Outback: Fast and Communication-efficient Index for Key-Value Store on Disaggregated Memory

Yi Liu
University of California Santa Cruz
yliu634@ucsc.edu

Minghao Xie
University of California Santa Cruz
mhxie@ucsc.edu

Shouqian Shi
University of California Santa Cruz
sshi27@ucsc.edu

Yuanchao Xu
University of California Santa Cruz
yxu314@ucsc.edu

Heiner Litz
University of California Santa Cruz
hlitz@ucsc.edu

Chen Qian
University of California Santa Cruz
qian@ucsc.edu

ABSTRACT

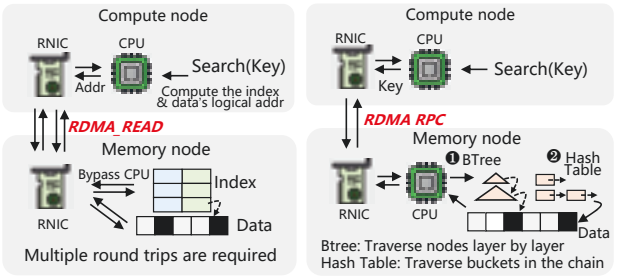
Disaggregated memory systems achieve resource utilization efficiency and system scalability by distributing computation and memory resources into distinct pools of nodes. RDMA is an attractive solution to support high-throughput communication between different disaggregated resource pools. However, existing RDMA solutions face a dilemma: one-sided RDMA completely bypasses computation at memory nodes, but its communication takes multiple round trips; two-sided RDMA achieves one-round-trip communication but requires non-trivial computation for index lookups at memory nodes, which violates the principle of disaggregated memory. This work presents Outback, a novel indexing solution for key-value stores with a one-round-trip RDMA-based network that does not incur computation-heavy tasks at memory nodes. Outback is the first to utilize dynamic minimal perfect hashing and separates its index into two components: one memory-efficient and compute-heavy component at compute nodes and the other memory-heavy and compute-efficient component at memory nodes. We implement a prototype of Outback and evaluate its performance in a public cloud. The experimental results show that Outback achieves higher throughput than both the state-of-the-art one-sided RDMA and two-sided RDMA-based in-memory KVS by 1.06-5.03×, due to the unique strength of applying a separated perfect hashing index.

PVLDB Reference Format:
Yi Liu, Minghao Xie, Shouqian Shi, Yuanchao Xu, Heiner Litz, and Chen Qian. Outback: Fast and Communication-efficient Index for Key-Value Store on Disaggregated Memory. PVLDB, 18(2): 335-348, 2024.
doi:10.14778/3705829.3705849

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yliu634/outback.

1 INTRODUCTION

Disaggregated memory systems [33, 39, 42, 43, 47, 49, 53, 61] represent a transformative departure from traditional computing architectures, distributing memory storage and computational resources into distinct pools of nodes – compute pools include nodes that carry rich CPU resources, and memory pools include nodes that carry rich DRAM and storage resources. This framework is prevalent in contemporary data centers and cloud infrastructures [60, 62], providing benefits such as enhanced resource utilization efficiency and flexibility to scale the system out by deploying more hardware. Disaggregated memory systems can harness Remote Direct Memory Access (RDMA)-capable networks [14, 26, 40, 43, 46], featuring substantial throughput capacities (ranging from 40 to 400 Gbps) and small latency within the microsecond range. Memory-intensive applications, such as transaction systems [11, 52, 57, 58] and key-value stores (KVSs) [14, 20, 28, 66], store the data and index data structures at the memory nodes and perform computation tasks at the compute nodes.



(a) An example of one-sided RDMA. (b) An example of two-sided RDMA.

Figure 1: Examples of two types of RDMA systems.

Existing RDMA networks for disaggregated memory can be categorized into two types. 1) One-sided RDMA [28, 34, 37, 48, 66] as shown in Fig. 1(a). This type of network completely separates computation and memory access tasks. Each data request requires multiple round trips of communication between the compute node and the memory node. At least two round trips are necessary: one to access the index and the other to access the stored data. Note that many indices require multiple layers of accesses [37, 48], hence they need much more than two round trips [36, 37]. 2) Two-sided RDMA or RDMA RPC [21, 22], as depicted in Fig. 1(b), involves computation tasks on both compute and memory nodes, requiring only a single round-trip communication for each request. However, two-sided RDMA cannot bypass the CPU on the memory node, necessitating the CPU on the memory node to execute the computation of the index structure, such as hash computations and key comparisons. Since the CPU resource on a memory node is very limited in disaggregated systems, this design may lead to CPU

Proceedings of the VLDB Endowment, Vol. 18, No. 2 ISSN 2150-8097.
doi:10.14778/3705829.3705849

bottlenecks and potentially higher latency compared to one-sided RDMA [17, 48].

A natural question arises: "Can we design a one-round-trip RDMA-based network that does not incur computation-heavy tasks on memory nodes?" Achieving this goal is extremely challenging because putting the index on memory nodes leads to CPU bottlenecks while putting the index on compute nodes causes memory bottlenecks and consistency issues.

This paper presents the first solution to this research problem. Our key innovation is to design and implement an RDMA RPC-based system, called Outback, which decouples its index into two components. The first component is memory-efficient and includes most computation operations of the index, which is placed onto the compute nodes. The second component contributes to the most memory cost of the index, but its computation is trivial, and it is on the memory nodes. Such a design principle of decoupling the index is ideal for disaggregated memory systems: all computation tasks for `Get` requests and the majority of computation for data `Insert` requests are offloaded on compute nodes, while memory nodes focus on providing service for memory read and write. Hence, this approach is particularly effective for real-world workloads dominated by `Get` requests. It is also well-suited for emerging disaggregated memory systems equipped with SmartNICs with limited computation resources [1, 6, 54].

Similar to prior one-round-trip RDMA networks [21, 22], Outback also relies on two-sided RDMA. We implement Outback as a distributed KVS application. The index design of Outback is motivated by a recent advance of dynamic minimal perfect hashing (DMPH), called Ludo hashing [44]. The original design of Ludo hashing did not decouple the index into computation-heavy and memory-heavy components, but its perfect hashing property offers the opportunity for a novel decoupling approach that allows data `Get` requests in one round trip with trivial computation on memory nodes. For data `Insert` requests, we design additional operations to update the index on both the compute and memory nodes to ensure data consistency.

Overall, this paper makes the following contributions:

- We present a novel solution that provides one-round-trip RDMA with RPC that incurs minimal computation tasks on memory nodes. The design principle of decoupling the index works effectively for emerging disaggregated memory systems.
- We design the Outback system as a distributed KVS. We design a decoupled index based on a recent data structure of DMPH. We also designed the algorithms and protocols for supporting data operations and system updates.
- We implement a prototype of Outback and evaluate the performance on YCSB workloads [12] and four real-world datasets from SOSD [35]. The experimental results show that Outback achieves higher throughput than both the state-of-the-art one-sided RDMA and two-sided RDMA-based in-memory KVS by 1.06-5.03×.

2 BACKGROUND

2.1 Disaggregated Memory with RDMA

Disaggregated memory systems with RDMA can be categorized into two types: one-sided RDMA systems [28, 34, 37, 48, 66], and two-sided RDMA (RDMA-RPC) systems [21, 22]. An example of one-sided RDMA systems [28, 34, 37, 48, 66] is illustrated in Fig. 1(a). These systems support applications such as KVS and transaction systems with various index data structures, including B/B+ trees, hash tables, radix trees, and learned indexes. However, it is widely recognized that multiple round-trip communications are needed for each `Get` request: at least one for querying the index and one for reading data. The high communication cost results in both long latency and network congestion.

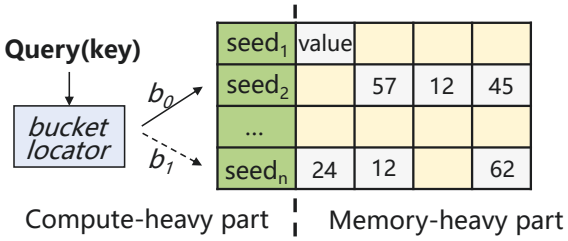


Figure 2: Ludo hashing.

Two-sided RDMA-based systems [21, 22] have been investigated to dispatch compute nodes' requests to the memory node via RPC over the RDMA network with only one round trip. As depicted in Fig. 1(b), a data index, such as a B-Tree or hash table, is maintained at the memory node. When a data query occurs, in addition to polling the RNIC and posting messages, the CPU of the memory node is responsible for traversing the index. The memory node has to perform computational tasks, including hash computation, fingerprint checking, and key comparisons. This process introduces additional computational overhead and memory accesses. Existing solutions [11, 29, 66] that store keys' fingerprints in their hash tables to save memory usage also introduce extra computation. For example, if the memory node employs the state-of-the-art (2,4)-Cuckoo hash table [38], each `Get` request requires one fingerprint computation and, at most eight rounds of fingerprint checking.

2.2 Dynamic minimal perfect hashing

In this subsection, we first introduce the background of DMPH and then present an existing MPH implementation, Ludo hashing [44].

