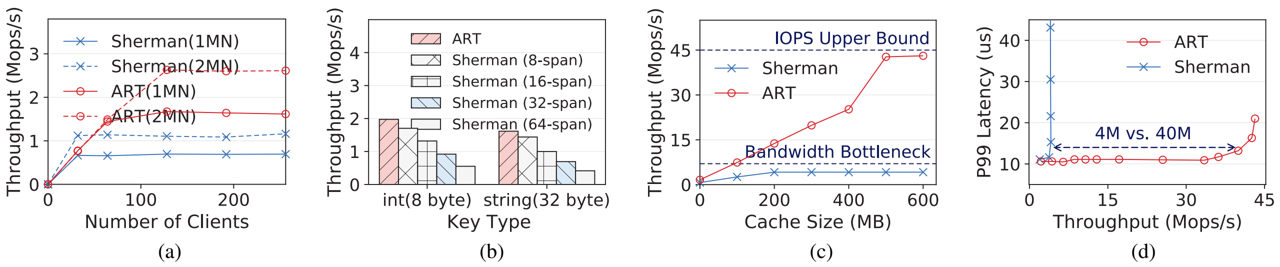


Figure 3: The read performances of Sherman and ART under the YCSB C workload (100% read). (a) The throughput bottleneck with no cache. (b) The impact of key size and span size with no cache. (c) The peak throughput with various sizes of caches. (d) The latency deterioration with excess requests.

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 $\frac{M_3+E}{E} = \frac{1B+96B}{96B} = 1.01$.

3.1.2 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 [23]. 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 3a, 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 3b, the larger the key size or the span size (*i.e.*, the number of keys stored in a leaf node) 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 3c, 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 § 3.1.1.

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 3d, 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.

3.2 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 [33]. However, lock operations are expensive on DM and lead to poor write performance, as shown in Figure 4a. 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

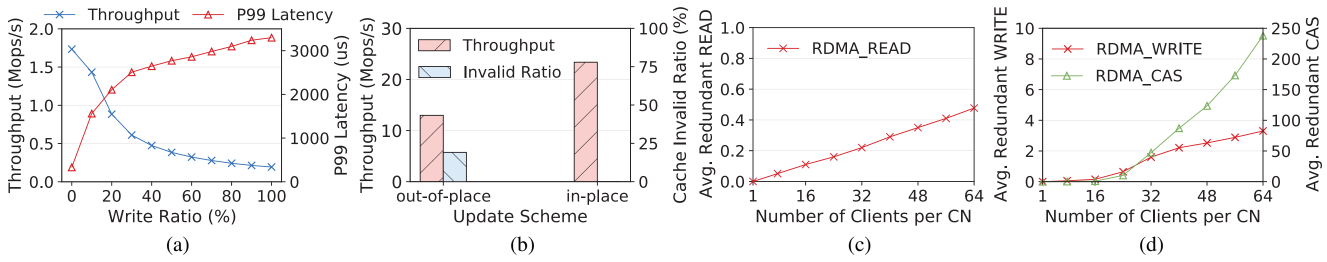


Figure 4: (a) The write performance of ART under the YCSB insert workload (100% insert) with no cache. (b) The performance degradation caused by cache thrashing under the YCSB A workload (50% read + 50% update) with sufficient caches. (c-d) The inter-client redundant I/Os on DM in terms of reads and writes.

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 4b. The out-of-place scheme brings about an average of 19.1% invalid cached 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 1KB 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 4c, 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.

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

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 4d, 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 § 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, 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 5 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 (§ 4.1). To save the limited IOPS of RNICs (**Challenge 2**), we propose a *read-delegation and write-combining (RDWC)* technique to eliminate inter-client redundant I/Os (§ 4.2). To handle the cache validation (**Challenge 3**), we co-design SMART with an *ART cache* (§ 4.3), including a reverse check

