Towards Scalable Manycore-aware Persistent B+-Trees for Efficient Indexing in Cloud Environments

Safdar Jamil[1], Awais Khan[1], Bernd Burgstaller[2] and Youngjae Kim[1]
[1]*Sogang University, Seoul, Republic of Korea*
[2]*Yonsei University, Seoul, Republic of Korea*
{safdar, awais, youkim}@sogang.ac.kr, bburg@yonsei.ac.kr

***Abstract*—The emergence of manycore machines with Intel DC Persistent Memory (DCPM) aims to provide high performance and scalability with persistence guarantees. Thus, it is required to offer opportunities to port DRAM-based index data structures to DCPM to fully exploit the performance of these machines. Fast & Fair (F&F) is the state-of-the-art concurrent variant of the B^+-tree for DCPM. However, its adoption on manycore machines suffers from scalability limitations due to lengthy, lock-based synchronization including structure modification operations (SMOs). In this work, we propose F^3-tree, a concurrent, persistent future-based B+-tree that shows superior scalability on DCPMs. F^3-tree design relies on thread-local future objects and a global B^+-tree. We employ an in-memory hash table to mitigate the read overhead for the key searches in thread-local future objects. We implemented the proposed ideas atop F&F and performed experiments on Linux (kernel v5.4.0) using both synthetic and real-world workloads. We evaluated F^3-tree with F&F and the results show that F^3-tree outperforms F&F by 3.4x on average for sequential, random, and mixed workloads.**

I. Introduction

Cloud environments are facing unprecedented challenges due to the enormous data and the number of applications that must be handled [1]–[3]. Therefore, many cloud providers have deployed manycore machines, aiming to provide high performance and reduce the total cost of ownership by consolidating multiple users within a single server. However, the recent inclusion of Intel DC Persistent Memory (DCPM) in manycore machines brings in persistency at memory level [4] but severely degrades application scalability on manycore machines. Applications that run extensively on cloud services include databases and file systems that manage user-generated data [5]. These applications rely heavily on indexed data structures for high performance to access the data.

Several indexing methods have been proposed for DCPM [6]–[9]. The B^+-tree is one of the most popular index data structures used in databases and file systems. A few studies used B^+-trees on DCPM [7], [8], [10]. Fast&Fair (F&F) [8] is the state-of-the-art concurrent variant of the B^+-tree studied on DCPM. However, its adoption on manycore machines leads to scalability limitations. The write operations in F&F need to obtain a lock to ensure mutual exclusion, which becomes a point of contention when multiple threads attempt to access a B^+-tree node. Structural modification operations such as node splitting and merging increase contention when a thread triggers a chain of SMOs from a leaf to the root of the B^+-tree. This chain needs to acquire a per-node lock from the leaf to the root node where a number of threads are already trying to acquire the lock for the node.

Y. Kim is the corresponding author.

Lock optimization techniques such as MCS [11], FC-MCS [12], and HMCS [13] cannot solve the inherent scalability limitations of B^+-trees (refer to Section II-B). An alternative to lock optimization techniques is to employ scalable-friendly future objects (FO) [14], proposed to improve the performance of shared data structures. FOs are data objects that promise to deliver the results of an operation once the results become available. Futures are thread-local objects where each thread allocates its own FOs. The operations represented by FOs are applied to the shared data structure when their *evaluate* method is called. The evaluation of futures can be done in a flexible manner. For instance, threads can accumulate pending future operations to process all the FOs in a batch to perform a single operation on the shared data structure. Some operations may cancel each other out before taking effect over the shared data structure. Threads can delegate their FO operations to another thread.

Adopting futures for indexing data structures such as B^+-trees can improve their insertion performance, but it comes with its own challenges. Integrating FOs with a B^+-tree can cause consistency issues. The evaluate method must incorporate operations from thread-local FOs to the shared B^+-tree in a crash-resilient manner; otherwise, data loss may occur, leaving the B^+-tree in an inconsistent state. Thread-local FOs can severely degrade the read performance of B^+-trees. The read operations have to traverse the thread-local FOs to search for updated keys, which incurs an additional read overhead. Placing FOs in PM requires a durability guarantee; otherwise, after a crash, the FOs may be in an inconsistent state, such as missing pointers between thread-local FOs.

