Future-based Concurrent Tree Index Data Structures For Manycore Machines

서강대학교 ▌Safdar Jamil · Abdul Salam · Youngjae Kim*

1. INTRODUCTION

Cloud vendors are generating enormous data and facing catastrophic challenges due to huge amount of data that needs to handle by several applications [1]-[3]. The emergence of manycore machine and Intel DC Persistent Memory (DCPM) [4] aimed to increase the concurrency of applications and provide higher performance. For manycore machines, adopting DCPM results in persistency at the memory level but degrades the application performance. Applications managing user-generated data by cloud services include databases and filesystems [5]. These applications heavily rely on indexed data structures for high performance to access the data.

Several index data structures have been proposed for emerging memory technologies such as B+-tree, R-tree etc. [6]-[9]. These data structures are tree-based and have been commonly used in databases and file systems. The B+-tree and R-tree are among popular index data structures. Few recent studies explored B+-trees and R-trees for DCPM [8], [10]. Fast&Fair (F&F) [8] is a B+-tree variant and FBR-tree [10] is a state-of-the-art variant of R-tree on DCPM. However, adoption of DCPM-based data structures on manycore machines lead to scalability limitations. The write operations in F&F and FBR-tree acquire locks to ensure mutual exclusion, that becomes a point of contention where multiple threads attempt to access tree-based data structures such as B+-tree and R-tree. Node splitting and merging is common operations in tree-based data structures to maintain the balance of the tree, commonly known as Structural modification operations (SMO). When the chain of SMOs are triggered from leaf to root node they additionally cause increase in thread contentions. These chains of SMOs are required to hold per-node lock from the leaf to the root node where multiple threads try to hold lock for the node on tree-based data structures.

MCS [11], FC-MCS [12], and HMCS [13] have been used to optimize lock techniques for such data structure. But these techniques are unable to capture and solve the inherited concurrent limitation of tree-based data structures such as B+-tree and R-tree. Future Objects (FO) [14] are concurrent and substitute for such lock optimization techniques. FOs have been proposed to improve the performance of shared data structures. FOs data objects promise to deliver the results of an operation once the results become available. Futures are applied on each thread level where each thread assigns its own future objects. Each operation that is represented by FOs are than applied to the shared data structure when their evaluate method is called.

Adopting futures for indexing data structures such as B+-trees and R-tree can improve the operation performance, but it comes with its own challenges. For instance, to integrate the future objects with tree-based index data structure can cause consistency issues. Whereby, evaluate method, that is responsible to

* 정회원

† This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1A2C2014386) and by the Institute of Information and Communications Technology Planning and Evaluation (IITP), Korea government (MSIT) (Development of low-latency storage module for I/O intensive edge data processing) under grant No.~2020-0-00104.

checkpoint the data from future objects to shared data structures, must incorporate in a crash-resilient manner; otherwise, data might be lost eventually will leave the index data structure in an inconsistent state. FOs that are thread local will also severely degrade the lookup operation performance of shared index data structure. The read operations have to traverse the additional layer overhead of future objects to look for key. Furthermore, positing future objects in DCPM also requires durability guarantee to help recovery of the index data structure after a crash. Without durability guarantee, future objects might lay in an inconsistent state, such as missing data or pointers between future objects.

To address aforementioned problems, we applied futures on two different state-of-the-art index data structures i.e., single dimension B+-tree and multi-dimensional R-tree. Our design focused to improve the concurrency among the threads and composed of three main components i) Thread-Local Future Objects (TLFOs), ii) a shared global index data structure and iii) In-memory hash table.

We applied well known producer-consumer approach in our design where threads are being divided into two groups. Each thread group is responsible to handle the designated task. Application threads, act as producer, are allowed to push the data to the TLFOs. Meanwhile, designated asynchronous thread, act as consumers, are responsible to perform operation where each consumer thread will update the TLFOs to the shared global index data structure. To adopt the futures on DCPM, we converted DRAM-based FOs to DCPM-based future objects and durable linearizability is used to guarantee the correctness conditions after a crash. We compensate the look operation performance by employing an in-memory hash table to avoid key search in TLFOs. We evaluated the proposed design against baseline F&F and FBR-tree with different workloads. The experimental results confirm that futures can be used to improve the concurrency of the index data structure.

2. BACKGROUND AND MOTIVATION

2.1 BACKGROUND

There have recent studies that proposed DCPM-based index data structures that provides lock-free reads and tolerate the inconsistencies. Fast and Fair (F&F) is B+-tree based DCPM index data structure transform the B+-tree into consistent state by completely avoiding the logging overhead. Lookup operations as well detect and indulge the inconsistent states such as duplicated elements in sorted list. However, the write operation acquire lock to avoid the mutual exclusion among threads. F&F offered two algorithms, Failure Atomic Shift (Fast) and Failure Atomic In-place Re-balance (Fair). Fast is responsible to insert the new key into a node of the B+-tree that is also managed to perform atomic shift operations to keep the order of the keys in sorted manner. The Fair algorithm uses sibling pointer and does not require to use of logging in the B+-tree nodes.