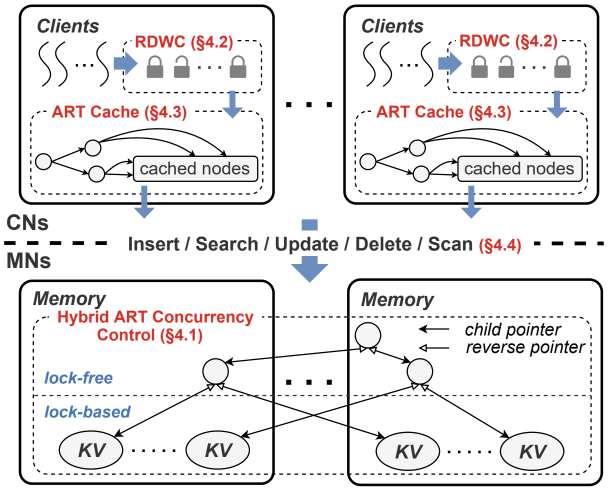
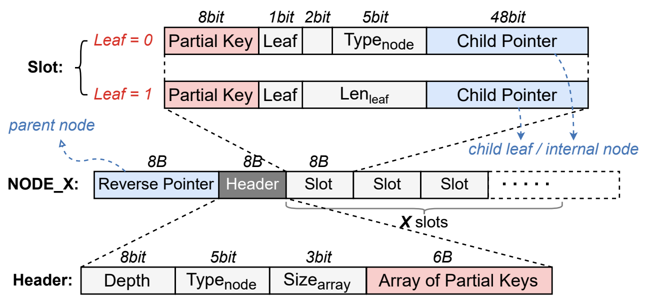
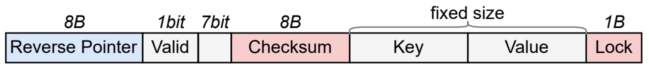


Figure 5: The overview of SMART.



(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.

Figure 6: 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.

mechanism. Lastly, we summarize the operations (*i.e.*, insert, search, update, delete, scan) that SMART supports (§ 4.4).

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 6a, 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 § 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 (§ 4.5). 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.

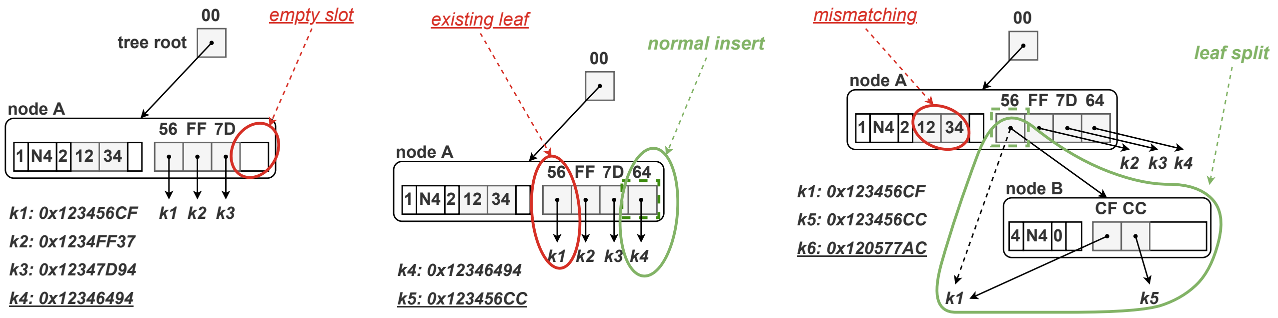
(2) Pessimistic 8-byte header of the internal node. We choose the pessimistic method for path compression since both the optimistic and hybrid methods need two tree traversals to insert a nonexistent key. One entire tree traversal is required to search for the nonexistent key since not all compressed partial keys are stored in the header. The other traversal executes the actual insertion. In contrast, the pessimistic method can insert the nonexistent key through one traversal.

Besides, following previous designs [29, 33, 37], we fix the header size to 8 bytes, which can be changed atomically. If some partial keys overflow from the header, we store them in an empty following node. Although this may increase the tree height, we mitigate this with the help of cache (§ 4.3).