Perfect hashing [16] represents a family of schemes that designs and manipulates hash algorithms to distribute keys to different buckets in a hash table without collisions. Since it is impractical to find a single hash function that generates no collisions for a large set of keys, a common approach is to use two levels of mapping. The first level maps keys to a number of groups, each of which contains several keys. The second level addresses key collisions inside each group. Minimal perfect hashing maps n keys to exactly n buckets, but it is inflexible for key insertions and only applicable to a static set. To allow key dynamics, dynamic minimal perfect hashing (DMPH) may use $(1+\epsilon)n$ positions for n keys [44, 64]. One primary advantage of perfect hashing is that it does not need to store the keys in the hash table. Since perfect hashing eliminates collisions, a key query does not need to compare keys to address collisions. Avoiding storing keys can significantly reduce memory costs, because as a secondary index, the size of keys (usually hundreds of bits) is much longer than the queried value in a hash table (usually a storage address in tens of bits).

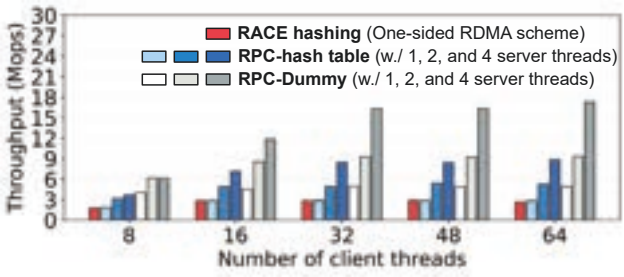
One of the most recent solutions of DMPH is called Ludo hashing [44]. As shown in Fig. 2, Ludo hashing [44] first uses a data structure called Othello [56], a dynamic implementation of Bloomier filters [8] with two arrays, as the *bucket locator* to distribute keys into different buckets, each of which includes exactly 4 slots. Then, in each bucket B_i, Ludo hashing uses brute force to find a hash seed s_i such that the hash function with s_i can map the 4 keys in the bucket to 4 different slots without collision. Hence, there is no need to store keys in the table for collision resolution. The space cost of Ludo is $3.76 + 1.05l$ bits per key, where l is the length of the record value, which is claimed to be the smallest memory cost in the literature [44]. The bucket locator leverages Othello arrays [56], which costs 2.33 bits per key. Each bucket contains a 5-bit long seed shared by four keys in Ludo, i.e., 1.25/0.95 bits per key when we set the load factor as 95%. Also, the majority of memory cost is for storing the values in the buckets, costing $1.05l$ bits per key. We observed that the computation for looking up the slot only needs the bucket locator and the seeds, which are memory efficient. On the other side, the hash table buckets/slots part storing all data values contributed to most memory of this index, but it requires little computation.

3 MEASUREMENT AND MOTIVATION

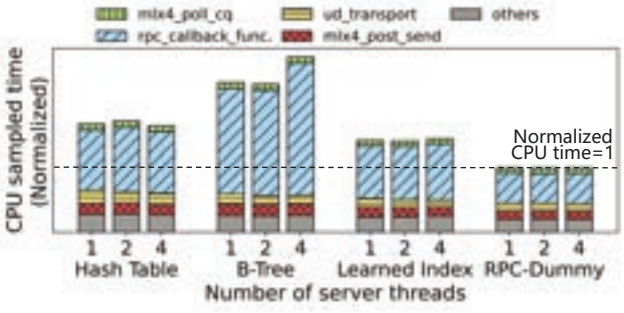
We wonder if, we remove the computation cost at the memory node, will RDMA-RPC demonstrate much higher throughput than the state-of-the-art one-sided RDMA? **If the answer is "Yes", then there is a great opportunity to design a high-throughput RDMA-based KVS by reducing the computation cost at the memory node.**

Toward this objective, we conduct experiments to analyze the throughput performance of both one-sided RDMA and RDMA-RPC systems with 9 r320 servers in CloudLab [15], each is configured with a Mellanox CX3 adapter (50Gbits). We compare the performance of the following systems with `Get`-only workload. (1) RACE hashing [66], a state-of-the-art one-sided RDMA-based scheme. Its hashing index is crafted for disaggregated memory, facilitating data retrieval within two round trips. (2) RPC-hash table, a two-sided RDMA method whose compute nodes and memory nodes communicate in RDMA unreliable datagram (UD) mode. Each memory node maintains a chained hash table in its local memory to handle remote data requests. (3) RPC-Dummy. A hypothetical RDMA-RPC method that incurs minimal computation cost at each memory node. RPC-Dummy only implements one memory access and then returns any data in the accessed memory at the memory node, with no extra computation tasks. RPC-Dummy's throughput can be considered the upper bound among all possible RDMA-RPC systems. We use this method to explore the performance potential of our design objectives. We vary the number of memory node threads as 1, 2, and 4 in RPC-based approaches, and each memory node thread maintains one Queue Pair (QP) and runs in a distinct CPU core.

The results are shown in Fig. 3(a). For one memory node thread (one core), RPC-hash table achieves a throughput similar to that of RACE hashing. For RACE hashing, multiple reasons limit its throughput, including the two round trips to complete one data `Get` operation and multiple RC connections of the compute node threads that incur resource contention in the RNIC cache [10]. RPC-hash table requires only one round trip, but the complexity of querying



(a) Throughput of different systems with limited number of memory node threads.



(b) The CPU time breakdown on a memory node with one thread.

Figure 3: Observations from the microbenchmarks.

the index on the memory node introduces extra latency and limits its throughput. The throughput of RPC-hash table increases correspondingly when we increase the number of threads to 2 and 4. In contrast, RACE hashing maintains a static performance. RPC-Dummy can outperform RPC-hash table by around 2× under the cases of both single and multiple memory node threads. Hence an RDMA-RPC network that introduces little computation overhead to the memory node can achieve higher throughput than both existing one-sided RDMA and RDMA-RPC solutions. The results suggest that RPC-based KVS has a potential for throughput improvement by reducing computation tasks at memory nodes, which motivates the design of this project.

CPU utilization breakdown for RPC-based approaches. We run RDMA-RPC with different indices: hash table, Btree, and learned index, at the memory node. The CPU time consumed by these four RPC-based KVS systems while handling an equal number of data `Get` requests is normalized and presented in Fig. 3(b), with the number of compute node threads fixed at 64. RPC-Dummy takes the least time. Other approaches consume more time in different amounts. For RPC-Btree, in addition to the communication overheads for polling `mlx4_poll_qp` (4.03%), posting messages `mlx4_post_send` (7.52%) and UD transport (6.85%) from connection management, the most CPU-consuming event is the RPC callback function (70.59%), which executes local index lookup and data access. In all four schemes, the RPC callback function consumes the most CPU time, and the variations in CPU consumption among them are mainly attributed to differences in the RPC callback function. RPC-Btree consumes the most CPU time for RPC callback, followed by RPC-hash table. RPC-Dummy spends the least CPU time on the RPC callback function (46.11%) and serves the most data requests because there is no computation burden for the memory node in RPC-Dummy. In disaggregated systems, tasks such as computing hash functions on a hash table, traversing tree nodes in a

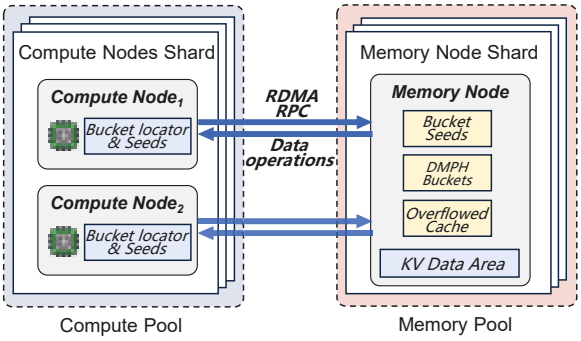


Figure 4: Outback overview

B-Tree, and executing learned models on a learned index are not ideally suited for memory nodes. **The throughput of RDMA-RPC methods is mainly limited by CPU usage during the RPC callback function for index lookups and data reads. High CPU consumption from complex index computations on memory nodes reduces throughput, particularly when CPU resources are constrained, indicating that optimizing these computations can enhance performance.**

4 DESIGN OF OUTBACK

4.1 Overview

Based on the motivation presented in the previous section, we design and implement an RDMA-RPC network that aims to minimize computation tasks on memory nodes, consequently enhancing the system throughput. This section presents the design of Outback, a scalable RDMA RPC-based disaggregated KVS that tackles the performance limitations of existing RDMA RPC and one-sided RDMA-based schemes. To accomplish this design objective, we decouple the index of Outback into two components: 1) a computation-heavy component running on compute nodes, and 2) a memory-heavy component running on memory nodes. In particular, DMPH provides an opportunity for this decoupling. By carefully examining the DMPH's read and insertion operations, we observe that the final step consistently is directly retrieving the value from a specific memory location, while all the previous steps are employed to determine that location. Contrary to DMPH, other hash tables necessitate retrieving the key from the hashed location by key probing and comparison, and only when the key matches the search key, the value can be returned. The distinctive process of DMPH motivates us to store all values in the memory-heavy components because they can be read without extra computation. And the steps to determine the location of the value can be placed in the compute-heavy component running on the compute nodes.

Outback requires only a single round trip for data requests while supporting a large number of concurrent compute nodes's requests. In contrast to other RDMA RPC-based approaches [20, 22], Outback substantially reduces CPU resources required on the memory node. In the following, we elaborate on the components maintained in the compute pool and memory pool of Outback.