To address the aforementioned problems, we propose the F^3-tree, a concurrent persistent B^+-tree for PM-based manycore machines. The F^3-tree design relies on two important elements, i) thread-local FOs, and ii) a shared, global B^+-tree. Our approach is inspired by the producer-consumer design principle where application threads are only allowed to update the thread-local FOs (as producers), while the designated asynchronous threads are privileged to perform update operations on the global tree (as consumers). We converted DRAM-based FOs to persistent FOs and rely on durable linearizability as the

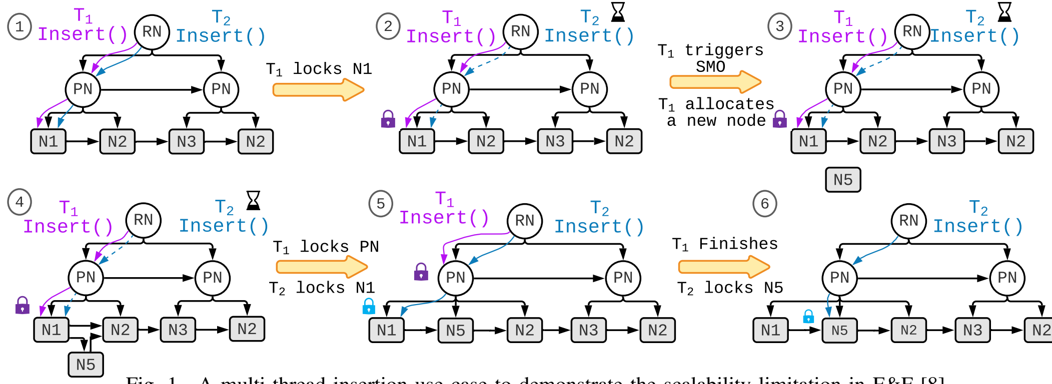


Fig. 1. A multi-thread insertion use case to demonstrate the scalability limitation in F&F [8].

correctness condition to guarantee crash consistency. The F^3-tree is equipped with two operational modes to update entries to the global tree, i.e., key-based and batch-based. In the key-based mode, the asynchronous evaluate threads checkpoint the keys within a single thread-local future object one by one to the global tree. In the batch-base mode, the asynchronous evaluate threads checkpoint a single whole future object to the global tree. To mitigate the read performance, we employ an in-memory hash table to avoid key search in thread-local future entries. We evaluate the proposed F^3-tree against baseline F&F with sequential and random workloads. The experimental results confirm an average performance improvement of 3.4x for both workloads.

II. Background and Motivation

A. *Fast & Fair: B^+-tree*

A recent study, F&F, proposed a DCPM-based durable and concurrent B^+-tree, that provides lock-free reads [8]. F&F completely avoids the logging overhead by transforming a B^+-tree to another consistent state or a transient inconsistent state that readers can tolerate. The reads detect and tolerate inconsistencies such as duplicated elements in a sorted list. Writers hold a lock for mutual exclusion. The writes detect inconsistencies such as duplicated elements, and try to fix them.

F&F is composed of two algorithms, Failure Atomic Shift (Fast) and Failure Atomic In-place Rebalance (Fair). Fast is used to insert the keys within a node of the B^+-tree by performing atomic shift operations to maintain the sorted order of the keys. Since B^+-tree node is an array of entries, the shift operation is a sequence of load and store instructions in cascading order, and it maintains the total store order. This also helps in avoiding excessive calling of `FLUSH+FENCE` instructions, as the updated entries within the B^+-tree node can be flushed together. Maintaining a consistent view of the tree-based indexing data structure during the structural modification operations is one of the challenging tasks since it requires for additional caretaking in terms of logging. Logging becomes an additional overhead as it duplicates the number of pages, increases the write traffic, and blocks the concurrent access of tree nodes. The fair algorithm avoids the use of logging by maintaining the sibling pointers in the B^+-tree nodes and creates a B-link tree [15].