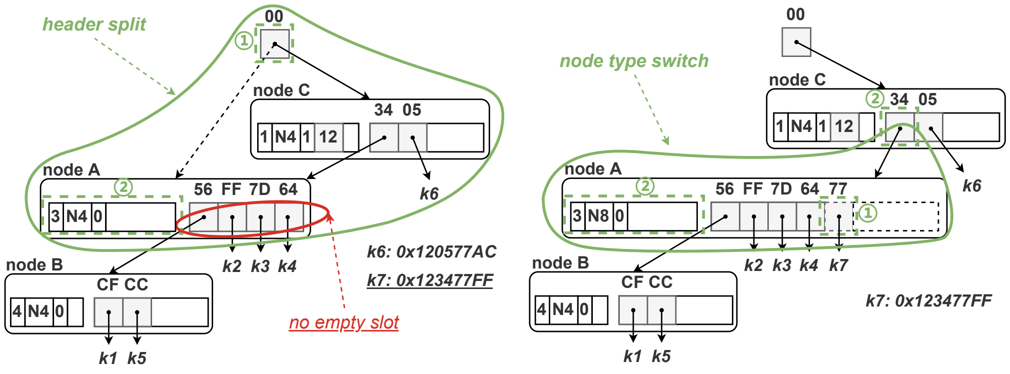
As shown in Figure 6a, a header consists of an 8-bit $Depth$ field, a 5-bit $Type_{node}$ field, a 3-bit $Size_{array}$ field, and a 6-byte array of partial keys. The $Depth$ field indicates the start position for matching the target key. The $Type_{node}$ field is used for cache validation, which will be illustrated in § 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-



(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***.

Figure 7: 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.

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 [40, 53], 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, the reader conducts a re-read.

(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 [12].

As shown in Figure 6b, 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 § 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 7. Except for the in-place leaf update, all the sub-operations are lock-free. The complete operation process will be described in § 4.4.

Normal insert. During an insert, the target partial key may not be in the internal node yet. As shown in Figure 7b, 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, 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, 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, a leaf split is needed as shown in Figure 7c. 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, the client continues to traverse following the return pointer.

Header split. If a mismatching for in-header partial keys is found, a header split is required as shown in Figure 7d. 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, 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 (§ 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 7e. 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, 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 [23].

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), a doorbell batching is conducted to reduce PCIe overhead.

4.2 Read Delegation and Write Combining

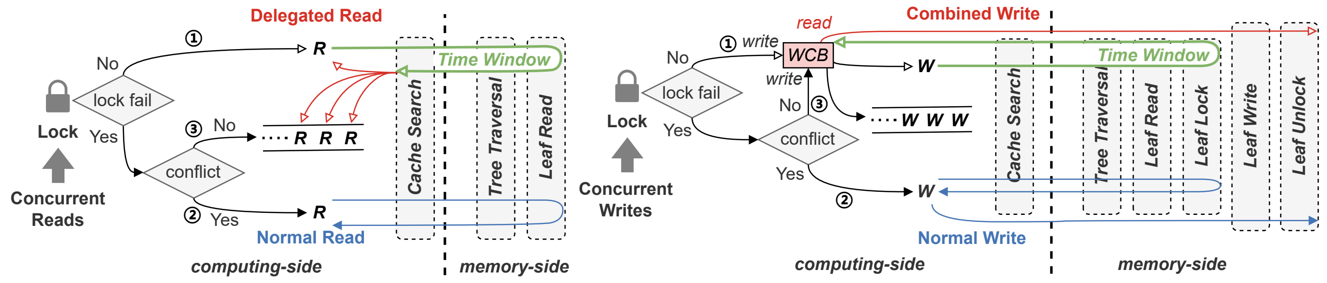
SMART proposes the read-delegation and write-combining (RDWC) technique on DM to eliminate inter-client redundant I/Os in terms of reads and writes, respectively, to break through the IOPS upper bound.

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 [53]. 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, 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.

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].



(a) Read delegation.

(b) Write combining.

Figure 8: The processes of the read delegation and the write combining on SMART respectively.

We implement this as shown in Figure 8a. 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, 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.

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 so as to save the system bus bandwidth. This idea, also known as write coalescing, is applied to many storage systems [22, 28, 50]. 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 so as to save the network bandwidth and the limited IOPS of RNICs.

We implement WC on DM as shown in Figure 8b. 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 a 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, 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).

