HBTree: an Efficient Index Structure Based on Hybrid DRAM-NVM

Yuanhui Zhou
Wuhan National Laboratory for Optoelectronics
Huazhong University of Science and Technology
Wuhan, Hubei, China
zhouyuanhui@hust.edu.cn

Taotao Sheng
Wuhan National Laboratory for Optoelectronics
Huazhong University of Science and Technology
Wuhan, Hubei, China
shengtaotao@hust.edu.cn

Jiguang Wan*
Shenzhen Huazhong University of Science and Technology Research Institute
Wuhan National Laboratory for Optoelectronics
Huazhong University of Science and Technology
Wuhan, Hubei, China
jgwan@hust.edu.cn

Abstract—**Non-volatile Memory (NVM) with extremely high storage performance is the key storage device to build the next generation of storage systems. Various key-value (KV) store index structures have been designed for NVM, but these designs based on single level NVM suffer from significant write consistency overhead. The DRAM-NVM hybrid schemes improve the write performance but at the cost of very long recovery time. In this study, we proposed a novel DRAM-NVM hybrid index structure named HBTree (Hybrid B+Tree), which not only achieves better basic KV operating performance, but also greatly shorten the reconstruction time. HBTree uses the hot and cold characteristics of data access in actual application scenarios to cache the hot data into DRAM, so that improving the efficiency of KV operation. Meanwhile, it applies logging to ensure the data reliability of the write operations on the cached data. To reduce the system's failure recovery time, HBTree periodically backs up the index on DRAM to NVM. The experimental results indicate that for basic KV operation, the HBTree is 1.2∼1.7 times faster than FAST&FAIR and 1.1∼1.8 times faster than FPTree. Compared to FPTree, the recovery time of the HBTree is reduced by 70%.**

Index Terms—**Non-Volatile Memory, Index structures, DRAM-NVM hybrid storage, B+ Tree**

I. Introduction

Non-volatile memory (NVM), such as PCM [1], STTRAM [2], and RRAM [3], is a new type of storage device with characteristics of low latency, non-volatility, low power consumption, high storage capacity, and byte addressing. The first 3D X-Point [4] persistent memory (PM) product, Optane DC PMM, was released by Intel in April 2019 [5]. NVM breaks up the performance and capacity gap between DRAM and SSD/HDD, which can be connected to the CPU through memory interface as a large capacity main memory, and it can be directly used as a block device. Currently there are many hybrid storage systems composed of NVM and DRAM.

Indexing is a fundamental technology for persistent KV stores. The efficiency of KV operations, such as Put, Get and Delete, largely depends on the operation efficiency of the indexing structure, which makes it one of the research hotspots in the storage field. Conventional indexing structures are not suitable for NVM or DRAM-NVM hybrid memory systems because they are specially designed for hard disks or SSDs.

*Corresponding author: jgwan@hust.edu.cn

A large plethora of research focuses on optimizing indexing structures for PM based on conventional indexes. The indexing structures include LSM-Tree [6], Hash [7], B+tree [8], [9], B+Tree and hash hybrid structure [10], etc. These studies can be roughly divided into three categories. The first one optimizes indexing structures to adapt to the characteristics of NVM and improve system performance; the second reduces the consistency overhead on NVM and speeds up failure recovery; the third studies hybrid structure, and use DRAM to optimize system performance.

B+Tree is one of the most popular data structures. Compared with the Hash structure, B+Tree has better range query performance and its access method based on random read and write is more suitable for the characteristics of NVM [11]. Therefore, the B+ tree indexing structure is more suitable for PM. However, our experimental results in §2.2 show that FAST&FAIR [9], a representative solution to optimize the B+tree index structure on a single-layer NVM, is limited by the performance of the NVM and data consistency issues [12], [13]. Also, there is an obvious gap compared with the B+tree and log solution on DRAM. FPTree [8], another representative B+ tree indexing structure based on the DRAM-NVM hybrid structure, exploits the better performance of DRAM to improve indexing efficiency, but it sufferd from a long failure recovery time.