Fig. 4 depicts the overall structure of Outback, which leverages a shared-nothing architecture [47] for separating data into different shards with consistent hashing [23]. The compute pool comprises multiple compute shards, each accommodating several compute

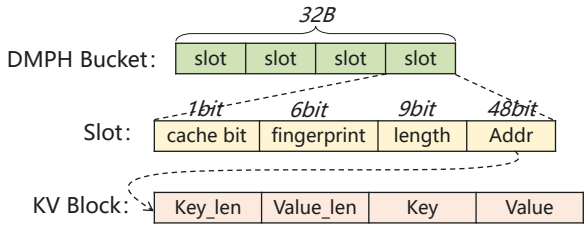
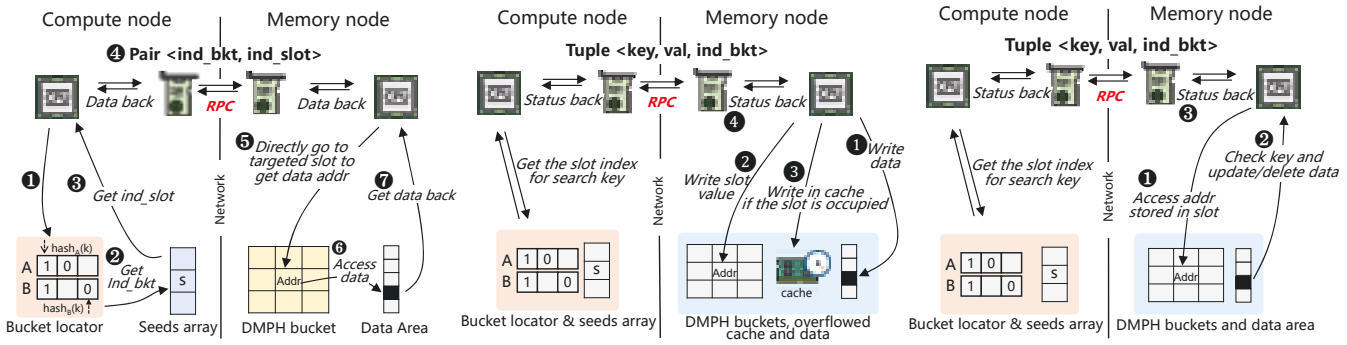


Figure 5: The data layout in a DMPH bucket.

nodes. Note that the configuration for the number of shards and the number of compute nodes depends on the memory budget in compute nodes and the whole size of the datasets. For each shard, an index is built based on the keys of the shard, and the returned values of the index represent the memory locations that store the corresponding data associated with the keys. The index is decoupled into the compute-heavy and memory-heavy components. Each compute node is allocated a memory budget for caching the compute-heavy component, including the bucket locator and the seeds. The default setting is there are 64 million keys in a shard, and the memory overhead on each compute node is less than 50MB (§5.8). This is considered a small overhead because recent one-sided RDMA solutions cost over 300 MB on each compute node for index caching and other purposes [28, 50]. All compute nodes in the same shard will connect to the memory node with RDMA RPC for data operations and one-sided RDMA for new bucket locator fetching after index resizing – the details will be explained in §4.4. Each shard consists of one memory node, which contains the most updated bucket seeds, overflowed cache, DMPH buckets, and KV data in the shard. The DMPH buckets store the data addresses in the KV data memory space of the keys in the shard. The latest bucket seeds are maintained to ensure the consistency of data insertion. Additionally, the overflowed cache for KV pairs is used to temporarily hold the pair of the new key and the address, which cannot be inserted into DMPH buckets without the need for hash table resizing. We leverage a hash table to work as the overflowed cache in Outback. The KV data in each shard is replicated to two other shards, serving as replicas with checkpoints. These two replica shards can be chosen as the two successive shards in the consistent hashing ring. Each key's primary replica shard is referred to as the *primary shard* of the key. Each shard is identified by a `uuid`. We assume there is a *service layer* in front of the compute nodes responsible for only forwarding data requests to one of the compute nodes in the primary shard based on the key's hash value in the consistent hashing ring. After the memory node in the primary shard completes a data update operation, it forwards the update to its replica shards. To ensure load balance among compute nodes within a shard, the service layer maintains a counter for each shard and distributes requests to the compute nodes in a round-robin fashion.

4.2 Decoupled DMPH index

In this section, we explain the detailed data structure and its components maintained in the compute node and the memory node. We reuse the design of Ludo hashing as introduced in §2. There are two candidate buckets for each key, and the bucket locator runs a data structure called Othello [56] to determine which bucket the value of the search key is stored in. Each Ludo bucket contains one



(a) `Get` operation. (b) `Insert` operation. (c) `Update` and `Delete` operation.

Figure 6: Data operation protocols in Outback.

seed and four slots. By computing a hash value with the search key and the seed, the key is mapped to an exact slot of the bucket without colliding with other keys within the same bucket. The value stored in the slot represents the key's data address and is utilized to retrieve the corresponding data.

We decouple the entire data structure of Ludo hashing into two components. The compute-heavy component running on each compute node stores both the bucket locator and the seeds for all DMPH buckets. This component completes all computations related to finding the location that stores the value of the search key and costs only $3.76n$ bits – $2.33n$ bits for the bucket locator and $1.43n$ bits for the seeds, where n refers to the number of KV pairs in a shard. Within the memory node, the memory-heavy component consists of all DMPH buckets that store the data addresses for all keys in the shard. Assuming the load factor of the DMPH table is set to ϵ with a default value of 0.95, the number of DMPH buckets will be $n/(4 \cdot \epsilon)$ as each bucket accommodates four slots. The detailed layout for each DMPH bucket is illustrated in Fig. 5, and each bucket is 32-Byte long with four packed slots. There are four fields in each slot: cache bit (1 bit), fingerprint (6 bits), length (9 bits), and data address (48 bits). The cache bit serves as an indicator to identify whether another key(s) share the same slot, with its index stored in the overflowed cache. Meanwhile, the 6-bit fingerprint is only utilized during the index update process to verify if the KV data referenced by the address in this slot corresponds to the search key or not. This fingerprint check is exclusively applied during data write requests, and any false positives do not impact the final result. This is because a comprehensive recheck of the full key occurs after accessing the actual KV data block on the compute node side. Note that read requests do not need to check the fingerprint. The address signifies the starting offsets of the KV block, while the length indicates the byte length of the entire KV block in the underlying KV data area. In the underlying data area, the KV block is compactly stored with four fields. The initial two numbers, each occupying 8 bytes, denote the length of the key and the subsequent value field.

The overflowed cache accommodates the key-address pair that cannot be inserted into the mapped DMPH bucket without modifying the bucket locator or resizing the entire hash table.

For an estimation, if $\epsilon = 0.95$, the component at the compute node contributes to only 5.5% of the total memory size of the index while the component at the memory node accounts for the larger portion of 94.5%.

4.3 Outback operations and protocols

This subsection presents the data operations and the corresponding protocol of Outback, including the data `Get`, `Insert`, `Update`, and `Delete` operations, as shown in Fig. 6.

4.3.1 Data Get operation. As shown in Fig. 6(a), the compute node maintains the bucket locator (two Othello arrays A and B) and the seed array s. Meanwhile, the memory node maintains the DMPH buckets that store KV addresses and the KV data in a disjoint memory area. When there is a data `Get` request for key k, the compute node will ❶ compute the bucket index from the bucket locator by looking up two bits on the two arrays, respectively. Assuming the bucket index that stores the queried key is *ind_bucket*, the compute node will then proceed to ❷ compute the slot number within the bucket with the hash function and the seed $s[ind\_bucket]$. At this point, the compute node ❸ gets both the bucket index and slot index in the MPH buckets, and it ❹ posts them to memory nodes with RDMA_SEND in the opaque fields. After the memory node gets the message and parses the index numbers of the bucket and slot, *ind_bucket* and *ind_slot*, it will ❺ go directly to the MPH buckets to access the exact slot without any extra computation. Then, the memory node ❻ gets the data offset in the underlying KV data area from the last 48-bit field of the slot. At last, ❼ the KV data will be read back and returned to the initiating compute node for full key check. For example, when a compute node requests data for key 5, it computes the bucket index 10 and slot index 0 based on the bucket locator and the locally stored seeds. Then, the pair of indices (10,0) is sent to the memory node. The data index stored in the indicated slot of the memory node is read, and the corresponding data block is returned. Lastly, the compute node checks the cache bit and a full key to see if the MakeupGet is needed.

There could be some KV pairs that are temporarily inserted into the overflowed cache during the updates and reconstruction of the index. In this circumstance, the compute node is tasked with checking the cache bit, ensuring that the returned full key aligns with the queried one. If the key does not match the requested one, and the cache bit in the slot is set to 1, the compute node will initiate another `Get` makeup request with the *ind_slot* specified as -1, signaling the memory node that the returned key does not match the requested key. While it is possible to offload the full key

comparison task to the memory node, saving one round trip, this approach introduces computation overheads on the limited remote core resources. To make the common case easy, we opt to assign the full key check task to compute nodes.

Makeup `Get` request. When the KV data returned to the compute node does not match the requested key, there are two reasons: (1) The requested key is kept in the overflowed cache. The KV pair is inserted after the DPMH table is constructed, and the hashed slot is occupied by another key. (2) The requested key is in another slot of the hashed bucket. This case results from changing the order of keys based on the new seed within the bucket when the inserted key can fit into the current DMPH table (detailed in Section 4.3.2). Due to the above two situations, the compute node will send the makeup `Get` request with the *ind_slot* as -1 to the memory node. The memory node will search the overflowed cache first; if there is a cached item matching the full key of the requested key, it will read the data and return it to the compute node. If not, it will read out all the KV blocks referred by the hashed bucket (at most four) and compare the keys until it finds the requested key. Additionally, the new seed will be returned back to the compute node if the key is found in another slot, and the compute node will update the copied seeds array for this bucket locally.