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, 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. We use the same time window for read-delegation and write-combining to address this issue. In this way, 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 readers and writers operating on the same key fairly acquire the same local lock, where the winner decides the time window. Each local lock is associated with two waiting queues, *i.e.*, a read queue and a write queue, so as to conduct read delegation and write combining exclusively and concurrently. In our implementation, 4M 32-bit local locks are sufficient on each CN, consuming only nearly 3% of cache size.[2]

4.3 ART Cache

ART-indexed 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. As shown in Figure 9, 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).

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

[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.

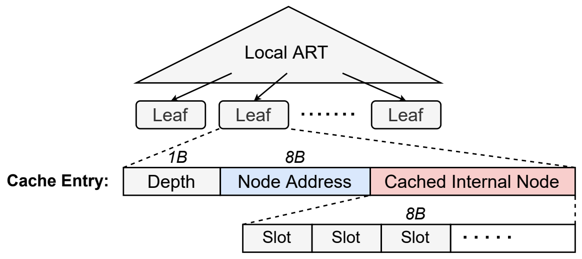


Figure 9: The structure of the ART cache.

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 6a (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 § 4.5.

For slot delete, this type of cache invalidation corresponds to Type 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 6. If the client reads a remote node according to a cached pointer, it checks whether the reverse address is equal to the node address in the cache entry. If not, 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 6a. If the client reads a remote node according to a cached pointer, 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, 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 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, start the traversal from the tree root slot.

Search. 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, invalidate the cache entry and retry this search. 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. 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, read the next node along the child pointer in the slot and repeat the process. Otherwise, the target item does not exist.

Insert/Update. The client first reads the node and conducts a ***reverse check*** like the search. After that, as for a leaf node, if its key is the same as the target key, execute an ***in-place leaf update***. Otherwise, a ***leaf split*** is needed. As for an internal node, if a mismatching for the in-header partial keys is found, conduct a ***header split***. Otherwise, turn to search among the slots. If the current target partial key can be found in a slot, read the next node along the corresponding child pointer in the slot and start the process again. Otherwise, conduct a ***normal insert*** with the next empty pointer slot. If no empty slot can be found, a ***node type switch*** is needed.

Delete. Delete operations have a similar process as insert operations. A normal delete clears the slot pointing to the target leaf node via RDMA_CAS and unsets the *Valid* bit of the deleted leaf node. Opposite operations of ***leaf split*** and ***header split*** are conducted for path compression.

Scan. At each level of traversal, the client conducts parallel RDMA_READs to fetch all nodes inside the target key range. 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. Like many other existing tree indexes [53, 59] on DM, SMART does not guarantee the scan is atomic with concurrent insert or update operations.

4.5 Discussion

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

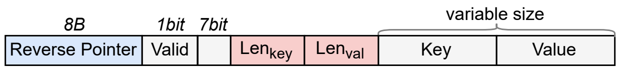


Figure 10: The structure of the variable-sized leaf node.

variable-sized key-value items are the same as that on fixed-sized ones.

As for the leaf node structure, SMART can follow the design in RACE [60]. As shown in Figure 10, 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.

Moreover, the cache validation mechanism (§ 4.3) 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, the cached slot is invalid.

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.

The first lock-free ART design. A pure lock-free ART can be formed with the lock-free node design in Figure 6a 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

Testbed. We run all experiments on 16 physical machines (16 CNs and 2 MNs)[3] on the Clemson cluster of CloudLab [13]. Each machine has two 36-core Intel Xeon CPUs, 256GB of DRAM, and one 100Gbps Mellanox ConnectX-6 IB RNIC. Each RNIC is connected to a 100Gbps Ethernet switch. Each MN owns 64GB DRAM and 1 CPU core for network connection and memory allocation. Each CN owns 4GB 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 [12].

[3]Like Sherman [53], we make two physical machines act as both CN and MN to save machine resources.

Workloads. Without explicit mention, we use the index microbench [55] to generate YCSB [10] workloads like previous work [6,26,39]. We evaluate SMART with 6 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. For most workloads, we test 2 key types, *i.e.*, integer (8-byte) and string (32-byte). For string workloads, we use 125 million publicly available email addresses [15] and conduct a common pre-processing (*i.e.*, swap username and domain fields of email addresses) like previous work [32,38,39,55]. We use 8-byte values consistent with prior work [6,24,38,41,53,56]. For each workload, we populate 60 million keys before conducting 60 million operations, except for the LOAD test.