To improve the performance of the indexing structure and shorten the reconstruction time, HBTree, an indexing structure based on DRAM-NVM hybrid structure is proposed in this paper. The design of HBTree is based on the hot and cold characteristics of data access in actual application scenarios. The NvmTree where the hot data is located is placed in the DRAM and the consistency of the CacheTrees on DRAM is guaranteed through the log method, thus improving the read and write performance of the hot data. Besides, the index on the DRAM is backed up to reduce the system's failure recovery time.

The HBTree, FAST&FAIR, and FPTree are implemented and evaluated with the widely used Yahoo! Cloud Serving Benchmark (YCSB) on the real PM device, Intel's Optane DC PMM. The test results demonstrate that HBTree delivers the best performance, and it outperforms FAST&FAIR by 1.7x and FPTree by 1.8x at most on IOPS. Compared to FPTree, the recovery time of the HBTree is reduced by 70%.

The rest of the paper is organized as follows: In Section 2, the details of NVM and the challenges of the B+tree index for PM are introduced. In Section 3, the HBTree is proposed. In Section 4, the performance of HBTree is evaluated. Section 5 concludes this paper.

II. Background and Motivation

A. Non-Volatile Memory

Compared to traditional disk and flash, emerging Non-Volatile Memory (NVM), such as Optane DC PMM, is a new type of storage device that can provide better persistence, greater capacity, byte addressability, better persistence, and read latency similar to DRAM. Of course, NVM has weaknesses such as poor tolerance and asymmetric read and write performance. Due to the characteristic of persistence, the consistency of the persistent data in NVM has to be considered to ensure the correctness of system failure recovery.

B. Operational Efficiency

FAST&FAIR and FPTree are two representative optimization schemes for the B+tree indexing structure on NVM. The operational efficiencies of these two solutions are evaluated and compared with that of the B+tree on DRAM and that of the B+tree with log on DRAM solution (DRAM-B+tree-Log). Among them, the B+ tree on DRAM exploits the widely used stx-btree [14]. The DRAM B+tree combined with the log is denoted as DRAM-B+Tree-Log, and the log is used to maintain data consistency. The size of the B+tree node of all schemes is 512B, and the key and value size are both 8B. The operation efficiency of the basic operations of *Load*, *Put*, *Get*, and *Delete* is considered. The four solutions are evaluated as follows. First, 200 million key-value pairs are loaded. Then, 10 million key-value pairs are randomly put. Next, 1 million keys are randomly gotten, and finally, 1 million keys are randomly deleted. These tests are conducted in the same evaluation environments as described in §4.

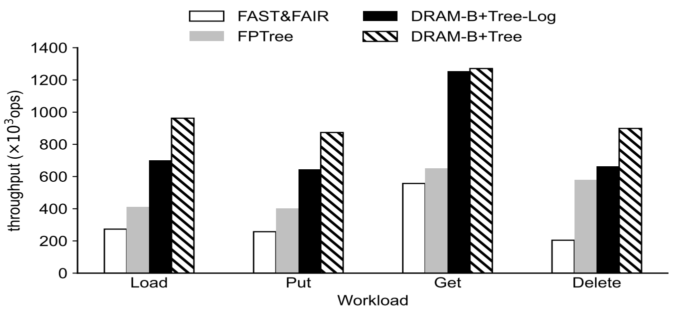

Fig. 1. Throughput of four indexing structures in Kops/s.

Figure 1 shows the throughput of the four indexing structures. Among them, FAST&FAIR performs the worst because it is built on a full NVM device. In this case, all operations need to access NVM, and there is a large amount of write consistency overhead; FPTree is built on a DRAM-NVM hybrid structure. The inter-mediate nodes of FPTree are placed in DRAM and the internal data of the leaf node is not guaranteed to be in order. In this case, the cache line refreshes, and the consistency overhead is reduced. Thus, FPTree performs better than FAST&FAIR for random write, but the disordered leaf nodes still affect its read performance. The write operation of the DRAM-B+tree-Log solution only needs to persist the Log record once, which further reduces the consistency overhead. Thus, this solution achieves significantly better write performance than FPTree. Meanwhile, the read operations of the solution are completed on faster DRAM, so its random read performance is 1.96x that of FPTree.