4.3.2 Data Insert operation. The main idea of implementing the data `Insert` operation of Outback is to determine if we can insert the key into the index without significant changes to the current bucket locator. If an `Insert` operation only requires changing the value in one DMPH bucket, Outback can make this change directly. However, if a `Insert` operation will cause the index to resize, which usually happens after a number of insertions, Outback needs to ensure the correctness of the `Insert` operation and following lookups during index resizing. As shown in Fig. 6(b), like `Get` operation, the compute node will get *ind_bucket* and *ind_slot* from the bucket locator and the seeds through multiple hashing computations. Different from `Get`, the RPC message posted to QP should include the full key. Thus, the memory node can parse the *ind_bucket*, *ind_slot*, and the key from the message and execute the following steps. ❶ the memory node will write the data into the underlying data area, then it can get the data length and the address (offset in the data area) for indexing. After the memory node composes the value from the corresponding slot with the cache bit (set to zero by default), fingerprint, length as well as address, it ❷ will try to insert it in the DMPH table.

We discuss the rest of `Insert` in three cases:

• **`Insert` without bucket locator and seed change.** The memory node checks the slot indicated by *ind_bucket* and *ind_slot*. If the length field is empty (length is 0), signifying there is no key associated with this slot, the memory node inserts the composed slot value (Fig. 5) into this location and returns `SUCCESS` to the compute node. Conversely, if the length is non-zero, indicating that an existing key is using this slot, the memory node proceeds to check the fingerprint and compares the full key to determine if the original key in this slot matches the inserted key. If they match, the insertion is resolved and treated as an `Update` operation. The fingerprint can prevent the memory node from reading the full key in the KV data area if they are not the same.

• **`Insert` with seed changes but the bucket locator remains the same.** If the key associated with the targeted slot does not match the newly inserted one, an examination is made to determine if there is another available slot within this bucket. Assuming there are only three keys in this bucket, and the slot indicated by *ind_slot* is already occupied by a different key, the memory node endeavors to find a new seed that accommodates all four keys in the bucket without causing collisions, thereby preserving perfect hashing policy in this bucket. The other three keys are read from the underlying KV data area, and the memory node employs a brute-force approach to identify a new seed for perfect hashing within this bucket. Importantly, the bucket locator does not need to change because all four keys remain in the same bucket. Subsequently, the updated seed for this bucket is returned to the compute node, which then propagates this modification to other compute nodes in the same shard.

• **Insert data to overflowed cache.** When all four slots within the bucket are occupied, and the memory node is unable to find an empty slot for the inserted key, the pair of the key and the KV block address will be ❸ placed in the overflowed cache. Also, the cache bit in the conflicted DMPH slot will be set to 1 to indicate at least one key in the overflowed cache sharing the same hash slot. Instead, when the number of KV pairs in the overflowed cache reaches a predefined threshold, the memory node initiates the index resizing process to accommodate more KV pairs in a new DMPH table.

The data insertion process on each memory node works as follows. At first, the memory node will lock the data operations on the targeted bucket to prevent the potential data operations on this bucket. The inserted key might have been stored in the DMPH table before. Thus, the memory node will check if the insert request can be resolved to a data update operation by comparing the fingerprint and the underlying full key. Then, the memory node first writes the KV block to the underlying data area and processes the data insert request based on the stored bucket keys into the mentioned three cases. Finally, the memory node unlocks the bucket after it finishes the data insert operation. Note that the data insert request tuple sent by the compute node consists of the KV pair and *ind_bucket*, not including *ind_slot*. The reason is that the memory node keeps the most update seeds array in the shard and can use the seeds to do the hash computation as the slot locator. Also, the bucket locator is not maintained in the memory node, and the data insert operation will not modify it after the DMPH table is constructed every time. This choice is made because modifying the bucket locator requires changing seeds for keys in at least two buckets, leading to more computational overhead.

4.3.3 Data Update and Delete operations. For data update and deletion, the compute node also acquires the *ind_bucket* and *ind_slot* from the bucket locator and the seeds array. Like the `Insert` operation, the compute node transmits the full key to the memory node. As illustrated in Fig. 6(c), the memory node directly accesses the address of the KV data from the DMPH bucket and verifies whether the requested key matches the underlying data. Once the memory node confirms the key, for `Delete`, it marks the length of the slot value as zero and returns the corresponding status. In the case of `Update`, it writes the new data to the underlying data area. If the cache bit is set to 1 and the keys differ, the memory node will go to the overflowed cache to get the data address.

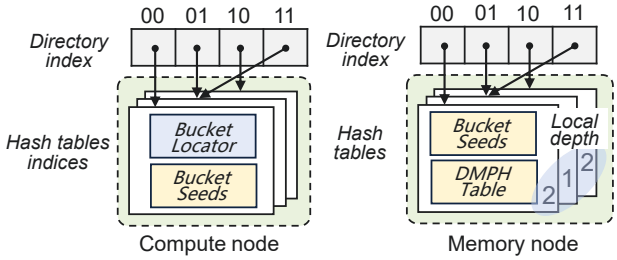


Figure 7: Extendible hashing in Outback.

4.3.4 Concurrency control. Each bucket in the DMPH table within the memory node has a mutex lock. Prior to executing any `Insert`, `Update`, or `Delete` operation, the relevant bucket is locked, blocking any access to its indices. Subsequently, the operation is executed and the lock is released. During the lock period, all other operations targeting this bucket are buffered and only processed once the lock is released.

4.4 Index resizing

When the number of KV pairs in the overflowed cache surpasses a predefined threshold, index resizing and reconstruction become necessary to accommodate the KV pairs into a new hash table. This resizing process introduces two challenges: (1) managing data operation requests during resizing and (2) efficiently coordinating the compute node and memory node to transfer the bucket locator and seeds.

To support data requests on runtime while index resizing, we apply extendible hashing [32, 66] to allocate a new DMPH table to accommodate more keys' indices, and a *directory index* is used to identify the multiple DMPH tables, which is an additional hash layer as shown in Fig. 7. This approach reduces the number of keys that need to be moved during index resizing and shortens the resizing duration. Compute nodes maintain the bucket locator and seeds array for each single hash table, while memory nodes store the most update seeds array and DMPH tables, as well as local depth array [32, 66].

In each shard, we have two size thresholds for overflowed cache; One is for slowing down insertions, s_{slow}. The memory node reaching this threshold will enter the index resizing process. The other threshold is the size when the memory node stops any following insertions s_{stop} even if the index resizing is not finished and $s_{stop} > s_{slow}$. We set s_{slow} as the load factor of the DMPH table becomes 97%, or the overflowed cache is filled with half of the size. s_{slow} is set when the overflowed cache is filled with over 90% space.

As shown in Fig. 8, when ❶ the overflowed cache size reaches s_{slow} after an `Insert` request from a compute node, ❷ the memory node will return the status `PRE_RESIZE` to the compute node, and the compute node will create a new connection manager for preparing and listening to build a one-sided RDMA connection with the memory node. The memory node will return `PRE_RESIZE` to the data requests for all compute nodes in this shard and count up the number of compute nodes that got the information. After all the compute nodes get it or the overflowed cache size reaches s_{stop}, The memory node will build the one-sided RDMA connection (RC) with all compute nodes. The registered memory area in the memory node consists of five fields: (1) The value of the first eight bytes N_{cNode} indicates the number of compute nodes in this shard, but

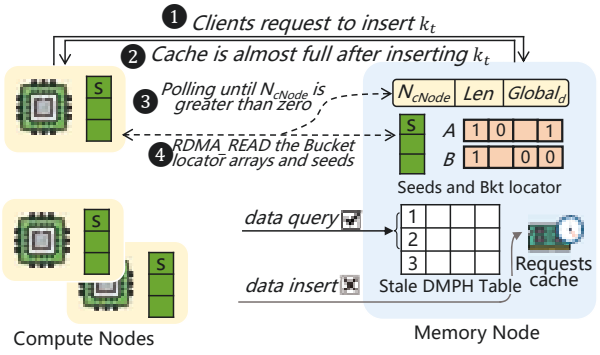


Figure 8: Index resizing in Outback.

it is set to zero at the beginning to indicate that the new index has not been completely reconstructed. After it finishes, the value will be set to the number of compute nodes in this shard; (2) the second value of the following eight bytes *len* refers to the total length of the newly written bucket locator arrays and seeds array; (3) $Global_d$ refers Global depth [32] value in current extendible hashing; (4) newly computed seeds array; and (5) bucket locator arrays A and B.

On the compute node, once a connection is established with the memory node, it continuously sends RDMA_READ requests to retrieve the first two values N_{cNode} and *len* in the registered memory of the memory node. If N_{cNode} is greater than zero, that means the bucket locator arrays and the seeds array have been successfully constructed and written into the memory area. ❸ The compute node then issues another RDMA_READ requests to fetch all the subsequent *len* data. Additionally, an atomic primitive of fetch-and-add `FAA` is executed to decrement N_{cNode} by one, signifying the completion of a compute node fetching the new index data.

Before the new bucket locator and seeds array is constructed, upon receiving an `Insert` or `Delete` request, the memory node returns a `FALSE` status to compute nodes. Then, the memory node caches the `Insert`/`Delete` requests and implements them later after the index data moves to the new DMPH table. For `Get` and `Update` requests, the memory node will continue serving it on the stale DMPH table. The reason is that no new data insertion would be implemented during resizing, and the keys' *ind_bucket* and *ind_slot* will not change.