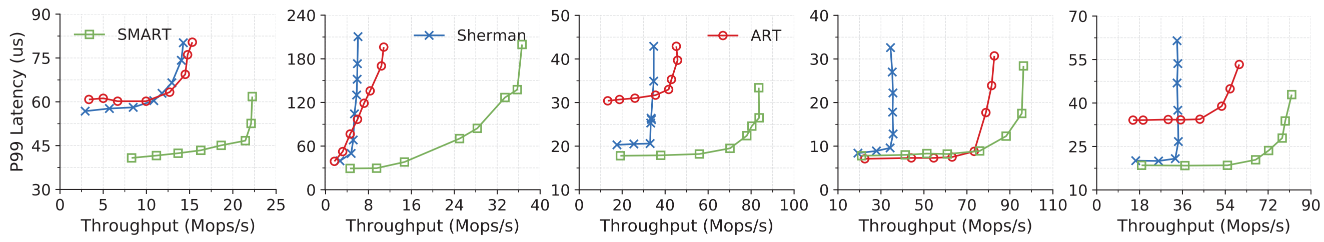
Comparisons. We compare SMART with two state-of-the-art tree indexes, *i.e.*, Sherman [53] and ART [32]. We use the default configuration of Sherman (*e.g.*, a span size of 32 for long key) with all optimizations enabled (*e.g.*, on-chip memory). Since ART is not designed for DM, we port it to DM by re-implementing it from scratch (as mentioned in § 3), including its synchronization design (*i.e.*, ROWEX [33]). For better baseline performance, we apply the HOCL of Sherman to ART and any other baselines of SMART. Coroutines are used in each client to hide RDMA polling overhead.

5.2 Performance Comparison

Figures 11 and 12 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.

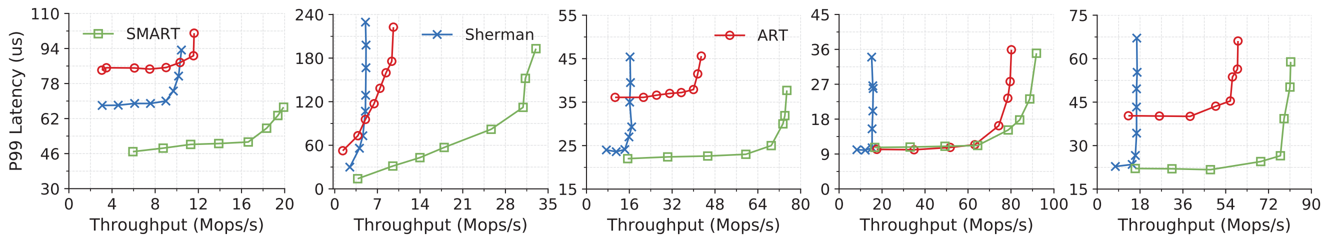
Search-only workload (*YCSB C*). For the YCSB C workload, SMART outperforms Sherman by 2.8× due to no leaf read amplification, as mentioned in § 3. Moreover, it outperforms ART by 1.2× due to the read delegation mechanism for reducing redundant I/Os. It is worth noting that SMART achieves up to 96M requests per second, which breaks through the total IOPS upper bound of memory-side RNICs (about 90 Mops in total with the two MNs). This is because the read delegation can perform concurrent duplicated reads with only one delegated read. 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 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-



(a) YCSB LOAD (b) YCSB A (c) YCSB B (d) YCSB C (e) YCSB D

Figure 11: The performance comparison of tree indexes on DM under YCSB workloads of integer keys.



(a) YCSB LOAD (b) YCSB A (c) YCSB B (d) YCSB C (e) YCSB D

Figure 12: The performance comparison of tree indexes on DM under YCSB workloads of string keys.

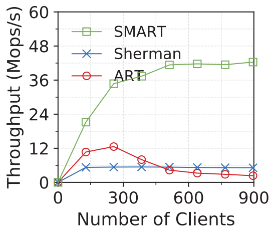


Figure 13: The scalability of tree indexes under the YCSB A workload of integer keys.