In summary, the performance of the FPTree based on a DRAM-NVM hybrid structure index is better than that based on NVM only but lower than the B+tree solution on DRAM. Compared with the B+tree based on NVM, DRAM-B+tree-Log has an obvious weakness because all the log data needs to be replayed during system restart. In this case, the solution takes a long time and occupies too many DRAM resources when the data is large.

C. Recovery Time

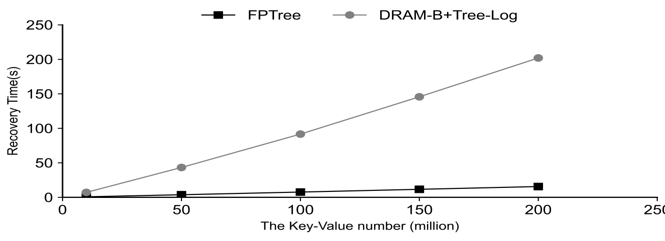

Fig. 2. Recover time for FPTree and DRAM-B+Tree-Log

When the system fails, the availability of the system is seriously affected by the recovery time of the indexing structure. Since FAST&FAIR is built on the full NVM, the data can be persisted. Thus, the recovery time of DRAM-B+tree-Log and FPTree is evaluated. It can be seen from Figure 2 that the recovery time is proportional to the data volume, and the recovery time of DRAM-B+tree-Log is far longer than that of FPTree. This is because FPTree only needs to rebuild the intermediate nodes on the DRAM according to the persistent leaf nodes in the NVM to restore the indexing structure. But when the data volume is 200 million, the recovery time of FPTree is close to 16 seconds, which still has a greater impact on the system availability.

III. HBTree Design

HBTree is proposed in this paper to improve operational efficiency and reduce the recovery time of existing PM indexing structures. It is a new tree index based on a DRAM-NVM hybrid structure, and it can improve read and write performance by caching hot data in DRAM to reduce write overhead. Besides, it writes logs on NVM to ensure data consistency and periodically backs up the index structure to NVM to speed up the system failure recovery.

A. General Architecture

HBTree is a hybrid three-layer persistent index. As shown in Figure 3, the HBTree's data structure includes the index layer, middle layer, and data layer. The design of the HBTree from the bottom layer to the top layer and the motivation behind the design are introduced in the subsequent content. **Data layer.** To achieve an efficient scan, B+Tree is exploited to

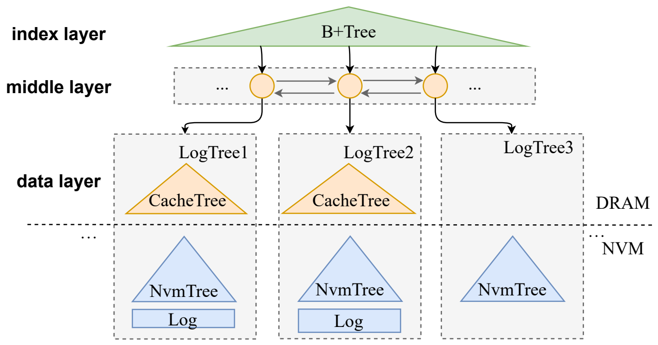

Fig. 3. Overview of the HBTree

manage data. However, due to the increasing height of B+Tree caused by the increase of data volume, more NVM accesses are required. To reduce NVM accesses and achieve efficient index operations, the global B+Tree is broken into numerous small B+ Trees called NvmTree with contiguous key ranges. Also, the hot NvmTree is cached in DRAM as CacheTree, and Log is used to ensure consistency. So, CacheTree, NvmTree and Log consist of LogTree are designed for the data layer.