Once all compute nodes have obtained the new bucket locator arrays and seeds, N_{cNode} in the memory node becomes zero. The memory node detects this change through periodic checks at a frequency of 2 times a second. It proceeds to discard the bucket locator arrays to free up memory space, as they will remain unchanged until the next MPH resizing. The memory node will also delete all moved keys in the stale DMPH table by marking the length field as 0. Then, the reliable connections with all the compute nodes will be terminated by the memory node, and all the compute nodes shift to use both the DMPH tables with the extendible hashing for processing data requests.

Note that all hash table-based disaggregated KVS require enlargement and shrinking capacity at runtime. The computation time for the extendible hashing layer is the same for Outback and prior works [14, 32, 66]. In Section 5.9, we will show the influence on Outback throughput during index resizing.

4.5 Analysis

In this section, we provide the theoretical analysis of the time complexity of the various data operations, as well as the estimation of the memory cost in both compute nodes and memory nodes.

Time complexity. For `Get` operations, each compute node is tasked with determining locations of the DMPH bucket and slot that stores the address of the requested KV. This involves two hash computations, namely $hash_A(k)$ and $hash_B(k)$, to access two bits in the bucket locator arrays. Subsequently, an additional hash computation with the bucket seed is performed to locate the specific slot. Then, the memory node can access the slot without further computation and proceed to read data from the referenced KV block. By default, we use a (2,4)-Cuckoo hash table [38] as a fallback table if no seeds can perfectly hash the four elements. In the worst case, accessing the Cuckoo hash table requires two additional hash computations and at most 8 key checks, resulting in a time complexity of O(1) for operations involving the Cuckoo hash table. Therefore, the worst case complexity remains O(1). For both the compute node and the memory node, the data `Get` operation incurs a small constant time. This time complexity extends to data update and data removal operations.

The only difference in `Insert` lies in the potential time overhead incurred in finding a new seed for the keys in the bucket. To address this, we have set a maximum number of trying times to 256 (8-bit seed). The reason is that we have not encountered a scenario in which no seed can be found within [0, 255] to separate those four keys without collision. We also have a fallback table (storing the key and the KV block address) to deal with rare cases when a group of keys appears that cannot be distributed into distinct slots by MPH. Statistically, we have observed no buckets that cannot be perfectly hashed with a seed length of 8. Therefore, the time cost associated with data insertion is also constant.

Memory usage. In compute nodes, the memory usage is allocated to the bucket locator and bucket seeds. According to Ludo [44], the bucket locator arrays consume 2.33 bits per key. The 8-bit seed is shared among four keys in a bucket. Assuming there are n KV pairs in a shard, with a load factor of ϵ for the MPH table, the memory cost in a compute node is calculated as $(2.33 + 2/\epsilon)n$ bits.

In addition to the underlying KV data, memory nodes allocate memory to encompass the latest bucket seeds, DMPH buckets, and the overflowed cache. Each bucket incurs a cost of 32 bytes, and the cache item contains the full key size and the data address. Given a cache size of m and a cache item size of c bits, the overall space budget (in bits) for indexing in a memory node is $66n/\epsilon + m \cdot c$.

4.6 Discussion

General applicability on traditional data structures. The design principle of Outback can boost data search in traditional data structures with the capability of serving range queries. Specifically, perfect hashing can boost the search process with one-time hash computation with low memory costs that can be cached in compute nodes. For example, the binary search in B/B+ tree leaf nodes can be replaced by perfect hashing computation by searching a seed for hashing keys in leaf nodes.

Ship computation to data. Outback decouples the process of DMPH into a memory-heavy component at memory nodes and a compute-heavy component at compute nodes and allows them to communicate via RDMA-RPC primitives. However, the memory accessing based on the given *ind_bucket* and *ind_slot* still needs a weak power computation unit close to data [55]. We can apply Outback to another two promising approaches without using two-sided RDMA verbs.

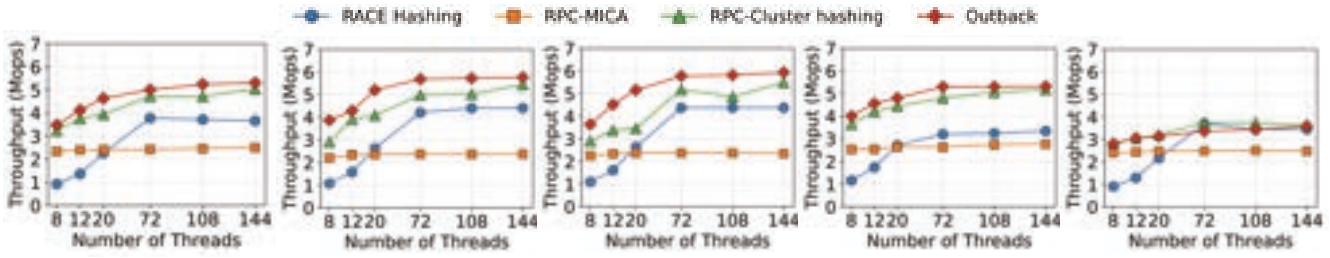
- *Extended RDMA READ verb.* PRISM [7] proposes and simulates an extended one-sided RDMA indirect reading verb `RDMA_READ` (`ptr` *addr*, `size` *len*, `bool` *indirect*), where *indirect* indicates if RNIC is supposed to read back the data pointed by the *addr*. This embedded one-sided RDMA verb can free the memory node's CPU and offload the memory reading task in Outback to RNICs. The reason is that Outback can get the exact requested data address without potential data probing.
- *Performance capacity of Outback with hardware accelerators.* In-network computation [63] has gained attention for accelerating data services in distributed systems by offloading tasks to in-network computation devices [18, 59] such as SmartNICs/DPUs and CXL [2]. The idea of Outback can reduce the computation burden on SmartNICs by employing one round-trip, one-sided RDMA_READ. For example, a SmartNIC [6, 30, 41, 45] can be placed on the memory node side, and function as an additional computation unit, and indirect data access tasks can be offloaded to it [51]. After the compute nodes in Outback issue a one-sided RDMA to read the queried key's slot and retrieve the address from the DMPH buckets, the SmartNIC can read the memory again via the PCIe switch and obtain the queried data through an additional PCIe round trip. The computation and data search tasks offloaded to the SmartNIC can be alleviated with the assistance of DMPH for the least computation burden.

Shared-nothing architecture. Outback utilizes a shared-nothing architecture [47] to prevent the update of cached seeds across compute nodes in different shards. The number of KV pairs in each shard depends on the overall size of the database and the number of shards. A greater number of shards results in fewer KV pairs on each memory node. Consequently, the memory allocation for DMPH seeds and bucket locator on each compute node can be reduced, although additional memory nodes are required. Determining the granularity for sharding KV pairs has always been a tradeoff [65], and it is recommended to choose the configuration based on the specific application.

5 PERFORMANCE EVALUATION

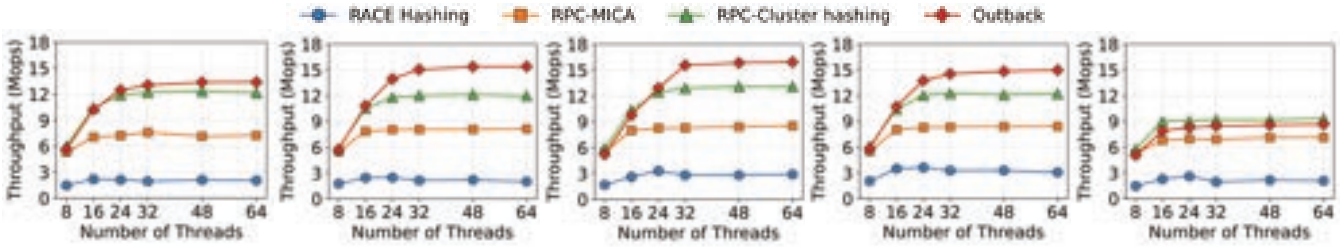
5.1 Methodology

Testbed. We run experiments in two environments. 1) 6 r650 machines from a public cluster CloudLab [15]; each of them is equipped with one Two 36-core Intel Xeon Platinum 8360Y CPU at 2.4GHz, 256 GiB DRAM and one Dual-port Mellanox ConnectX-6 (CX-6) 100 GbE NIC with Driver version as MLNX_OFED_LINUX-4.9-5.1.0.0. We conduct experiments with two shards, and each shard contains 3 machines. We use one machine as the memory node and the other two as compute nodes. The memory node registers the memory with huge pages to reduce RNIC's page cache misses, which is beneficial for memory-intensive applications [50, 66]. On compute nodes, we use two coroutines on each client thread to increase the query efficiency (See analysis in Section. 5.5). This is the default experiment environment unless otherwise stated. 2) 9 r320 machines in CloudLab [15], each of them is equipped with one Xeon E5-2450



(a) Workload A. (b) Workload B. (c) Workload C. (d) Workload D. (e) Workload F.

Figure 9: Throughput under YCSB benchmark with single memory node thread with Mellanox CX-6.



(a) Workload A. (b) Workload B. (c) Workload C. (d) Workload D. (e) Workload F.

Figure 10: Throughput under YCSB benchmark with Mellanox CX-3 RNICs.

