Consistent RDMA-Friendly Hashing on Remote Persistent Memory

Xinxin Liu, Yu Hua*, Rong Bai
WNLO, Huazhong University of Science and Technology, Wuhan, Hubei, China
*Corresponding author: Yu Hua
E-mail: {xinxin, csyhua, bair}@hust.edu.cn

2021 IEEE 39th International Conference on Computer Design (ICCD) | 978-1-6654-3219-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCD53106.2021.00037

***Abstract*—Coalescing RDMA and Persistent Memory (PM) delivers high end-to-end performance for networked storage systems, which requires rethinking the design of efficient hash structures. In general, existing hashing schemes *separately* optimize RDMA and PM, thus partially addressing the problems of *RDMA Access Amplification* and *High-Overhead PM Consistency*. In order to address these problems, we propose a continuity hashing, which is a "one-stone-two-birds" design to optimize both RDMA and PM. The continuity hashing leverages a fine-grained contiguous shared region, called SBuckets, to provide standby positions for the neighbouring two buckets in case of hash collisions. In the continuity hashing, remote read only needs a single RDMA read to directly fetch the home bucket and the neighbouring SBuckets, which contain all the positions of maintaining a key-value item, thus alleviating RDMA access amplification. Continuity hashing further leverages indicators that can be atomically modified to support log-free PM consistency for all the write operations. Evaluation results demonstrate that compared with state-of-the-art techniques, continuity hashing achieves high throughput, low latency and the smallest number of PM writes with acceptable load factors.**

I. Introduction

High-speed networks and efficient persistent storage contribute to the high performance of cloud applications. Therefore, many schemes coalesce RDMA (remote direct memory access) and PMs (persistent memories) to deliver high performance [1], [2]. The coalesced RDMA and PM require rethinking the design of hash-based index structures. However, applying hashing schemes to RDMA and PM environments needs to address two main challenges: **RDMA Access Amplification.** RDMA is well-known for one-sided operations (e.g., read, write and atomic operations), which can bypass remote CPUs and provide better performance than two-sided operations over RC (reliable connection) mode [3]. However, a single one-sided RDMA operation only reads/writes one contiguous memory region. Therefore, accessing non-contiguous remote memory requires multiple one-sided RDMA round-trips. We refer to this problem as *RDMA Access Amplification*. **High-Overhead PM Consistency.** Due to the existence of volatile parts in PM-based systems (e.g., the CPU caches), in order to ensure crash consistency in case of a system failure, updating data larger than the 8-byte atomic write unit usually requires undo/redo logging or copy-on-write (COW) [2], [4]. However, double write operations in these mechanisms consume the limited endurance of PM. Moreover, guaranteeing write ordering typically needs the aid of flush/fence instructions, thus resulting in high system performance overheads.

Existing hashing schemes separately optimize RDMA or PM, and partially address the above challenges. Specifically, **RDMA-friendly hashing schemes** are usually designed to address the problem of *RDMA access amplification* [3], [5]. However, these RDMA-friendly hashing schemes fail to mitigate *High-Overhead PM Consistency*. For **PM-friendly hashing schemes**, many works have been proposed to guarantee crash consistency and optimize PM writes [4], [6], [7]. However, these PM-friendly hashing schemes typically cause *RDMA Access Amplification* due to indirect layers [4] or non-contiguous standby positions [6].

Unlike existing hashing schemes, we propose a coalescing hashing solution for both RDMA and PM, called continuity hashing, which mitigates *RDMA access amplification* and PM writes, as well as guaranteeing PM crash consistency. In the continuity hashing, two buckets with adjacent bucket numbers share a contiguous memory region between them, called shared buckets (SBuckets). These SBuckets provide standby positions for the neighbouring two buckets in case of hash collisions. A bucket and the neighbouring SBuckets build a segment, which contains all the potential positions of a specific KV item. Therefore, to read a requested record, clients only need a single RDMA read to directly fetch the corresponding segment, thus reducing the potential multiple RDMA round-trips. Write requests are handled by the server in order to simplify read-write competition and ensure consistency with low overheads. We use an indicator in the SBuckets for each two overlapping segments to indicate whether each slot in the two segments contains a consistent KV item. An indicator can be modified with an 8-byte atomic write, thus supporting log-free consistency guarantee on PM. Evaluation results demonstrate that compared with the PM-friendly level hashing [6] and the RDMA-friendly P-FaRM-KV [5], our continuity hashing achieves the highest throughputs (i.e., $1.45X$ – $2.43X$). For latencies, the continuity hashing has better search performance than P-FaRM-KV, and significantly outperforms the level hashing by an average of $2.19X$. The continuity hashing also has better write performance than level hashing, and further achieves a $1.99X$ improvement on average compared with P-FaRM-KV. Continuity hashing also achieves the smallest number of PM writes with acceptable load factors.