Middle layer. To efficiently manage the Log-Tree in the data layer, the middle layer that is a double-linked list with each key representing a LogTree is designed. The middle layer also conducts statistics on the access frequency of each LogTree and identifies the hot NvmTree cached in DRAM to speed up the read and write operations. To reduce recovery time, all nodes of middle layer will be stored in NVM immediately. **Index layer.** It is a B+tree on DRAM to index the metadata nodes in the middle layer. It can quickly find the metadata of key ranges on the middle layer. The index can be built fast by traversing the middle layer without considering the consistency after the system is powered off or breakdown.

Overall, the HBTree can be regarded as a tree structure, and the leaf nodes are the LogTrees with small heights and contiguous key ranges. the HBTree has four characteristics. First, the index layer is all located in DRAM without consideration of consistency, which improves the efficiency of the overall index structure. Second, the combination of CacheTree on DRAM and Log on NVM reduces the data consistency overhead of the overall structure. Third, for a large data volume, the LogTree is used as a unit to identify hot and cold data, and only LogTrees with hot data have a CacheTree in DRAM , which effectively utilizes a small number of DRAM resources. Lastly, by backing up the CacheTree periodically, the recovery time in the case of system power failure is reduced. After the CacheTree is reconstructed, the index layer and middle layer with little data can be reconstructed fast, which contributes to a short recovery time of the index structure.

B. Hotspot Statistica Algorithm

The middle layer stores the metadata information of the LogTree, including the root node pointers of the CacheTree and NvmTree, Log, and temperature records of the LogTree. When the data in the LogTree changes, the cache node on DRAM and the persistent node on NVM is updated immediately.

The hot LogTree can be identified based on the access frequency of the LogTrees. So, the NvmTree of hot LogTree can be cached in DRAM and synchronized periodically, which reduces the access of NVM and improves performance. Of course, it is necessary to ensure consistency at a slight cost of writing logs.

Based on the historical access information of the LogTree, a heat statistics algorithm is proposed in this paper to calculate the temperature of the LogTree. If the historical access frequency of a LogTree is high, the LogTree is likely to be frequently accessed in the future period of time. The temperature of LogTrees is calculated through Equation (1):

$$T_{(t+\Delta t)} = A * T_t + Operate_{\Delta t} \tag{1}$$

Where T_t is the temperature at time t; A is a cooling coefficient in the range of (0, 1] with a default value of 0.5; $Operate_{\Delta}t$ is the number of accesses to the LogTree during the time interval Δt, and the initial temperature T_0 is set to 0. When a LogTree is split, the temperatures of the new LogTree are half of the original.

C. Data Layer

The data layer of the HBTree consists of LogTrees. The LogTree provides data persistence, fast recovery, and high-speed access, and it includes a CacheTree on DRAM and an NVMTree on NVM, and Log. Initially, LogTree only contains NvmTree, and all read and write operations access NVM directly. When the NvmTree is frequently accessed, the operation efficiency of CacheTree is low due to the limitation of NVM performance. Therefore, CacheTree is built on DRAM as the cache of NvmTree to improve the operation efficiency of the index. Meanwhile, writing logs are exploited to ensure consistency.

The write operations of CacheTree need to write logs to ensure data consistency. When the log of a LogTree is generated, the log space is allocated, and it is reclaimed later by a large log pool. Because the key and value are separated, only the operation type, key, and value pointer are recorded in logs. In this way, the size of log records is small and the write overhead is reduced. When a log is written, the log record is added first, and the current allocation address is then modified. Finally, the log record and the cache line of the modified address are flushed by the clflush command to ensure data persistence.

As for the data that is accessed highly frequently, there is a CacheTree inside the LogTree. The write request first appends the log, and the data is then written to the CacheTree. This avoids writing lower NVM directly and improves write performance. The read operations can directly query the CacheTree in DRAM. Besides, to reduce the recovery time needed by a large log record, the CacheTree will synchronously update the dirty node back to the NvmTree and then recycle the logs. If the LogTree does not have a CacheTree, the frequency of data access is small, and the delay of both operations directly accessing the NvmTree on the NVM is acceptable. The NVM guarantees the consistency of written data and does not write logs.