CPU (8 cores, 2.1Ghz), 16 GiB DRAM, and one Mellanox MX354A Dual port FDR CX3 adapter. We use 1 machine as the memory node and the other 8 as compute nodes. We utilize 64-byte RDMA messages for all workloads to encapsulate various operation types (RC READ, UD SEND, and UD RECV), ensuring each request is padded to span two cache lines [21]. We do not use batching at any layer to minimize the latency in all evaluations.

Workloads. To evaluate the overall performance of Outback and other baselines, we employ YCSB [3, 12] workloads along with two diverse real-world datasets [35]. These datasets are (1) FB, encompassing a random assortment of Facebook user IDs to analyze patterns within social media interactions; (2) OSM, providing digitized infrastructure footprints from Open Street Map to represent geographical and spatial data usage; To ensure the datasets reflect general, unsorted data conditions, we shuffle them if initially sorted upon loading. Unless specified, we use 8B keys and 8B address values to configure all workloads like existing schemes [25, 28] for comprehensive evaluations. For each run, we precondition the memory node and warm up the database with 64 million KV pairs at first and then issue 10M requests to the benchmark on top of it.

Baselines. We develop a prototype of Outback based on RDMA libraries rlib and r2 [52] with over 4000 LoC in C++. We compare Outback with the other three baselines, one is a recently proposed one-sided RDMA scheme, RACE hashing [66], which utilizes RDMA RC READs for its operations; The other two are two-sided RDMA schemes that operate on RDMA SENDS/RECVs, differing in their underlying data structures – MICA [20, 29] and Cluster hashing [11].

- **RACE hashing.** RACE hashing [66] is a representative one-sided RDMA scheme developed recently. It offloads all data operations to compute nodes to free the memory node CPU with one-sided RDMA primitives. RACE Hashing adopts an RDMA-friendly hash table to combine the overflow bucket for collided keys and the hashed bucket. Thus, all the candidate buckets containing the requested key can be read back together. We develop RACE hashing with over 1,400 lines of C++ code, excluding the benchmark part that is shared with other baselines.
- **RDMA RPC-MICA.** RPC-MICA is a two-sided RDMA-based scheme with a data structure MICA [20, 29], which is an efficient hopscotch hash table and it has been used in existing two-sided RDMA [20, 22]. The overflowed KV pairs can be stored in the bucket adjacent to its hashed bucket. We implement hash computation for the bucket number on the compute node and send the queried key's fingerprint and bucket number to save computation on the memory node. We apply the open-source code from MICA [29] in our benchmark, utilizing it as the underlying data structure for the RPC-based approach without batching.
- **RDMA RPC-Cluster hashing.** RPC-Cluster hashing is a two-sided RDMA baseline with Cluster hashing, a chained-based hash table with associativity, running on memory nodes [11, 52]. The overflow keys that are hashed to a full bucket will be put in the linked indirect bucket. Each slot in a bucket includes 14 bits of fingerprint for key comparison. We apply the open-source code [4] of the cluster hashing as the data backend of our RPC-based scheme suit.

5.2 Performance on YCSB

Performance with CX-6 RNICs. We show the throughput of all evaluated methods by increasing the request load of running 8, 12, 20, 72, 108, and 144 compute node threads in a shard. On the memory node, we consistently allocate only one thread to run on a single core. As shown in Fig 9, these five figures illustrate the throughput and latency results under YCSB workloads A, B, C, D, and F, respectively.

`Get` and `Update` workloads (YCSB A and B). YCSB A and B workloads include 50% and 5% data `Update` respectively and the remaining is `Get`. Outback can achieve 5.50 and 5.82 Mops

throughput for YCSB A and B, as shown in Fig. 9(a) and Fig. 9(b). All other methods show lower throughput with the same number of threads. Outback can provide up to 1.07× and 1.06× throughput improvements on workloads A and B respectively, compared to RPC-cluster hashing. Compared to other RPC baselines with associative hash tables, the memory node in Outback is offloaded with less computation because it only needs to read the targeted key, and no data probing or traversing is needed to find the targeted value of the key. RACE hashing requires three round trips for updating data consistently, significantly increasing the latency and limiting the throughput. By comparing the results between workloads A and B, when more `Update` requests are issued, Outback spends more computation resources for value rewriting and key checking by reading the underlying KV blocks indicated by the computed MPH slot. Hence, Outback under YCSB B provides higher throughput than Outback under YCSB A.

`Get`-only workload (YCSB C). For `Get`-only workload, Outback can achieve 6.01 Mops throughput. When the number of compute node threads reaches 72, Outback outperforms RACE hashing, MICA, and Cluster hashing by 1.31×, 2.43×, and 1.11× on total throughput, respectively. The performance of RACE hashing is bottle-necked by its two round trips and the limited RNIC memory to cache queue pair (QP) state of a larger number of reliable connections. Outback reduces the average memory node's CPU time for data `Get` request with less computation overhead than the other two RPC-based baselines while looking up a key.

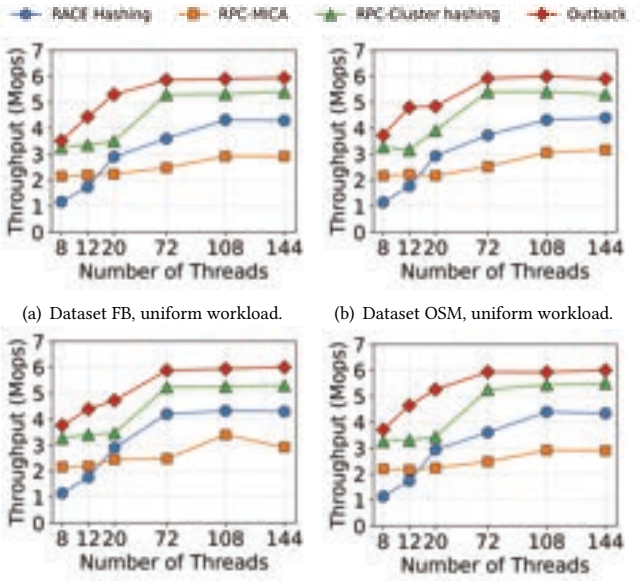
`Get` and `Insert` workloads (YCSB D and F). YCSB D contains 5% `Insert` and 95% `Get` operations. YCSB F contains 25% `Insert`, 25% `Update`, and 50% `Get` operations. Under YCSB D, Outback still shows the highest throughput among all methods. For `Insert` operations, Outback will check if a slot in the target bucket is available. Key-checking is also required, and a new seed will be calculated if the target slot stores an existing value. The high rate of `Insert` operations in YCSB F pulls the throughput down to 3.62 Mops, which is similar to RPC-Clustering hashing (3.64 Mops) when the number of client threads reaches 144.

Performance with CX-3 RNICs. As shown in Fig. 10, we show the throughput with the 4 memory node threads and a set of compute node threads numbers 8, 16, 24, 32, 48, and 64, respectively. Outback can consistently achieve the highest throughput for read-intensive workloads (A, B, C, and D). Significantly, Outback outperforms RACE hashing, MICA, and Cluster hashing by 5.03×, 1.79×, and 1.23× on total throughput for workload C, respectively. When we use a weaker CPU, the advantage of Outback is more significant. Unfortunately, CloudLab does not offer a weaker CPU with a high-performance network.

In summary, Outback demonstrates the highest throughput for most types of workload (YCSB A, B, C, and D). For a workload that is `Insert`-intensive such as YCSB F, Outback provides comparable throughput to other RDMA-RPC methods but still higher than that of one-sided RDMA.

5.3 Evaluations on Real-World Datasets

We leverage the SOSD datasets [35] for evaluations. Fig. 11 illustrates throughput results with the number of compute node threads as 8, 12, 20, 72, 108, and 144 in a shard. We set the number of memory node threads to 1. Each compute node thread issues 10 million



(a) Dataset FB, uniform workload.
(b) Dataset OSM, uniform workload.
(c) Dataset FB, zipfian workload.
(d) Dataset OSM, uniform workload.

Figure 11: Data `Get` throughput performance with SOSD datasets with uniform and zipfian-0.99 workloads.

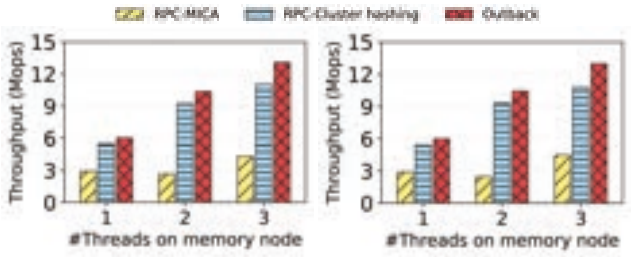
key lookup requests selected from the datasets in a uniform or zipfian distribution.

Compared to RACE, Outback achieves throughput of 1.38×, 1.35×, 1.39×, and 1.38× respectively on these four different settings when the number of threads reaches 144. RACE's performance is constrained by the multiple round trips. Compared to RPC-MICA and Cluster hashing, Outback achieves a throughput of 2.03× and 1.1× respectively on dataset FB when the threads number reaches 144 in Fig. 11(a). The reason that Outback can outperform them is that Outback can go directly to access data without extra check computation and indirect data accessing to probe the hash chain or buckets. Also, Outback outperforms RACE hashing, RPC-MICA and RPC-CLuster hashing by 1.35×, 2.05×, and 1.13× respectively on dataset FB when the workload follows the Zipfian distribution, as shown in Fig. 11(c). We observe the same trend in performance comparison with the dataset OSM.