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 61772212.

978-1-6654-3219-1/21/$31.00 ©2021 IEEE
DOI 10.1109/ICCD53106.2021.00037

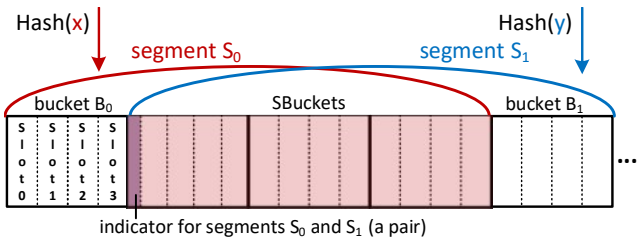

Fig. 1: Continuity hashing index structure.

II. The Design of Continuity Hashing

Continuity hash tables have two kinds of hash buckets, i.e., the numbered buckets and the unnumbered shared buckets (SBuckets). Specifically, the numbered buckets are addressable by a hash function, and two numbered buckets with adjacent bucket numbers share a fine-grained contiguous memory region between them, which consists of several (e.g., 3) unnumbered SBuckets, as shown in Figure 1. These SBuckets store conflicting key-value items when the corresponding home buckets (i.e., the neighbouring two numbered buckets) are full, playing a similar role of additional storage named stash. But the differences are that 1) we organize all the numbered buckets and the SBuckets into a contiguous memory region, and 2) we adopt a fine-grained shared region between each two numbered buckets. The design of the contiguous memory region for hash collisions comes from the RDMA property that each one-sided RDMA operation can only access one contiguous memory region. Therefore, all the potential positions of a specific KV item are in a small contiguous memory region, which can be fetched with a single one-sided RDMA operation, thus reducing potential multiple RDMA round-trips. The fine-grained shared region is designed as a suitable trade-off between the high space utilization of the hash table and the small size of the data retrieved via one-sided RDMA. The number of SBuckets in the fine-grained shared regions can be flexibly configured each time resizing occurs. We enable each bucket to contain multiple slots like existing schemes [4], [6].

As the basic unit of a one-sided RDMA read operation from clients, a segment is interpreted as a numbered bucket and the neighbouring several SBuckets. Each two segments overlap on the SBuckets. For the convenience of description, we call each two overlapping segments a *segment pair*. We use an indicator for each segment pair to indicate whether each slot in the two segments contains valid data. Specifically, the number of bits per indicator is the slot number in the corresponding segment pair, and each indicator is stored at the beginning of the corresponding SBuckets. Therefore, when a client issues an RDMA read to fetch a segment, the corresponding indicator indicates the slots with the valid data, without the need for another network round-trip. For the example in Figure 1, a 20-bit indicator for a segment pair is sufficient, which can be modified with an 8-byte atomic write.

The communications between the server and clients leverage the fast RDMA networking. For the procedure of remote reads from clients to the server, we explore the advantages of one-sided RDMA operations that do not involve remote server's CPU and have higher bandwidth and lower latency than two-sided RDMA operations [3], [5]. Clients compute the remote location of a requested key based on its hash value:

$$Bucket\_Number = hash(k)\%N \quad (1)$$

where $hash(k)$ is the hash value, and N is the total number of the numbered buckets. A one-sided RDMA read can access at most 1GB contiguous memory region. In order to reduce the number of network round-trips, clients only use one RDMA read to directly obtain the corresponding segment, which contains all the potential locations of the requested KV item in the continuity hash table. If the computed bucket number is even (e.g., bucket B_0), the offset of the segment to be read remotely (e.g., segment S_0) is:

$$Ofs = hash(k)\%N/2 * (size_{se} + size_{bu}) \quad (2)$$

If the computed bucket number is odd (e.g., bucket B_1), the offset of the segment to be read (e.g., segment S_1) is:

$$Ofs = (hash(k)\%N - 1)/2 * (size_{se} + size_{bu}) + size_{bu} \quad (3)$$

where $size_{se}$ and $size_{bu}$ are the segment size and the bucket size respectively. If the requested key exists, clients will find the KV item locally with the aid of the indicator. For the procedure of write requests, we put the burden of handling write requests on the server, and use server's CPU to guarantee persistence, like existing schemes [1], [2]. Specifically, clients query the server using RDMA write_with_immediate operations, which have higher throughput than RDMA send/recv operations over RC [3]. After receiving remote write requests, the server processes these requests and then notifies the clients that their requests have been completed.