D. Dynamic Extension

To identify hot and cold data based on the granularity of LogTree, the proposed HBTree makes all LogTrees have a small height (H), thus avoiding large data in a LogTree. When the height of a LogTree (maximum height of a CacheTree or NvmTree) is about to exceed H, the Log-Tree will be dynamically expanded and split into two new Log-Trees, which have the same height as the original Log-Tree and no overlapping keyword ranges. If a CacheTree exists in the LogTree, the data in the CacheTree must be flushed to NvmTree before the LogTree is split, and then the CacheTree and NvmTree start to split. As an example, each step of splitting the NvmTree is illustrated in Figure 4. It is supposed that there is only NvmTree in LogTree A and the height of LogTree A exceeds H. First, the root node of the NvmTree is evenly split to generate two new nodes. The pointer of the key is 56 in the original root node, and it is used as the left pointer of the new node 2. Meanwhile, the key word 56 is added to the parent node in the index layer as a split key. Then, LogTree B is generated with the new node 2 as the root node of the NvmTree, and the metadata of LogTree B is added to the metadata node list in the middle layer. Subsequently, the NvmTree root node of LogTree A is replaced with the new node 1 to form LogTree A, and the original root node is recycled. Finally, the index data of its parent node is modified.

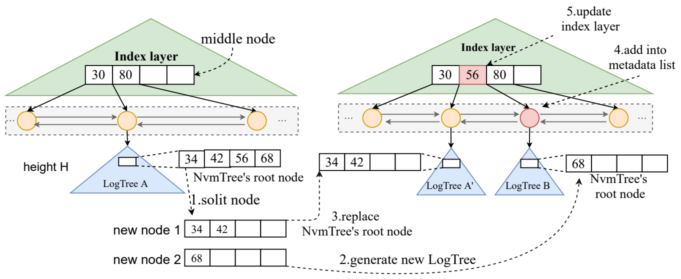

Fig. 4. NvmTree Split

E. CacheTree Management

Considering that the DRAM capacity is limited, only CacheTree is built for the hot LogTree and the CacheTree of the cold LogTree is deleted to save the total DRAM space occupied by the CacheTree. Based on this, higher performance can be achieved with less DRAM capacity.

CacheTree create. Because the structure of CacheTree is consistent with NvmTree, NvmTree can be copied from NVM directly, and the pointers connected between the nodes are then modified. The CacheTree is built in the background. In this case, the read operation will be blocked temporarily, and the write operation is finished directly by recording in a log only. When it is completed, the log will be played back.

CacheTree Synchronization. To quickly rebuild the CacheTree after the system is powered off, it is necessary to back up the CacheTree to NvmTree to avoid the big differences between the two structures. To speed up synchronization, the data marked as dirty nodes is synchronized each time. A new log will be created before the CacheTree is synchronized, and the old log is deleted when the synchronization is completed.

CacheTree Recycle. When the middle layer finds a cold LogTree, it needs to recycle the CacheTree. First, the read operation is paused, and the dirty nodes are updated to NvmTree. Then, all the nodes of the CacheTree are released, and the log record is played back to NvmTree. Finally, the NvmTree can work and the log will be deleted.

F. Data Consistency

Crash consistency is fundamental to persistent index structure. Since NVM has long write latency, the write operation on NVM with data size over 8B needs to consider consistency. First, CacheTree is exploited to reduce the write operation of NvmTree on NVM. Meanwhile, copy-on-write is adopted to ensure the consistency of all write operations of NvmTree. Besides, copy-on-write instead of overwrite operation is also adopted by CacheTree during synchronization, and atomic modification of pointers can ensure the consistency of the leaf node synchronization. In addition, if a crash occurs during dynamic expansion, the middle layer can be restored by traversing the NvmTree to determine whether there is data overlap.

Moreover, the log is exploited to ensure consistency, and the log is created by append write. After each log record is written, the log will be persisted, and the current point of the log will be updated. The pointer is atomic, and the log record is valid only after the pointer is persisted. So, the scope of the log can be determined by the starting address of the log and current address after the system is powered off.