5.4 Scalability with memory node threads

In this set of experiments, we vary the number of memory node threads from 1 to 3 and observe the throughput of different methods using real-world datasets FB and OSM. To exhaust the CPU resources on the memory node side, we use four r650 servers as compute nodes with 288 compute node threads.

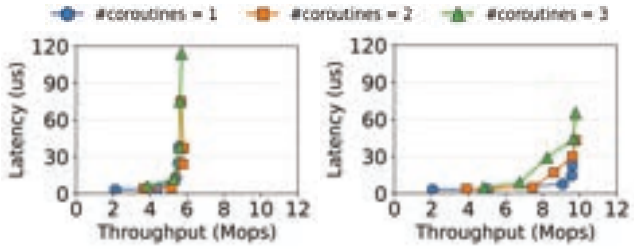
Fig. 12 shows the throughput of three RDMA-RPC schemes, by varying the memory node threads from 1 to 3. The throughput of Outback is around 1.10-1.21× of Cluster hashing and around 3× of MICA for dataset FB. The results of the two datasets exhibit the fact that as the number of compute node threads increases, the performance ratio between Outback and RPC-Cluster hashing/MICA remains similar. The reason is that Outback can ease the CPU burden on the memory node and allow it to handle more data requests from the compute node threads by offloading the computation of indexing to compute nodes.



(a) Scalability with memory node threads on dataset FB.

(b) Scalability with memory node threads on dataset OSM.

Figure 12: Throughput vs. the number of memory node threads.



(a) Latency-throughput curve on YCSB-C with 1 memory node thread.

(b) Latency-throughput curve on YCSB-C with 2 memory node threads.

Figure 13: Latency vs. the number of coroutines.

The fact that Outback achieves higher relative throughput to other RPC methods under a small number of memory node threads actually demonstrates the main advantage of Outback: achieving high performance when the memory node carries weak CPU power in a disaggregated memory system.

Note that the aim of Outback is not to saturate RNIC but to increase the throughput when there are limited CPU resources in a memory node with two-sided RDMA primitives. The results in this section show that Outback can achieve higher CPU efficiency with the same throughput goal, and Outback can realize higher throughput with the same CPU resources. In disaggregated systems, this can motivate the industry to satisfy the user's throughput goal with less TCO by reducing the CPU resources equipped on memory-optimized cloud instances [5].

5.5 Influence of the number of coroutines

The coroutines within compute node threads are designed to yield upon dispatching a request and resume operation upon receiving responses from two-sided RPCs. The default setup of Outback uses two coroutines per thread, but we extend our evaluation to explore the influence of one or more per thread to ascertain the optimal configuration for maximizing server CPU utilization. Fig. 13 studies the latency-throughput performance of Outback in YCSB-C workload with different numbers of coroutines in a compute node thread. In Fig. 12(a), we have only one worker thread in the memory node and vary the total of compute node threads as 8,20,72,144 and 216 distributed among three compute nodes, respectively. We can observe that a larger number of coroutines results in higher throughput when the number of compute node threads is less than 72, and the latency doubles or triples after the throughput reaches around 6

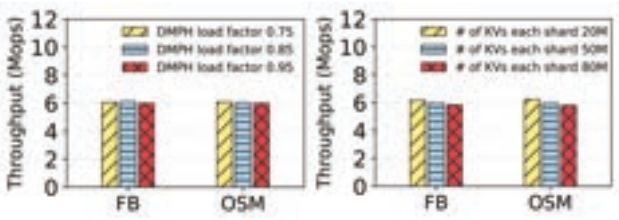


Figure 14: Influence of different load factor set in DMPH.

Figure 15: Influence of the varied number of KV pairs.

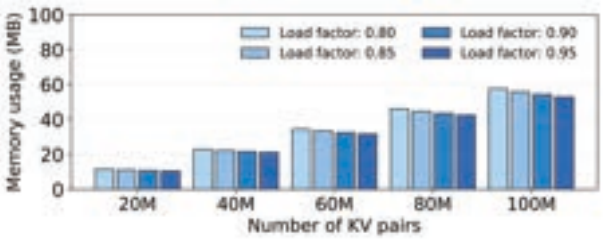


Figure 16: Memory usage on compute node with the varied number of KV pairs.

Mops, the maximum throughput one memory thread can support. This phenomenon is similar when the number of memory node threads is 2, as shown in Fig. 12(b), because the CPU resource on the memory node can handle 144 compute node threads, and the total throughput of a memory node can reach to 9.89 Mops. However, the extra coroutines will incur high latency of the data query after the number of memory node threads becomes a bottleneck for serving 216 threads.

5.6 Influence of load factor in DMPH

The load factor in a hash table is the ratio of stored elements to the total number of available slots or buckets. Maintaining an optimal load factor balances memory usage and data operation throughput. We evaluate the data `Get` throughput in Outback with varied load factors from 0.75 to 0.95.

As shown in Fig. 14, Outback can achieve around 6 Mops with 72 data query threads from compute nodes in a shard for the dataset FB. Similarly, the influence of the varied load factors on the throughput is trivial based on the results of the dataset OSM.

5.7 Influence of the number of KV pairs

Fig. 15 studies the impact of the number of KV pairs in each shard. We load 20M, 50M, and 80M KV pairs in Outback and evaluate the data `Get` throughput on two real-world datasets, respectively. Outback's read throughput decreases from 6.02 to 5.83 Mops as database size enlarges on the dataset FB. Similarly, we can observe the data read throughput decreases by 3.1% on the dataset OSM.

5.8 Memory usage in compute nodes

In a disaggregated memory system, compute nodes are regarded as the ones with rich computing resources but limited memory space. To make the memory node serve data requests with the least computation based on RDMA RPC primitives, we offload as much computation to the compute side with the help of DMPH. In this section, we evaluate the memory cost of Outback on each compute node with the varied number of KV pairs in each shard. The memory usage on a compute node consists of the bucket locator and the seeds array.

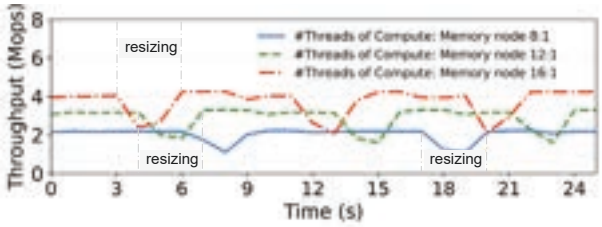


Figure 17: Influence of extendible hashing resizing.

As shown in Fig. 16, we vary the load factor used in the DMPH table from 0.80 to 0.95, and we use an 8-bit seed for keys in each bucket. The memory usage at each compute node for 20 million KV pairs per shard is around 12.5MB, and the cost is below 60MB for 100M KV pairs per shard. This is considered a small overhead because recent one-sided RDMA solutions cost hundreds of MBs or more on each compute node for index caching and other purposes [28, 50]. For example, in XStore [50], 100 million key-value pairs require over 600MB of memory at a compute node without including the cache.

5.9 Throughput during index resizing

We evaluate the throughput changes during index reconstruction and resizing. In this set of experiments, we bulk-load 20M keys to the database with the initial DMPH table to warm up, and we set one compute node with 8, 12, and 16 threads connecting to the memory node running only one thread, respectively. This emulates a challenging scenario because the memory node has limited computing resources to handle both resizing and lookups. The workload running on compute nodes is YCSB D, which contains 5% data insert and 95% read. As shown in Fig. 17, it takes around 3 seconds to recalculate the bucket locator and the seed for each bucket. Outback still supports partial `Get` requests during resizing with a decreased throughput by approximately 52% with only one thread in the memory node. The CPU contention causes a performance drop, and the performance goes back to normal after resizing.

5.10 Summary of evaluation

Data lookup throughput. Outback achieves 1.11-2.43× and 1.23-5.03× higher throughput than baselines with Mellanox CX-6 100Gb and CX-3 50Gb RNICs in data search workload, respectively.

Memory usage. The memory usage at each compute node for 20 million KV pairs per shard is around 12.5MB per shard, around 5 bits per key, with a load factor of 0.85 in DMPH.

Scalability of memory node threads. When the compute nodes with enough threads exhaust the compute capability on the memory node, Outback can achieve at least 18% performance advantage over other RPC-based baselines on read workload.

Load factors in Outback. The load factor value in DMPH causes a trivial impact on data lookup throughput with the same compute complexity. We recommend 0.8-0.9 to achieve the balance between memory usage and low frequent resizing, as the low load factor supports more incremental data insertion into the hash table.

6 RELATED WORK

RDMA-based storage systems. Existing RDMA-based storage can be classified into one-sided RDMA, RPC, or hybrid methods. One-sided RDMA-based approaches [9, 11, 14, 28, 34, 66] can bypass the memory node's CPU, managing data by RDMA_READ, RDMA_WRITE and other atomic verbs. Two-sided RDMA-based schemes [21, 22, 24, 31, 54] need only one round trip but suffer from the remote CPU bottleneck, posing challenges in saturating RNIC bandwidth due to the computation burden for the callback data service. The index data structures of existing two-sided RDMA, such as hash table [29, 36], learned index [27, 28] and Blink Tree [65], put the memory node's CPU in charge of nontrivial computation tasks. The hybrid methods [17, 20, 37, 52] combine two of the above approaches to boost the throughput.