The local writing process of the server provides log-free failure-atomicity guarantee. Existing hash tables use a 1-bit token that is associated with a slot to indicate whether the corresponding slot is empty [6]. We extend this design and group a set of tokens, called an indicator for each segment pair. An indicator is able to be updated in the atomic-write manner and, enables a log-free failure-atomicity guarantee for all the write operations on PM. **Atomic Insertion:** The server computes the home location via Equation (1), and then checks each bit of the corresponding indicator to find an empty slot. After the requested KV item is written to the empty slot, the server atomically sets the associated bit from 0 to 1. Even if a system crash occurs during writing the KV pair, the continuity hash table is still in a consistent state. Because the associated bit in the indicator has not been changed and thus the partial write is not visible. **Atomic Deletion:** After finding the KV item to be deleted, the server only needs to set the associated bit in the indicator from 1 to 0 in the atomic-write manner, and the KV item will be considered invalid by subsequent requests. **Atomic Update:** Continuity hashing adopts out-of-place update, which is a coalescence of insert and delete operations. To update a KV pair, the server locates the requested item, and further attempts to identify an empty

slot in the same segment. Since the old and the new locations of the requested item are associated with the same indicator, the server changes the values of the two corresponding bits in the indicator with an 8-byte atomic write. The update to the KV pair is invisible until the atomic update in the indicator is completed, thus ensuring crash consistency. **Log-free Resizing:** Resizing requires rehashing existing KV items into a new hash table. However, unlike the atomic update operations, the insertion and deletion for a KV pair during resizing cannot be completed atomically due to the updates to two different indicators. In fact, the operation sequence (first insert and then delete an item) ensures that data will not be lost in the event of a system failure. After restarting the server, we check the first existing KV item of the old hash table and perform a delete or rehash operation based on whether it has been inserted into the new hash table, thus restoring the hash table to a consistent state.

The continuity hashing aims to mitigate the *access amplification* via one-sided RDMA. In practice, not too high space utilization is acceptable due to the large capacity and the low price of the PM products [8]. But we still provide an optimization for the space utilization or the load factor (i.e., the ratio of the number of stored KV items to that of total storage units) without violating our design goals. We add a new scheme that dynamically increases the number of SBuckets for a small percentage of segment pairs before resizing. When the new SBuckets for a segment pair are added, the server will return the address and rkey of the added SBuckets region to the connected clients. Clients can locally know whether a segment pair has added SBuckets. Specifically, each segment pair will have at most one SBucket group added before resizing, and we empirically set the percentage of segment pairs with added SBuckets to $1/10$ by running different configurations. In this case, for a uniform read workload, a client needs at most two RDMA round-trips (one for the segment and the other for the added SBuckets) with a probability of 10%, and only one RDMA round trip with a probability of 90%. In addition, our proposed method still supports log-free consistency for all write operations on PM. Because the added SBuckets use the same indicator as the original buckets in the segment pair, and the indicator is able to be updated with an 8-byte atomic write.

III. Performance Evaluation

Our experiments are performed on two Linux machines, each of which is equipped with a Mellanox ConnectX-5 Infiniband HCA, two 2.1 GHz Intel Xeon $Gold6230R$ CPUs, 192GB DRAM and two 256GB Optane DIMMs. We generate our workloads via YCSB. YCSB-A is the update-heavy workload. YCSB-B is the read-mostly workload. YCSB-C is the read-only workload. YCSB-F is the read-modify-write workload, and consists of 50% reads and 50% read-modify-writes, where a record will be read, then modified and written back. Many schemes report that small-sized KV pairs dominate in production environment [9]. Therefore, we follow the setting of level hashing, where the key size is 16bytes, and the value size does not exceed 15bytes [6].