B. *Motivation*

F&F in highly concurrent write scenarios face scalability limitations on manycore machines. For instance, the critical section of F&F is composed of several sub-operations, i.e., linear search, acquiring MUTEX lock, shift operation, and SMOs. Figure 1 shows the concurrent write operation in F&F, where two threads, T_1 and T_2, perform insert operation. Both threads look up the candidate node for the key insertion, as shown in Figure 1 ①. Note that both threads select the same node as the candidate node for key insertion. However, F&F uses MUTEX locks for mutual exclusion; therefore, only a single thread acquires the lock and proceeds with insertion. As shown in Figure, T_1 wins and acquires the lock and proceeds with the insert operation, ②. At this point, T_1 checks the capacity of the candidate node and triggers the SMO, as the candidate node (N1) capacity is full. During SMO, T_1 allocates a new node, migrates half of the entries from N1 to the newly allocated node N5 and inserts the key into the corresponding node, steps ③ and ④. At step ⑤, T_1 acquires the lock at parent node PN and updates the links accordingly. Meanwhile T_2 is able to acquire the lock at node N1 and notices that the N1 has been split, and now the key needs to be inserted in the new candidate node N5. So, at step ⑥, T_2 releases the lock at node N1 and acquires the lock at new candidate node N5 and performs the insert operation.

This blocking mechanism limits the scalability of F&F on manycore machines where hundreds of application threads are contending to perform insert operations.

III. Design and Implementation

A. *System Overview*

Figure 2 shows the overall design of our proposed system. Our approach is inspired by the asynchronous computation design principle, i.e., producer and consumer model. Our design consists of three components, per-thread local future objects (PTFO), global B^+-tree, and in-memory HT (HT). PTFOs act as thread local buffers and allow application threads (producers) to perform write operations locally. We adopt the doubly linked list design for the PTFO. In the meantime,

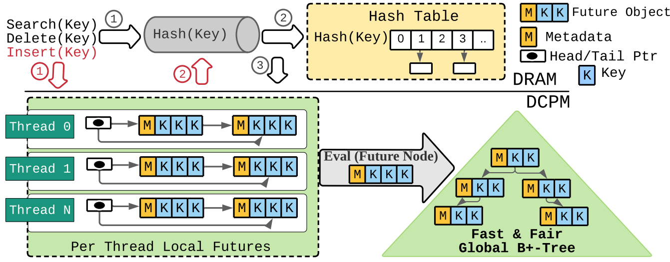


Fig. 2. F^3-tree design overview

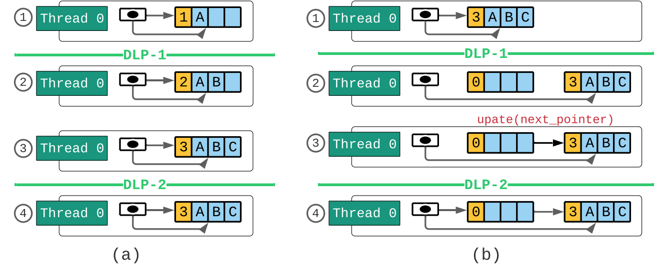


Fig. 3. Durable Linearizability examples. DLP represents the DL point achieved. (a) shows the single FO achieving DLP while (b) shows the multi FO DLP.

PTFOs are checkpointed to the global B$^+$-tree through the evaluate function, where a dedicated worker thread pool serves as consumers. To minimize the search query performance for keys resident in PTFO, we adopt an in-memory hash table (HT). A key is inserted into the HT after it is durably written to the PTFO.

B. Operation Flow

We show the operation flow for the insert, search, and delete operations in Figure 2. The red encircled steps show the insert operation flow. The insert operation will first insert the key-value pair into the thread local buffers, i.e., into the thread's PTFOs. Once the key is successfully written, it is pushed to the HT. Note that the delete and search operations use a different path compared to insert. Both delete and search perform a hierarchical key lookup, i.e., i) a lookup in the HT; If the key is not found, then ii) a lookup is performed in the corresponding thread local linked list followed by iii) the global B$^+$-tree. Note that the lookup complexity increases as the search operation progresses to higher hierarchies due to the increased search space.