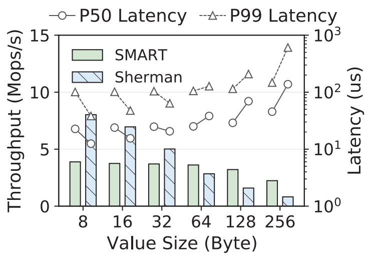


Figure 14: The performance of scan under the YCSB E workload of integer keys with different value sizes.

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 13, 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 14. 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 [4,58], the scan performance of Sherman becomes worse than SMART since the large-sized leaf nodes rapidly saturate the bandwidth bottleneck.

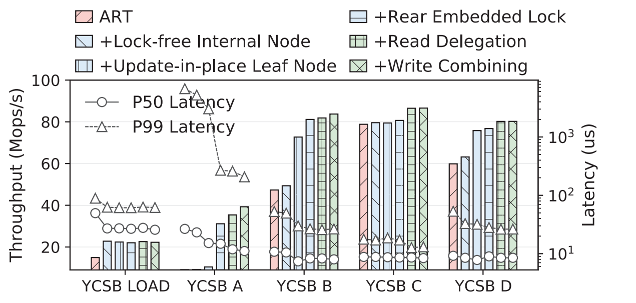


Figure 15: The factor analysis of overall performance on SMART.

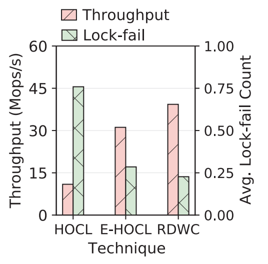


Figure 16: The efficiency comparison of HOCL, E-HOCL and RDWC under the YCSB A workload.

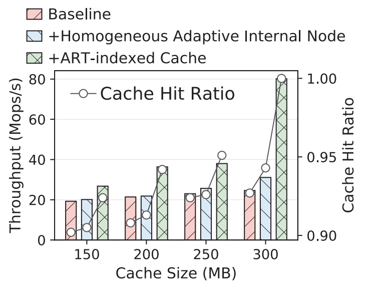


Figure 17: The factor analysis of cache efficiency on SMART under the YCSB C workload of string keys with different cache sizes.

5.3 Factor Analysis for SMART Design

Figure 15 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, so as 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.

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 16, 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 (128MB 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 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.

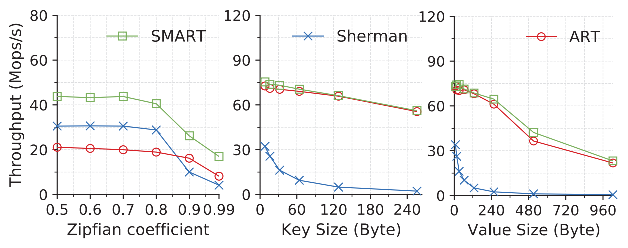
As the design of RDWC is transparent to the lower-level index structures, it will lead to the same amount of performance improvements on Sherman, *i.e.*, 1.3× and 1.1× under write-intensive and read-only workloads (Figure 15). Therefore, after applying RDWC to Sherman, SMART can still achieve 4.7× (= 6.1/1.3) higher throughput under write-intensive workloads and 2.5× (= 2.8/1.1) higher throughput under read-only workloads.

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 17, 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.

5.4 Sensitivity

In this section, we investigate how the workload skewness, key size, and value size affect the performance of SMART. We use 16 CNs with 16 clients each and integer keys for the sensitivity evaluation.

Skew test. Figure 18a shows the performances of different tree indexes on a generated Zipfian workload [35] (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



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

Figure 18: The sensitivity analysis.

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 performance 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.

Impact of key/value size. Figures 18b and 18c 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 1024 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 11d.

6 Related Work

Disaggregated Memory. The DM architecture is widely discussed in the literature [3, 9, 16, 19, 20, 27, 47], 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 [46] 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 [52] 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 [48] designs a fully memory-disaggregated key-value store that brings disaggregation to metadata management. ROLEX [34] proposes a scalable RDMA-oriented learned key-value store that dissociates the model retraining from data modification operations. RACE [60] is an extendible RDMA-based hashing index with lock-free remote concurrency control and efficient remote resizing. Sherman [53] is a B+ tree index on DM with RDMA-friendly software techniques to boost index write performance. SMART focuses on building a fast, scalable radix tree index on DM with small read and write amplifications.