Similarly, R-tree is one of the popular spatial index data structure used to store multidimensional data. R-tree is multidimensional data structure and is different from single-dimensional data structures such as B+-tree. This is because R-tree does not store the sorted array of key and values in tree nodes, instead it stores a set of minimum bounding rectangles (MBR). Each MBR includes all the MBRs of the respective sub-tree and leaf nodes of R-tree stores the MBR pointing to actual spatial data. FBR-tree is a DCPM-based R-tree. FBR-tree introduces a variable sized bitmap metadata update operation. FBR-tree also provides lock-free read operations in parallel to insert operations.

Additionally, maintaining consistent view of the tree-based indexing data structure during the SMOs are challenging tasks this is because tree-based index data structure requires additional logging. Logging cause additional overhead because it requires to maintain and duplicates the memory pages which increase the write traffic. This also block the concurrent access to shared data structures. Such tree-based data structures in highly concurrent write scenarios face scalability limitations that is when concurrent writes access the shared data structure on manycore machines where tens

and hundreds of application threads are performing write operation.

2.2 MOTIVATION

In this section, we explain the write operations in tree-based index data structure. Figure 1 shows the concurrent write operation in F&F, where two threads, T1 and T2 are performing insert operation. Thread T1 and Thread T2 first look up the specific node where a key insertion is needed to perform, as shown in Figure 1, step①. Note that both threads T1 and T2 selected the same node of the tree to perform insert operation. However, as F&F uses MUTEX locks to avoid the mutual exclusion only one thread will be able to acquires the lock on the node and start the insert operation. As shown in Figure 1, thread T1 first acquire the lock and started the insert operation, step ②. Now, thread T1 checks the capacity of the node where new data is needed to insert. It was observed that the node (N1 in this case) does not have capacity to accommodate new data, it triggers the SMO. During SMO, Thread T1 allocates a new node, shift half of the existing node entries from N1 to the newly created node N5 and then finally insert the data into the corresponding node, steps ③ and ④. At final stage step ⑤, thread T1 acquires the lock at parent node PN and updates the links. Meanwhile, now thread T2 can acquire the lock at node N1 and notices that the N1 has been split. Now the new data that thread T2 wanted to insert is required to insert in the new candidate node N5. So, at step ⑥, thread T2 releases the lock at node N1 and acquires the lock at new node N5 and proceeds with rest of the insert operation.

This example shows that tree-based index data structures get blocked with synchronization along with chain of SMOs dramatically limits its scalability on manycore machines. It effects the concurrency of the application where hundreds and thousands of application threads are performing insert operation. To address this problem of tree-based index data structures we proposed future-based index data structures. Where our goal is to focus on designing a concurrent scalable index data structure for DCPM-based manycore machines.

3. DESIGN AND IMPLEMENTATION

3.1 SYSTEM OVERVIEW

Figure 2 shows the overall design of our proposed future-based system for index data structures. Our approach is inspired by well-known asynchronous computation design principle, producer and consumer model. In our design we have three main components, Thread Local Future Objects (TLFOs), Shared Index data structure, and in-memory hash table. TLFOs in our system behaves as buffer for each thread.

Application threads, act as producer, are only responsible to perform the write operation on this buffer i.e. TLFOs. For TLFOs, we used the doubly linked list. In the meantime, a dedicated asynchronous thread pool, act as consumer, are responsible to checkpoint the data objects in TLFOs to global index