The evaluate operation works as follows: each evaluate thread is responsible for a particular thread-local linked list and it checkpoints each linked list's FO to the global B$^+$-tree once it meets a given threshold. We defined two thresholds for the checkpoint of PTFOs, i) time-based and ii) based on the number of KV pairs. A PTFO is accessed using the tail pointer of the linked list by the evaluation threads when either one of the thresholds is met. Once a PTFO is checkpointed, the tail pointer is updated atomically to the previous node.

A significant factor that affects performance is the size of the PTFOs. If there is no limit to their size, then producers do not get blocked by consumers that checkpoint those FOs to the global B$^+$-tree. On the contrary, limiting the number of PTFOs would degrade the performance of the producer threads. Once the threshold of future object allocation is met, the producer thread has to wait for the consumer threads to checkpoint the data to the global B$^+$-tree, so that the producer thread can service further requests. Producer threads are blocked for two major reasons. First, a too small number of consumer threads limits the scalability of the producers. Second, if we increase the number of consumer threads, the evaluate operation eventually meets the inherent scalability limitation of the global B$^+$-tree.

During the recovery phase after a system crash, the asynchronous `eval()` threads will checkpoint the PTFOs from before the crash to the global tree. In addition, for application threads, new PTFO and HT will be allocated in DCPM and DRAM, respectively. This will allow the application to continue its execution without waiting for the recovery process to finish.

C. Thread Local Future Objects

A future is a data object that promises to deliver the results of an operation when ready [14]. In this work, we define future objects as an array of keys, entry count, next and previous FO pointer, as shown in Figure 2. Every thread has a dedicated Local Future Object stitched in a doubly linked list.

The reasons to use a doubly linked list are as follows. First, it mitigates the contention between the producers and the asynchronous consumers, and we only allow producers to modify the linked list from the head pointer while the consumers flush from the tail of the linked list, as shown in Figure 2. Second, for checkpointing all the entries to the global B$^+$-tree even after a crash. For instance, if a crash happens in between updating the head point to the newly allocated FO (as shown in step ② and step ④ of Figure 3(b)) then the recovery mechanism would still be able to access the new node by traversing the thread local linked list through tail pointer.

With PM, providing a consistent view of PTFO is critical. Note that we do not rely on any existing locking mechanism while consuming data from PTFO. Therefore, we adopt durable linearizability to ensure that each PTFO takes effect in a sequential order.

D. Durable Linearizability (DL)

A concurrent data structure is linearizable if each operation takes effect in between the method's invocation and response [16]. With DCPM, a durability guarantee is additionally required because the data will be persistent and need to be crash resilient. A *durably linearizable* concurrent data structure satisfies the linearization property. In addition, after a full-system crash (i.e., all threads crash), the state of the data structure must reflect a consistent sub-history of operations that includes all operations completed by the time of the crash. We use the term *durability point* as the point in the execution history E where an operation becomes durable, i.e., its effects

are visible to other threads and persistent. After a durability point, if we execute the recovery mechanism, then the data structure will be in a consistent state. The durability order of an execution can be elaborated in terms of durability points, where each durability point implies an order on the operations.

For the F^3-tree, we achieve DL for the PTFO as shown in Figure 3. We do not consider the same for the global B$^+$-tree because it follows the same design principle of F&F. Figure 3 shows two examples for achieving a DL point. Figure 3(a) shows an example where we achieve the DL within a single FO by atomically updating the key-value pairs. Steps ① to ④ in Figure 3(a) show the write operation within an FO. The lines labeled DLP represent the DL points achieved by the insert operation by calling `FLUSH+FENCE` instructions. In Figure 3(a), there are two DLPs achieved, DLP-1 and DLP-2. If a crash happens in between DLP-1 and DLP-2 (step ② and step ③) the recovery mechanism will be able to achieve a consistent view of the thread-local linked list by DLP-1.