In addition to examining design primitives and communication protocols within RDMA-based systems. Cowbird [9] frees the CPU burden in compute nodes by offloading RDMA posting tasks on in-network computation devices (e.g. programmable switch [19]), so that the compute node can focus on computation duties. SmartNIC [7, 41, 45, 51] can also be put in the network interface and works as an extra compute core on the critical data path, and it enables compute nodes to access data without network or RPC overhead. Note that the computation resource required in memory nodes of Outback can also be offloaded to SmartNIC or SmartSSD, whose SOCs are closer to data.

Minimal perfect hashing for networked systems. Perfect hashing offers a rapid method for data indexing, effectively preventing hash collisions. Moreover, DMPH enhances memory efficiency by eliminating the need to store keys and mapping N elements into $(1+\epsilon)N$ space within the table. Besides the Ludo hashing shown in § 2, Setsep [64] leverages a novel two-level hashing scheme that distributes billions of keys across cluster servers with a memory cost of $0.5 + 1.5l$ bits/key. BuRR [13] is another MPH scheme that involves manipulating a matrix for each key, and the multiplication values of keys determine various ranks within the bucket.

7 CONCLUSION

This paper introduces Outback, an RDMA RPC-based index for key-value stores on disaggregated memory, designed to achieve high throughput with lower CPU utilization. The key innovation of Outback is the division of the data index into two distinct components: a compute-intensive component cached on compute nodes and a memory-intensive component residing on memory nodes. The performance improvements stem from the memory node's ability to access underlying data with minimal computational overhead with perfect hashing. We also design protocols for Outback that support data operations and index resizing using extendible hashing, ensuring both the correctness of operations and system consistency during updates. We conduct extensive experiments to evaluate the performance of Outback. The results show that Outback achieves higher throughput and requires smaller memory space on compute nodes, compared to the state-of-the-art baselines under most types of workload, especially for `Get`-heavy workload.

ACKNOWLEDGMENTS

We thank our three anonymous reviewers for their insightful suggestions and comments. This research was supported by the IAB members of the Center for Research in Systems and Storage (CRSS), and the National Science Foundation (NSF) under grants CNS-1841545, CCF-1942754, CNS-2322919, CNS-2420632, CNS-2426031, and CNS-2426940. The views expressed are those of the authors and do not necessarily reflect those of the funding agencies.

REFERENCES

[1] [n.d.]. AMD Alveo™ Adaptable Accelerator Cards. https://www.amd.com/en/products/accelerators/alveo.html

[2] [n.d.]. Compute Express Link: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/about-cxl

[3] [n.d.]. https://github.com/basicthinker/YCSB-C.

[4] [n.d.]. https://github.com/SJTU-IPADS/drtm.

[5] [n.d.]. Memory Optimized Amazon EC2 Instance. https://aws.amazon.com/ec2/instance-types/?nc1=h_ls

[6] [n.d.]. NVIDIA BlueField Networking Platform. https://nvidia.com/en-us/networking/products/data-processing-unit

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

[8] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Citeseer, 30–39.

[9] Xinyi Chen, Liangcheng Yu, Vincent Liu, and Qizhen Zhang. 2023. Cowbird: Freeing CPUs to Compute by Offloading the Disaggregation of Memory. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 1060–1073.

[10] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on reliable connection with efficient resource sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–14.

[11] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–17.

[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[13] Peter C Dillinger, Lorenz Hübschle-Schneider, Peter Sanders, and Stefan Walzer. 2021. Fast succinct retrieval and approximate membership using ribbon. *arXiv preprint arXiv:2109.01892* (2021).

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

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[16] Edward A Fox, Lenwood S Heath, Qi Fan Chen, and Amjad M Daoud. 1992. Practical minimal perfect hash functions for large databases. *Commun. ACM* 35, 1 (1992), 105–121.

[17] Shukai Han, Mi Zhang, Dejun Jiang, and Jin Xiong. 2023. Exploiting Hybrid Index Scheme for RDMA-based Key-Value Stores. In *Proceedings of the 16th ACM International Conference on Systems and Storage*. 49–59.

[18] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. 2023. {CXL-ANNS}:{Software-Hardware} collaborative memory disaggregation and computation for {Billion-Scale} approximate nearest neighbor search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 585–600.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 121–136.

[20] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 295–306.

[21] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 437–450.

[22] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 185–201.

[23] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*.

[24] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. Reflex: Remote flash = local flash. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 345–359.

[25] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*. 489–504.

[26] Sekwon Lee, Soujanya Ponnapalli, Sharad Singhal, Marcos K Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2022. DINOMO: an elastic, scalable, high-performance key-value store for disaggregated persistent memory. *Proceedings of the VLDB Endowment* 15, 13 (2022), 4023–4037.

[27] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: a fine-grained learned index scheme for scalable and concurrent memory systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.

[28] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 99–114.

[29] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 429–444.

[30] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E Stephens, Hassan Wassel, and Aditya Akella. 2023. RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1293–1308.

[31] Yi Liu, Shouqian Shi, Minghao Xie, Heiner Litz, and Chen Qian. 2023. Smash: Flexible, fast, and resource-efficient placement and lookup of distributed storage. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7, 2 (2023), 1–22.

[32] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302* (2020).

[33] Baotong Lu, Kaisong Huang, Chieh-Jan Mike Liang, Tianzheng Wang, and Eric Lo. 2024. DEX: Scalable Range Indexing on Disaggregated Memory [Extended Version]. arXiv:2405.14502 [cs.DB]

[34] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R Lyu, and Yangfan Zhou. 2023. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association.

[35] Ryan Marcus, Andreas Kipf, and Alex van Renen. 2019. Searching on Sorted Data. https://doi.org/10.7910/DVN/JGVF9A

[36] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 103–114.

[37] Christopher Mitchell, Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. 2016. Balancing CPU and Network in the Cell Distributed B-Tree Store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 451–464.

[38] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[39] Xi Pang and Jianguo Wang. 2024. Understanding the performance implications of the design principles in storage-disaggregated databases. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[40] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. 2022. {RDMA} is Turing complete, we just did not know it yet!. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 71–85.

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

[42] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. 2022. Towards a fully disaggregated and programmable data center. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*. 18–28.

[43] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 81–98.

[44] Shouqian Shi and Chen Qian. 2020. Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 2 (2020), 1–32.

[45] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. 2020. StRoM: smart remote memory. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[46] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 33–48.

[47] Jianguo Wang and Qizhen Zhang. 2023. Disaggregated Database Systems. In *Companion of the 2023 International Conference on Management of Data*. 37–44.

[48] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A write-optimized distributed b+ tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data*. 1033–1048.

[49] Ruihong Wang, Jianguo Wang, Stratos Idreos, M Tamer Özsu, and Walid G Aref. 2022. The case for distributed shared-memory databases with RDMA-enabled memory disaggregation. *arXiv preprint arXiv:2207.03027* (2022).

[50] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *14th USENIX Symposium on*

Operating Systems Design and Implementation (OSDI 20). 117–135.

[51] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. Characterizing Off-path SmartNIC for Accelerating Distributed Systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 987–1004.

[52] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 233–251.

[53] Chenyuan Wu, Mohammad Javad Amiri, Jared Asch, Heena Nagda, Qizhen Zhang, and Boon Thau Loo. 2022. FlexChain: an elastic disaggregated blockchain. *Proceedings of the VLDB Endowment* 16, 1 (2022), 23–36.

[54] Minghao Xie, Chen Qian, and Heiner Litz. 2020. Reflex4arm: Supporting 100gbe flash storage disaggregation on arm soc. In *OCP Future Technology Symposium*.

[55] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. 2021. Ship compute or ship data? why not both?. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 633–651.

[56] Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. 2017. A concise forwarding information base for scalable and fast name lookups. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE, 1–10.

[57] Ming Zhang, Yu Hua, and Zhijun Yang. 2024. Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association.

[58] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. 2022. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 51–68.

[59] Penghao Zhang, Heng Pan, Zhenyu Li, Penglai Cui, Ru Jia, Peng He, Zhibin Zhang, Gareth Tyson, and Gaogang Xie. 2021. NetSHa: In-network acceleration of LSH-based distributed search. *IEEE Transactions on Parallel and Distributed Systems* 33, 9 (2021), 2213–2229.

[60] Qizhen Zhang, Philip A Bernstein, Daniel S Berger, and Badrish Chandramouli. 2021. Redy: remote dynamic memory cache. *arXiv preprint arXiv:2112.12946* (2021).

[61] Qizhen Zhang, Yifan Cai, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Rethinking data management systems for disaggregated data centers. In *Conference on Innovative Data Systems Research*.

[62] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the effect of data center resource disaggregation on production dbmss. *Proceedings of the VLDB Endowment* 13, 9 (2020).

[63] Changgang Zheng, Haoyue Tang, Mingyuan Zang, Xinpeng Hong, Aosong Feng, Leandros Tassiulas, and Noa Zilberman. 2023. DINC: Toward distributed in-network computing. *Proceedings of the ACM on Networking* 1, CoNEXT3 (2023), 1–25.

[64] Dong Zhou, Bin Fan, Hyeontaek Lim, David G Andersen, Michael Kaminsky, Michael Mitzenmacher, Ren Wang, and Ajaypal Singh. 2015. Scaling up clustered network appliances with ScaleBricks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 241–254.

[65] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing distributed tree-based index structures for fast rdma-capable networks. In *Proceedings of the 2019 International Conference on Management of Data*. 741–758.

[66] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 15–29.