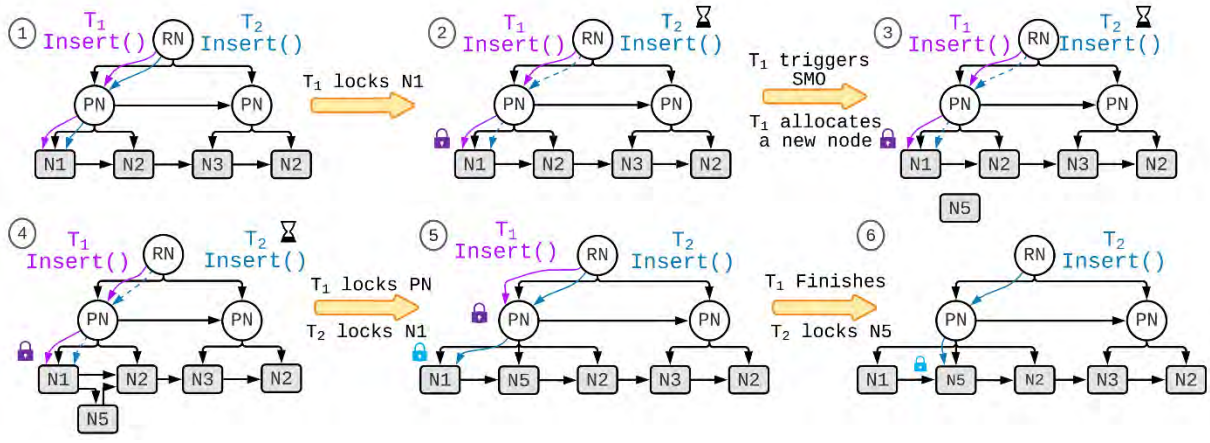

그림 1 A multi-thread insertion use case to demonstrate the scalability limitation in tree-based index data structure [15].

data structure through the evaluate function. To minimize the lookup performance overhead due to additional layer of TLFOs, for data objects residing in TLFO, we used an in-memory hash table (HT). A data object is inserted into the hash-table after it is durably written to the TLFO.

The doubly linked-list of TLFO is shown in Figure 2 where each node of the doubly linked-list is comprised of Future Objects (FO), next and previous pointer to the node, and a dummy node which stores the head and tail pointers. We adopted the design of doubly linked-list due to twofold: First, with doubly linked-list the contention between application threads and asynchronous evaluate thread is avoided by only allowing the application threads to perform update operation through the head pointer. Meanwhile, the asynchronous evaluate threads only checkpoint the FOs through tail pointer. To provide crash consistency, we rely on durable linearizability [16].

3.2 OPERATION FLOW

Figure 2 shows the insert, search, and delete operation flow of our proposed system design. The green arrows in Figure 2 along with dedicated asynchronous evaluate threads show the insert operation flow. During the insert operation, the application thread will first insert the data object to the thread-local linked-list as shown in step①. Once the object is written to the TLFO, an entry will be made in the in-memory hash table for lookup as shown in step②. On the other hand, the