Our proposed scheme is compared with two state-of-the-art hashing schemes, i.e., level hashing [6] and P-FaRM-KV [5], [7]. Level hashing is a PM-friendly hashing scheme, and we add RDMA communication procedures to facilitate comparisons, i.e., using one-sided RDMA reads for remote read and RDMA write_with_immediate operations for remote write like our continuity hashing. FaRM [5] proposes an RDMA-friendly hashing scheme (FaRM-KV) for DRAM-based systems. We convert FaRM-KV into the PM counterpart (P-FaRM-KV) following the guidance of RECIPE [7]. Note that even if we change the structure of FaRM-KV and add a bitmap to each bucket to support consistency within a bucket, FaRM-KV still needs to use logging when the updates occur across buckets for consistency guarantee.

Throughput. Figures 2 – 4 respectively show the average throughputs with various workloads. For YCSB-A, the continuity hashing achieves $1.45X$ and $2.24X$ throughput improvements, compared with level hashing and P-FaRM-KV. The continuity hashing outperforms the PM-friendly level hashing, since querying data in the level hashing requires multiple one-sided RDMA round-trips. Moreover, continuity hashing also significantly outperforms the RDMA-friendly P-FaRM-KV, since the P-FaRM-KV fails to optimize PM writes and employs the expensive logging to guarantee consistency. For YCSB-C, we observe that for level hashing, the average throughput of searches significantly decreases, since for each search, the level hashing needs to issue multiple RDMA reads to query the standby positions. Both continuity hashing and P-FaRM-KV are optimized for RDMA reads, and each query only needs nearly one RDMA operation.

Latency. We record the execution time of each individual operation as latency. Figures 6 and 7 show the average latencies of two read-dominated workloads. We observe that compared with level hashing and P-FaRM-KV, the continuity hashing reduces the latencies by an average of 61% and 9%. Figures 5 and 8 show the average latencies of YCSB-A and YCSB-F, which contain a number of write operations. The average latency of writes increases with the number of threads due to locking mechanism. Compared with level hashing and P-FaRM-KV, the continuity hashing reduces the latencies by an average of 34% and 37%. As shown in Figure 9, for the latencies of PM update operations, our continuity hashing respectively achieves $1.39X$ and $2.14X$ performance improvements on average, compared with the level hashing and the P-FaRM-KV. We have optimized the insertion operations of P-FaRM-KV to reduce PM writes and latency by replacing the iteratively displacing KV pairs in the original scheme with at most one movement. The results further demonstrate our performance advantages over the two state-of-the-art schemes.

The Number of PM Writes. We evaluate the number of PM writes by counting the number of flush instructions, as shown in Table I. In our continuity hashing, each insertion and update needs to sequentially write the KV pair in an empty slot and modify the associated bit in the indicator from 0 to 1, thus including two PM writes. Furthermore, a deletion operation only needs one PM write that modifies the associated bit from

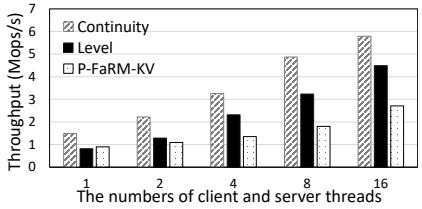

Fig. 2: The throughput of YCSB-A.

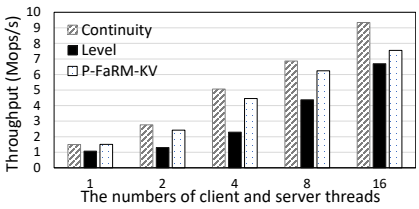

Fig. 3: The throughput of YCSB-B.

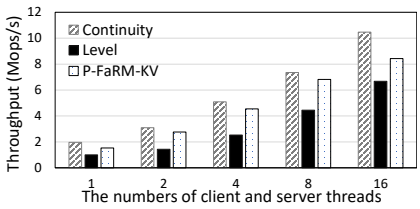

Fig. 4: The throughput of YCSB-C.

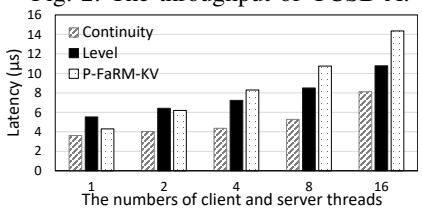

Fig. 5: The latency of YCSB-A.

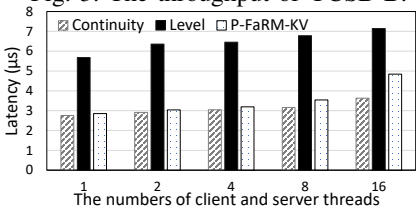

Fig. 6: The latency of YCSB-B.