Figure 3(b) shows an example of the second scenario where a new FO is updated within the thread-local linked list. Steps ② to ④ show the allocation of a new FO in the thread-local linked list. In this scenario, the DLP is achieved once the next pointer of the newly allocated FO is updated atomically, followed by the `FLUSH+FENCE` instructions. We call the `FLUSH+FENCE` instruction pair right after updating the next pointer of the new FO so that if a crash happens after the DLP, we are able to access the new FO by backward traversing. If a crash happens before the DLP-2, our recovery mechanism will be able to achieve the consistent state of the thread-local linked list to the DLP-1. There is a potential memory leak that needs to be addressed if a crash happens in between steps ② and ③. There are several mechanisms that can be adopted for memory leaks such as hazard eras [17] and the optimistic access scheme [18].

E. Space and Time Complexity

The space overhead of our proposed design is similar to the traditional B$^+$-tree, i.e., $O(N)$, because a key is either in the PTFO or in the global B$^+$-tree. Also, the in-memory HT is placed in DRAM and not DCPM, so we do not consider its space overhead for DCPM. Though, for DRAM the HT space overhead is $O(M)$, where M is the number of keys stored in the PTFO entries at a particular time instance.

The time complexity for the lookup operation is composed of two cases, one where a thread is required to traverse the PTFO and second where a thread only looks for the key in the global B$^+$-tree. In addition, the read operation has to go through the in-memory HT. Now, if a thread is looking for a key it has to first search the hash of the key in the HT, which has constant time complexity $O(1)$. If the key is found in the HT then the thread will traverse the particular PTFO linearly and return once the key is found. The time complexity (T) for this case is $O(1) + O(M)$. For the second scenario, if the key is not found in the HT, then the thread will directly look for the key in the global F&F tree. F&F offers lock-free read operations that are also based on linear search.

IV. Evaluation

A. Testbed Setup

We performed our experiments on a Linux machine (kernel v5.4.0) equipped with 4 Intel Xeon(R) E5-4640 v2 CPUs @ 2.20 GHz with 10 physical cores per node, 80 MiB last level cache, and 256 GiB DDR3 DRAM. We enable hyper-threading to increase threads for scalability evaluation. We emulate the latency of Intel DCPM as presented in [19]. We implemented the proposed F^3-tree on top of F&F. We used a synthetic benchmark with one million 8-byte key-value pairs with sequential and random key distribution. We bind the threads first to the first CPU node using the numactl command and then move to other nodes.

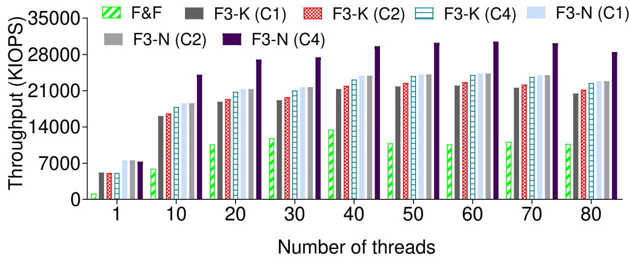
We compared the proposed approach to two different variants of F^3-tree, i.e., key-based (F^3-K) and node-based (F^3-N). In the former variant, the evaluate thread consumes the keys from PTFOs in a sequential order and checkpoint to the global B$^+$-tree. The later variant benefits from batching, i.e., a single PTFO consists of multiple keys and the evaluate thread checkpoints the entire future object to the global B$^+$-tree in a single operation. Therefore, the node-based F^3-tree variant imposes strict key sorting inside a single PTFO.