G. Recovery

In this section, the recovery of the HBTree after the system failure is described. First, the middle layer is quickly restored by traversing the persistent metadata node linked list in NVM. If an unfinished split is found, the NvmTree continues to split, and the double-linked list of the middle layer is restored. Then, LogTrees are restored. The CacheTree is built by NvmTree, and the logs are then scanned to execute the valid logs in order. Finally, the index is restored. The index layer is small and can be recreated directly through the middle layer. It should be noted that after the failure recovery of the middle layer, there is no key overlap between the LogTrees. Therefore, the recovery of LogTrees can be performed in a multi-thread to reduce the recovery time.

IV. Evaluation

In this section, the performance of the HBTree is evaluated. The experimental setup is first described, and the basic performance, the performance under hot data, and the recovery time of the HBTree are then evaluated.

A. Experimental Setup

All the experiments are performed on a Linux server (Kernel version 5.10.1) equipped with two 24-core Intel Xeon Gold 5218R CPUs (2.30 GHz) and memory with a capacity of 64GB. The persistent memory used in the experiment is two Optane DC PMMs, each with a memory capacity of 128 GB. The maximum bandwidth of the PM for a single thread with 4KB I/Os is 2515 MB/s for sequential write, 2353 MB/s for random write, 2922 MB/s for random read, and 3144 MB/s for sequential read.

The implementation of the HBTree uses the libpmem library from PMDK [15]. The YCSB benchmark is adopted to evaluate the performance of the indexes, and Put, Get, Update, Delete, and Scan operations are conducted on this benchmark. The number of randomly generated scans is less than 100, and most of the Workloads use the default configuration. Among them, Load C uses the Latest to generate the set of hot data. All workloads employ 8-byte and 8-byte key-value pairs to fully reflect the performance of the indexes.

B. Operation Efficiency

The HBTree is compared with two representative PM-based index structures, i.e., FAST&FAIR and FPTree. The comparison is performed on uniformly distributed KV data. The YCSB benchmark first populates the index with 200 million keys (called Load) and then runs the respective workloads A, B, C, D, E, and F with 10 million put and/or read requests. The workload E performs scan operation, and the randomly generated scan count is less than 100. The node size of the B+ tree of all schemes is 512B. Besides, the cache capacity of the HBTree is set to 500MB. As shown in Figure 5, the HBTree performs significantly better than FPTree and FAST&FAIR on various YCSB workloads. The performance of the HBTree on the Load workload is equivalent to that of FPTree and is much higher than that of FAST&FAIR. The result of the workload A and F shows that the HBTree the HBTree performance improvement is relatively small for intensive write operations. This is because before the HBTree writes to CacheTree, it also needs to write logs to ensure data reliability and adds a persistence operation. Besides, no data movement is required by FAST&FAIR and FPTree to perform update operations, and only a small number of persistence operations are required. In this case, less performance improvement is achieved by the HBTree. The results of the workload B, C, and D show that the HBTree is more suitable for read-intensive workloads because when there is hot data, most of the read operations are performed on the faster DRAM through the CacheTree cache. The result of the workload E shows that the HBTree has certain advantages in range query, because the internal data of the HBTree leaf nodes are ordered, and part of the hot leaf nodes are cached.

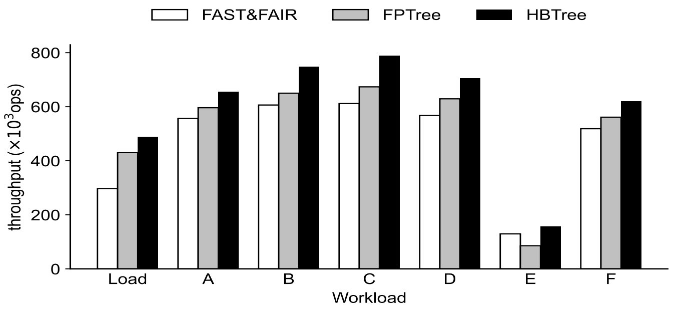

Fig. 5. Throughput on the YCSB workloads

C. Performance with Hotness Data

The performance of the HBTree in hotspot data accesses is further evaluated on the YCSB load A, load B, and load C workloads. The hotspot proportion of data is expressed as the ratio of the number of visits to the total number and visits to the hot data, and it is set to 50% to 80% in this experiment. The hot data in the dataset accounts for 20% of the total data. The YCSB benchmark first populates the index with 200 million keys (called Load), and then runs the workloads A, B, and C with 10 million put and/or read requests.