RDMA-based Tree Indexes. Attracted by the high performance of RDMA, there are increasing studies focusing on RDMA-based tree indexes [1, 41, 45, 53, 59]. Many studies conduct operations via remote procedure calls (RPCs), which is unsuitable for DM due to weak memory-side computation power. FG [59], designed as a B-link tree, is the first index that completely leverages *one-side verbs* for write operations and thus supports DM. Sherman [53] is the state-of-the-art B+ tree index with several RDMA-friendly software techniques. However, constrained by the structure of the B+ tree, it suffers from low peak throughput and early latency deterioration due to read and write amplifications. 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 [1, 7, 14, 25, 36, 44, 49]. 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 paper 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.

Acknowledgments

We sincerely thank our shepherd Steven Swanson and the anonymous reviewers for their constructive comments and suggestions. This work is supported by the National Natural Science Foundation of China (Project No. 61971145), the Natural Science Foundation of Shanghai (Project No. 22ZR1407900), and Huawei Cloud. Yangfan Zhou (zyf@fudan.edu.cn) and Pengfei Zuo (pfzuo.cs@gmail.com) are the corresponding authors.

References

[1] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*, pages 120–126. ACM, 2019.

[2] Marcos Kawazoe Aguilera, Wojciech M. Golab, and Mehul A. Shah. A practical scalable distributed B-tree. *Proc. VLDB Endow.*, 1(1):598–609, 2008.

[3] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. In *12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2020, July 13-14, 2020*. USENIX Association, 2020.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 53–64. ACM, 2012.

[5] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. Caching with delayed hits. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 495–513. ACM, 2020.

[6] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. HOT: A height optimized trie index for main-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 521–534. ACM, 2018.

[7] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. PRISM: Rethinking the RDMA interface for distributed systems. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 228–242. ACM, 2021.

[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. PolarDB Serverless: A cloud native database for disaggregated data centers. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2477–2489. ACM, 2021.

[9] Amanda Carbonari and Ivan Beschastnikh. Tolerating faults in disaggregated datacenters. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017*, pages 164–170. ACM, 2017.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[11] Intel Corporation. Write combining memory implementation guidelines. https://download.intel.com/design/PentiumII/applnots/24442201.pdf.

[12] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 401–414. USENIX Association, 2014.

[13] 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. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 1–14. USENIX Association, 2019.

[14] 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. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 51–66. USENIX Association, 2018.

[15] Fonxat. 300 million email database. https://archive.org/details/300MillionEmailDatabase, 2018.

[16] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 249–264. USENIX Association, 2016.

[17] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 202–215. ACM, 2016.

[18] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software co-designed disaggregated memory system. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 417–433. ACM, 2022.

[19] Sangjin Han, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker. Network support for resource disaggregation in next-generation datacenters. In *Twelfth ACM Workshop on Hot Topics in Networks, HotNets-XII, College Park, MD, USA, November 21-22, 2013*, pages 10:1–10:7. ACM, 2013.

[20] Eric Hooper. Intel rack scale design: Just what is it? https://www.datacenterdynamics.com/en/opinions/intel-rack-scale-design-just-what-is-it, 2018.

[21] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. iDistance: An adaptive B$^+$-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[22] Minwen Ji, Alistair C. Veitch, and John Wilkes. Seneca: Remote mirroring done write. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*, pages 253–268. USENIX, 2003.

[23] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 437–450. USENIX Association, 2016.

[24] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 1–16. USENIX Association, 2019.

[25] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostic, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 756–771. ACM, 2021.

[26] Wook-Hee Kim, Madhava Krishnan Ramanathan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. PACTree: A high performance persistent range index using PAC guidelines. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 424–439. ACM, 2021.

[27] HP Labs. The machine: A new kind of computer. https://www.hpl.hp.com/research/systems-research/themachine, 2014.

[28] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 2–13. ACM, 2009.

[29] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, pages 257–270. USENIX Association, 2017.