B. Results

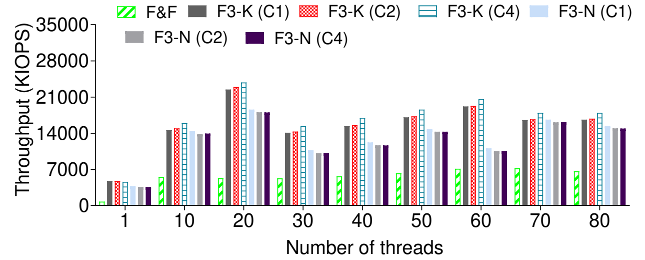
1) Sequential Workload Analysis: Figure 4(a) depicts the performance for sequential workloads. We clearly observe that, on average, F3-N outperforms F3-K with 1.3x and 3.3x compared to F&F. The reasons are manifold. First, F3-N benefits from the sequential order of keys within the workload and does not explicitly perform sorting. Second, F3-N checkpoints the whole PTFO to the global F&F tree, which leads to fewer shift operations and SMOs on the global tree. Third, F3-N incurs less thread synchronization and communication overhead with foreground threads. Similarly, F3-K also outperforms the F&F tree on average 2.4x on average due to its PTFO design. We also observed a scalable trend in F&F performance with varying threads for sequential workloads because F&F does not perform frequent shift operations due to sequential key order. Moreover, the workload is equally distributed among threads, leading to less contention on a single B$^+$-tree node.

Notably, we observed a scalable trend by all approaches for threads within a single CPU node. However, performance degrades with threads crossing the single CPU node boundary due to high remote memory accesses. Although the F^3-tree writes to PTFO still the asynchronous threads read the PTFO and checkpoint them to the global tree and thus suffer from remote memory accesses and performance saturation. We plan to address the NUMA issue (remote memory accesses problem) in our future work. Figure 4(a) also shows a similar trend for throughput with varying numbers of consumer threads, i.e., evaluate threads.

2) Random Workload Analysis: Figure 4(b) shows the results of the random workload. We observed that F3-K outperforms F3-N due to two major reasons. First, F3-N traverses the whole global tree to check if the key-value pairs within the local future object overlap with the key-values pairs



(a) Sequential insert workload



(b) Random insert workload

Fig. 4. Scalability analysis of F^3-tree on manycore machines. In F^3-K(C_i) and F^3-N(C_i), C_i represents the number of asynchronous evaluate threads.

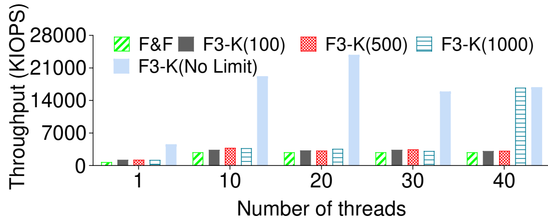


Fig. 5. PTFO size impact on performance.

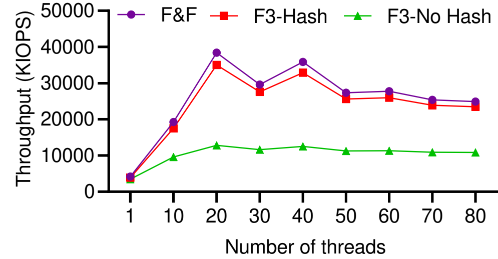


Fig. 6. Search performance analysis.

of any of the global tree nodes. Second, F3-N performs a double shift operation, i.e., on the PTFO and the global tree.

With the random workload, F&F performance saturates as the number of threads increases, i.e., after eight threads. This is due to excessive amount of shift operations, during non-SMOs and SMOs, and the increasing contention over the MUTEX lock of the tree node. With a smaller number of threads, we observed that shifting is the dominant operation of F&F because it has the highest execution-time contribution. Whereas, when the number of threads increases, contention to acquire the lock and shifting become the dominant factors for the performance saturation of F&F.

In addition, the F^3-tree suffers from remote memory accesses in the random workload as well. This is because, with the random workloads, the keys are distributed randomly between PTFOs, unlike the sequential workload. The asynchronous evaluate threads suffer from remote memory accesses and since the underneath global tree is F&F, it suffers from the same limitations explained above. Although we limit the asynchronous evaluate threads to four, the major factor in performance degradation are remote memory accesses to read the PTFO.