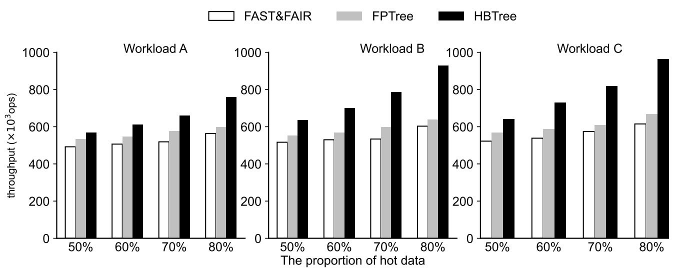

Fig. 6. Throughput under different data hotspots

Figure 6 shows that on workload A, the performance improvement of the HBTree increases with the proportion of the hots data. When the proportion of hot data is 80%, the performance of the HBTree is about 1.35 times that of the FAST&FAIR and 1.27 times that of the FPTree. Besides, the HBTree archives better performance on workloads B and C.

In general, the higher proportion of hot data, the better performance of the HBTree, and the greater performance improvement for more intensive read operations. The HBTree can achieve better read performance when the proportion of hotspot data is 80%. At this time, most read requests are completed by the CacheTree on DRAM, which reduces the number of read operations that are performed on relatively slow NVM devices and makes full use of DRAM's feature of fast I/O.

D. Recovery Time

To explore the rapid recovery capability of HBTree, this section tests the failure recovery time of HBTree under different data set sizes and different cache capacities and compares it with FPTree. To be fair, HBTree uses single-threaded recovery like FPTree. The cache size of HBTree is set to 500MB.

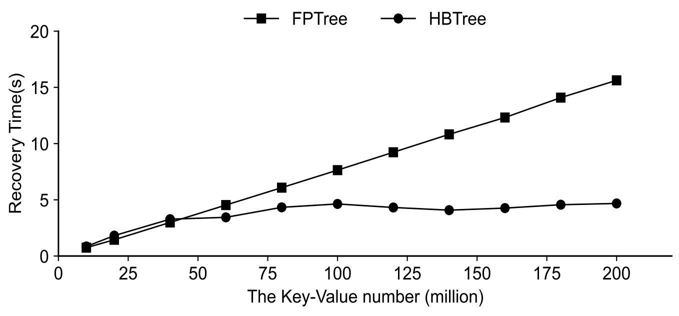

Fig. 7. Recovery time for HBTree and FPTree in various data volumes

Figure 7 shows that the recovery time of the HBTree is closer to that of the FPTree. With the increasing number of key-value pairs, the recovery time of the HBTree is maintained at a relatively stable level, while that of the FPTree is still increasing. The recovery operation of the FPTree is implemented according to the paper, and the time cost is mainly in the reconstruction of the middle nodes. The cost of the HBTree restoration is mainly concentrated on the restoration of the LogTrees. For the key-value pairs with data volume of 200M, the recovery time of the HBTree is only 30% that of the FPTree.

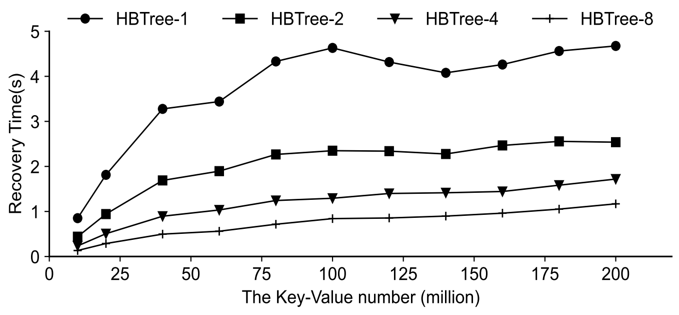

Fig. 8. Recovery time for different threads of HBTree

It is mentioned in Section III that the HBTree can be recovered with multiple threads. Figure 8 shows the recovery time of the HBTree under a different number of threads. As the number of threads increases, the recovery time continues to decrease. Meanwhile, the recovery time increases with the data volume. When the data volume exceeds 80 M, it tends to be flat because the recovery of CacheTrees in DRAM is completed quickly, but the recovery time of the B+ tree in the index layer is still long as the data volume increases.