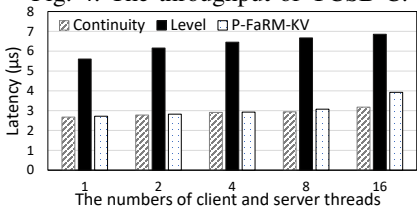

Fig. 7: The latency of YCSB-C.

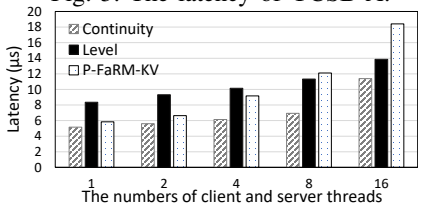

Fig. 8: The average latency of YCSB-F (the read-modify-write workload).

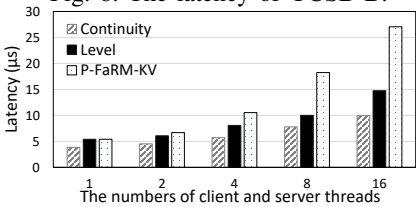

Fig. 9: The average latency of the update-only workload.

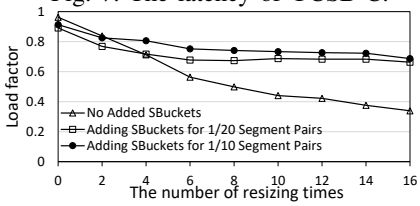

Fig. 10: The load factors of continuity hashing with different schemes.

1 to 0. In the level hashing, the number of PM writes for an insertion or an update is affected by the load factor of hash table. When the load rate increases, the probability of using logging to ensure crash consistency will also increase, which causes more PM write operations. P-FaRM-KV uses logging for each write operation, which incurs the highest number of PM writes among the three hashing schemes.

TABLE I: The number of PM writes with different operations.

	Insertion	Update	Deletion
Continuity	2	2	1
Level	2 – 2.01	2 – 5	1
P-FaRM-KV	5	5	5

Maximum Load Factor. We evaluate the optimization scheme that dynamically adds SBuckets for a small percentage of segment pairs to improve load factors. The evaluation uses YCSB-A workload. In Figure 10, we evaluate the load factors of continuity hashing and show the effectiveness of the added SBuckets in terms of space utilization. The initial hash table contains 20 buckets (i.e., 80 slots). Each resizing expands the hash table to twice the current capacity. We observe that as the number of resizing increases, the load factors of our original solution without added SBuckets gradually decrease. The optimization schemes with the added Sbuckets for $1/20$ and $1/10$ segment pairs achieve the load factors of about 70%, which is acceptable due to the large capacity of the available PM products [8]. We further evaluate the throughput of continuity hashing with different optional schemes. The figure is omitted here due to space limit. The results show that adding SBuckets for $1/10$ and $1/20$ segment pairs only slightly reduces the throughput of YCSB-A by 4% – 5%.

IV. Conclusion

Designing a high-performance hash structure is important for memory systems based on coalescing RDMA and PM. In order to address the problems of RDMA Access Amplification and High-Overhead PM Consistency, we propose the continuity hashing, a coalescing hashing solution for both RDMA and PM. The continuity hashing supports efficient remote read via a single one-sided RDMA operation and log-free consistency guarantee for all the write operations on PM. The evaluation demonstrates that our proposed continuity hashing achieves the high throughput, the low latency as well as the small number of PM writes, while obtaining acceptable load factors.

References

[1] J. Yang, J. Izraelevitz, and S. Swanson, "Orion: A distributed file system for non-volatile main memory and rdma-capable networks," in *FAST*, 2019.

[2] Y. Shan, S.-Y. Tsai, and Y. Zhang, "Distributed shared persistent memory," in *SOCC*, 2017.

[3] X. Wei, Z. Dong, R. Chen, and H. Chen, "Deconstructing rdma-enabled distributed transactions: Hybrid is better!" in *OSDI*, 2018.

[4] M. Nam, H. Cha, Y.-r. Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *FAST*, 2019.

[5] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *NSDI*, 2014.

[6] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *OSDI*, 2018.

[7] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: converting concurrent dram indexes to persistent-memory indexes," in *SOSP*, 2019.

[8] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[9] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *ASPLOS*, 2020, pp. 1077–1091.