3) Varying Future Objects: Figure 5 shows the performance impact of varying PTFO size. We limit the number of future objects to 100, 500, and 1000, shown as F3-K(100), F3-K(500), and F3-K(1000) in Figure 5. We limit the number of `eval()` threads to only four and compared them with the performance of F&F with four threads only. We can observe in Figure 5 that the scalability of our proposed system is limited by the number of future objects. Meanwhile, the application threads (producers) spend most of their time waiting for the `eval()` threads (consumers) to checkpoint the data from PTFO to the global B^+-tree. Furthermore, we observe that the F3-K(No Limit), where we do not limit the number of future objects, has the best overall performance. F3-K(1000) shows equivalent performance to F3-K(No Limit) for 40 threads because the number of PTFOs is less than the threshold of 1000 future objects, and so the producers are not blocked during the entire execution.

In summary, we conclude that the workload patterns play an important role. If the workload has a constant bursty pattern (as in our experiments), then the performance of the producer threads is limited by the performance of the consumer threads. On the contrary, in a realistic workload where the bursty pattern is not constant, the impact on the performance of producer threads will not be affected by the consumer threads at a notable rate.

4) Read Performance: We perform read experiments to show the overhead of PTFO in Figure 6. We compare F&F with our proposed F^3-tree, which includes in-memory HT and F^3-tree without in-memory HT. For this experiment, we place 20% key-value pairs at the PTFO while 80% key-value pairs are placed at the global F&F tree. We observed that F^3-tree-No Hash has the worst read performance. This is because every read operation has to go through the PTFO first and if the key is not found, it then searches the global tree, whereas F^3-tree with HT shows equivalent performance to F&F with negligible overhead to check the HT first and then looking for the key in the corresponding PTFOs.

5) Realistic Workload Analysis: We simulate a realistic workload scenario where an application performs a mixed workload of read and write operations. For this experiment, we used random key distribution and the key-based operation mode of the F^3-tree. During this experiment, each thread alternates between four insert queries, 16 search queries, and one delete query, as performed in F&F. Figure 7 shows that the F^3-tree outperforms F&F and gains about 2.6x speedup when the number of threads increases. This is due to high

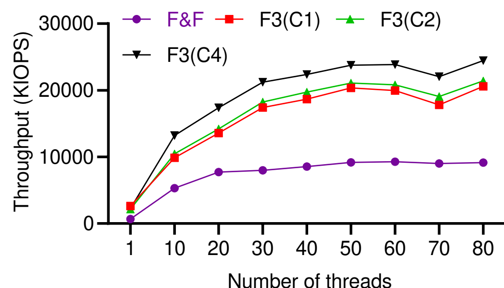


Fig. 7. Realistic workload analysis (2M Search/1M Insert/12.5K Delete). C_i shows the number of evaluate threads.

performing insert operations of the F^3-tree and the in-memory HT for search operations. Supporting range queries for the F^3-tree is challenging because range queries fetch multiple key-value pairs. The range of keys in the F^3-tree can overlap between PTFO and the global B$^+$-tree. Within PTFO, range query performance degrades even after using the in-memory HT because it does not support range queries. We plan to invest in range queries and NUMA-awareness in our future work.

V. Related Work

Recent studies on B+-tree indexing data structures can be classified into two types, i.e., hybrid (DRAM-PM) B+-trees [7], [10], [20] and PM-only B+-trees [6], [8], [21]. For PM-only, data is entirely in PM, ensuring the possibility of nearly instant recovery. For hybrid indexes, DRAM is used for auxiliary data that is rebuilt on recovery. DRAM has lower latency than PM, and this scheme usually results in improved performance at the cost of longer recovery time. But most of these studies have either violated the basic design of the B+-tree for performance improvement, such as allowing unsorted entries within the B+-tree nodes [21], or lack concurrency support [6]. F&F is the state-of-the-art persistent B+-tree variant that maintains the basic properties of B+-tree and also supports concurrent operations. However, none of the existing PM-based B+-tree write operations scale on manycore machines with hundreds of threads. In this work, we proposed F^3-tree, a highly concurrent persistent B$^+$-tree for DCPM-based manycore machines. We adopted future-based data structures from asynchronous computation [14] over F&F and achieve higher performance on manycore machines. Note that futures are not yet adopted for indexing data structures, and this is the first work that has adopted future-based data structures for B+-tree indexing data structures.