V. Conclusion

In this work, a new tree index with a DRAM-NVM hybrid structure is designed. The index structure can make full use of limited DRAM resources to improve system performance and exploit the characteristics of NVM to ensure efficient data persistence and rapid recovery. The HBTree structure is implemented with the PMDK development tool library on the latest NVM equipment. On the general workload of the YCSB benchmark, the performance of the HBTree is 1.2$\sim$1.7 times that of FAST&FAIR and 1.1$\sim$1.8 times that of FPTree. Meanwhile, in the case of a large data volume, the recovery time of the HBTree is reduced by 70% compared with FPTree, and it achieves higher availability.

Acknowledgment

This work was sponsored in part by the National Key Research and Development Program of China (No.2018-YFB1003305), the fund from the Science, Technology and Innovation Commission of Shenzhen Municipality (No.JCYJ-20190809095001781), and the National Natural Science Foundation of China (No. 62072196).

References

[1] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In Proceedings of the 36th Annual International Symposium 14 on Computer Architecture (ISCA 09), pages 24–33, 2009.
[2] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, et al. Spin-transfer torque magnetic random access memory (STT-MRAM). ACM Journal on Emerging Technologies in Computing Systems (JETC), 2013, 9(2): 1-35
[3] F. Zahoor, TZ. Azni Zulkifli, FA. Khanday. Resistive random access memory (RRAM): an overview of materials, switching mechanism, performance, multilevel cell (MLC) storage, modeling, and applications[J]. Nanoscale research letters, 2020, 15: 1-26.
[4] F. T. Hady, A. Foong, B. Veal, D. Williams. Platform storage performance with 3D XPoint technology. Proceedings of the IEEE, 2017, 105(9): 1822-1833.
[5] Intel Optane DC persistent memory, 2019. https://newsroom.intel.com-/news-releases/intel-data-centric-launch/.
[6] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, R. Arpaci-Dusseau. Redesigning LSMs for nonvolatile memory with NoveLSM. USENIX Annual Technical Conference. 2018: 993-1005.
[7] P. Zuo, Y. Hua. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. IEEE Transactions on Parallel and Distributed Systems, 2017, 29(5): 985-998
[8] I. Oukid, J. Lasperas, A. Nica, et al. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory, In Proceedings of the 2016 International Conference on Management of Data (SIGMOD), 2016, 371-386.
[9] D. Hwang, W. H. Kim, Y. Won, B. Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In 16th USENIX Conference on File and Storage Technologies (FAST 18), Oakland, CA, USA, 12-15 Feb. 2018, USENIX, 2018: 187-200
[10] F. Xia, D. Jiang, J. Xiong, N. Sun. Hikv: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In 2017 USENIX Annual Technical Conference (ATC 17), Clara, CA, USA, 12-14 Jul, 2017, USENIX, 2017: 349-362.
[11] S. Chen and Q. Jin, “Persistent b+-trees in non-volatile main memory,” Proceedings of the VLDB Endowment, vol. 8, pp. 786–797, Feb. 2015.
[12] H. Volos, A. J. Tack, and M. M. Swift, Mnemosyne: Lightweight Persistent Memory. In Proceedings of 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11), Newport Beach, CA, March 2011.
[13] S. Haria, M. D. Hill, M. M. Swift. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In Proceedings of the 25th ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20), Lausanne, Switzerland, March 2020.
[14] Timo Bingmann. STX B+ tree C++ template classes. 2008.
[15] Persistent Memory Development Kit, 2019. https://github.com/pmem-/pmdk.