[30] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent DRAM indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 462–477. ACM, 2019.

[31] Seung-Seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. MIND: In-network memory management for disaggregated data centers. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 488–504. ACM, 2021.

[32] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 38–49. IEEE Computer Society, 2013.

[33] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 3:1–3:8. ACM, 2016.

[34] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A scalable RDMA-oriented learned key-value store for disaggregated memory systems. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, pages 99–114. USENIX Association, 2023.

[35] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 429–444. USENIX Association, 2014.

[36] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto SmartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, pages 318–333. ACM, 2019.

[37] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. ROART: Range-query optimized persistent ART. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 1–16. USENIX Association, 2021.

[38] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 183–196. ACM, 2012.

[39] Ajit Mathew and Changwoo Min. HydraList: A scalable in-memory index using asynchronous updates and partial replication. *Proc. VLDB Endow.*, 13(9):1332–1345, 2020.

[40] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 103–114. USENIX Association, 2013.

[41] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. Balancing CPU and network in the cell distributed B-tree store. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 451–464. USENIX Association, 2016.

[42] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: Practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 16:1–16:12. ACM, 2018.

[43] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 315–332. USENIX Association, 2020.

[44] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-accelerated distributed transactions. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 740–755. ACM, 2021.

[45] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 433–448. ACM, 2019.

[46] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 69–87. USENIX Association, 2018.

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

[48] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. FUSEE: A fully memory-disaggregated key-value store. In *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, pages 81–98. USENIX Association, 2023.

[49] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. StRoM: Smart remote memory. In *EuroSys '20: Fifteenth EuroSys Conference 2020,*

Heraklion, Greece, April 27-30, 2020, pages 29:1–29:16. ACM, 2020.

[50] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD lifetimes with disk-based write caches. In *8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 23-26, 2010*, pages 101–114. USENIX, 2010.

[51] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. RFP: When RPC is faster than server-bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 1–15. ACM, 2017.

[52] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 33–48. USENIX Association, 2020.

[53] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed B+tree index on disaggregated memory. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 1033–1048. ACM, 2022.

[54] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu. Concordia: Distributed shared memory with in-network cache coherence. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 277–292. USENIX Association, 2021.

[55] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a Bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 473–488. ACM, 2018.

[56] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered key-value store using remote learned cache. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 117–135. USENIX Association, 2020.

[57] Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. Efficient B-tree based indexing for cloud data processing. *Proc. VLDB Endow.*, 3(1):1207–1218, 2010.

[58] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 191–208. USENIX Association, 2020.

[59] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Designing distributed tree-based index structures for fast RDMA-capable networks. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 741–758. ACM, 2019.

[60] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. One-sided RDMA-conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 15–29. USENIX Association, 2021.

A Artifact Appendix

Abstract

The artifact provides the source code of SMART and automated scripts to reproduce all the experiment results in the paper. 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.

Scope

Superiority of ART on DM. By reproducing the experiments of Figure 3, 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*.

Challenges of ART on DM. By reproducing the experiments of Figure 4, the artifact can validate that ART suffers from significant challenges on DM under *hybrid read-write workloads*.

Efficacy and Efficiency of SMART. By reproducing the experiments of Figure 11-18, 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.

Contents

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.

Automated scripts. The artifact also contains automated scripts to reproduce all the experiment results in the paper, *i.e.*, Figure 3-4, 11-18. Each figure has a Python script to automatically reproduce and visualize the experimental results.

Hosting

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

Requirements

The artifact is developed and tested using the r650 machines on CloudLab. 16 r650 machines are needed to reproduce all the results. Each r650 machine has two 36-core Intel Xeon CPUs, 256GB of DRAM, and one 100Gbps Mellanox ConnectX-6 IB RNIC. Each RNIC is connected to a 100Gbps Ethernet switch.

Tutorial

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.

Listing 1: Commands to set up the environment.

```
# Get the source codes
git clone 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
```

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 all the workloads to reproduce the results.

Listing 2: Commands to generate all workloads.

```
# 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 workloads
sh generate_full_workloads.sh
```

Results Reproduced. The artifact provides a single batch script to reproduce all the experiments. 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.

Listing 3: Commands to start all experiments.

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