VI. Conclusion

In this paper, we present F^3-tree, a highly concurrent persistent B$^+$-tree for DCPM-based manycore machines. F^3-tree achieves scalability and high write concurrency by adopting per-thread local future objects. The per-thread future objects are later checkpointed to the global B$^+$-tree in an asynchronous manner based on tunable threshold, i.e., time and size-based. The search queries are optimized by employing a volatile in-memory hash table. We evaluate F^3-tree on a manycore Linux machine with emulated DCPM. The results show that F^3-tree achieves high scalability (3.4x on average) compared to F&F.

Acknowledgments

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035, Research on High Performance and Scalable Manycore Operating System).

References

[1] A. Papadopoulos and D. Katsaros, “A-tree: Distributed indexing of multidimensional data for cloud computing environments,” in *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pp. 407–414, 2011.

[2] A. Khan, H. Sim, S. S. Vazhkudai, A. R. Butt, and Y. Kim, “An analysis of system balance and architectural trends based on top500 supercomputers,” HPC Asia 2021, p. 11–22, 2021.

[3] B. Jeong, A. Khan, and S. Park, “Async-lcam: a lock contention aware messenger for ceph distributed storage system,” *Cluster Computing*, vol. 22, pp. 373–384, 2018.

[4] A. Khan, H. Sim, S. S. Vazhkudai, J. Ma, M.-H. Oh, and Y. Kim, “Persistent memory object storage and indexing for scientific computing,” in *Proceedings of the 2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pp. 1–9, 2020.

[5] A. Khan, C.-G. Lee, P. Hamandawana, S. Park, and Y. Kim, “A robust fault-tolerant and scalable cluster-wide deduplication for shared-nothing storage systems,” in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 87–93, 2018.

[6] S. Chen and Q. Jin, “Persistent b+-trees in non-volatile main memory,” *Proc. VLDB Endow.*, vol. 8, p. 786–797, Feb. 2015.

[7] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, “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 ’16, p. 371–386, 2016.

[8] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, “Endurable transient inconsistency in byte-addressable persistent b+-tree,” in *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST’18, p. 187–200, 2018.

[9] A. Khan, H. Sim, S. S. Vazhkudai, and Y. Kim, “Mosiqs: Persistent memory object storage with metadata indexing and querying for scientific computing,” *IEEE Access*, vol. 9, pp. 85217–85231, 2021.

[10] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, “Nv-tree: Reducing consistency cost for nvm-based single level systems,” in *the Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)*, pp. 167–181, Feb. 2015.

[11] M. L. Scott, *Shared-Memory Synchronization.* Morgan and Claypool Publishers, 2013.

[12] D. Dice, V. J. Marathe, and N. Shavit, “Flat-combining numa locks,” SPAA ’11, p. 65–74, Association for Computing Machinery, 2011.

[13] M. Chabbi, M. Fagan, and J. Mellor-Crummey, “High performance locks for multi-level numa systems,” vol. 50, p. 215–226, Jan. 2015.

[14] A. Kogan and M. Herlihy, “The future(s) of shared data structures,” in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC ’14, p. 30–39, 2014.

[15] P. L. Lehman and S. B. Yao, “Efficient locking for concurrent operations on b-trees,” *ACM Trans. Database Syst.*, vol. 6, p. 650–670, Dec. 1981.

[16] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[17] P. Ramalhete and A. Correia, “Brief announcement: Hazard eras - non-blocking memory reclamation,” in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’17, p. 367–369, Association for Computing Machinery, 2017.

[18] N. Cohen and E. Petrank, “Efficient memory management for lock-free data structures with optimistic access,” in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’15, p. 254–263, Association for Computing Machinery, 2015.

[19] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, “An empirical guide to the behavior and use of scalable persistent memory,” in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pp. 169–182, Feb. 2020.

[20] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen, “Dptree: Differential indexing for persistent memory,” *Proc. VLDB Endow.*, vol. 13, p. 421–434, Dec. 2019.

[21] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, “Consistent and durable data structures for non-volatile byte-addressable memory,” in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST’11, p. 5, 2011.