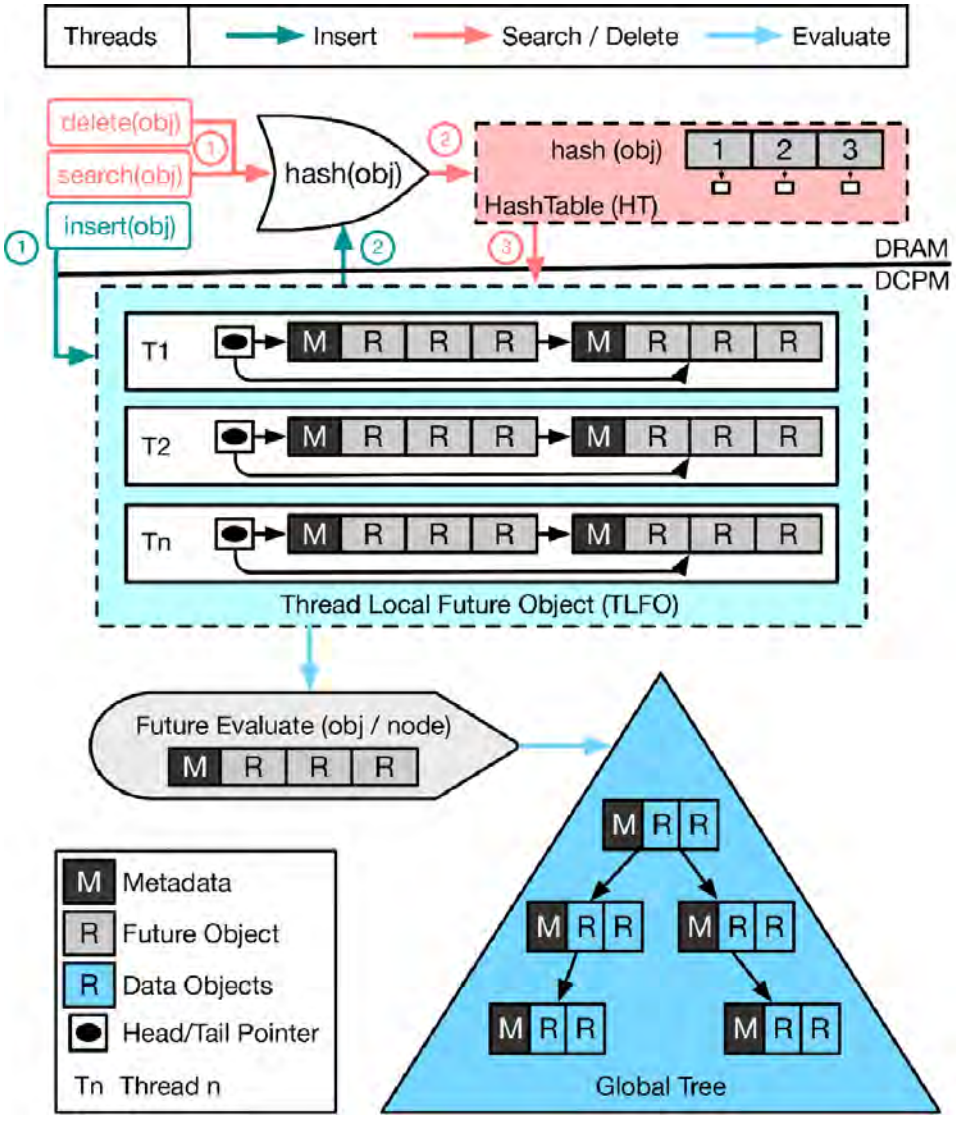

그림 2 Design overview of Future-based index data structure.

dedicated asynchronous evaluate threads are responsible to checkpoint the data objects from TLFOs to global index data structure, steps ③ of Figure 2. Each evaluate thread is responsible for specific TLFOs and only checkpoint the data from assigned doubly linked lists. The evaluation of TLFOs can be done based on two options; data in TLFOs can be written to global index data structure based on the amount of time or size of the TLFOs.

Our design provides hierarchical data object traversal for search and delete operations. Delete operation is comprised of lookup and update operation so delete and search operation follows the same path. To look for a data object in TLFO, an application thread will first start with traversing the in-memory hash table as shown by step ① in Figure 2. If a data object is found in in-memory hash table, then the corresponding TLFO will be traversed directly. If hash table is not adopted, then each application lookup operation must linearly traverse each TLFO based doubly linked list which results in high performance degradation as shown in the section IV. Once the data object is found, the application thread returns the data object to the client as presented by step ② . If a data object is not found at in-memory hash table, then application thread moves to traverse the global index data structure and follows the default index data structure's read path.

An important factor that has effects on the performance on Future-based system is the size of TLFO. If there is no limitation to the size of TLFOs, then application threads do not go into block state due to reaching the size threshold of TLFO. However, with limited TLFO size, the performance is limited by the global index structure's internal operations and lock contention. Additionally, the number of asynchronous evaluate threads play a vital role in the performance of future-based data structure. If the number of asynchronous evaluate threads is large than it will result in internal lock contention of the global data structure while small number of evaluate threads will cause the application threads to get blocked. So, it is important to identify the suitable number of asynchronous evaluate threads and the size of TLFOs.

To support the crash consistency and recovery mechanism, our proposed design relies on durable linearizability where each operation take affect durably in between its invocation and response. Linearizability is a common correctness condition employed in lock-free data structures. As in our design TLFOs do not acquire any lock and work independently, we adopted linearizability for correctness condition. If a system crash happens than in our design the asynchronous evaluate threads will be able to access the TLFOs that last durably written and start checkpointing them to the global index structure. However, for application threads, new TLFOs and in-memory hash table will be allocated, and application threads can start their operation right away.

3.3 DURABLE LINEARIZABILITY (DL)

A concurrent data structure is linearizable if each operation that is performed take effect between its

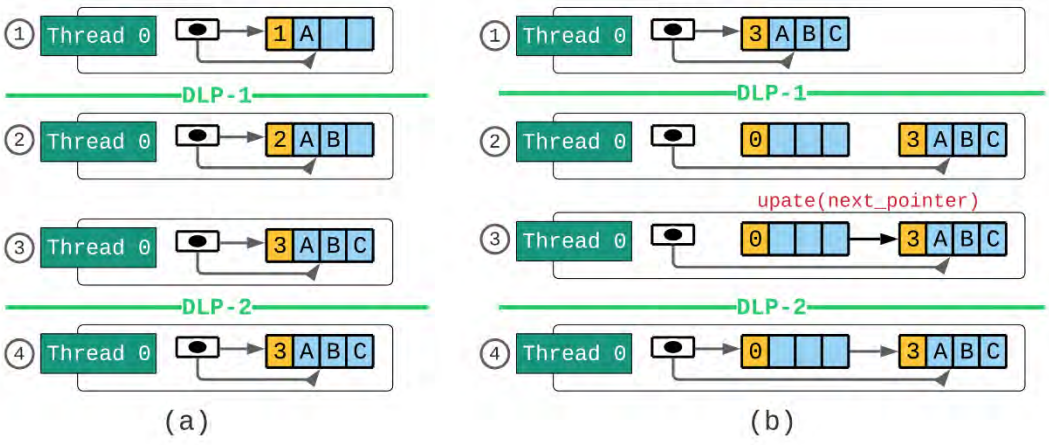

그림 3 Durable Linearizability examples. DLP represents the DL point.

invocation and response [17]. With DCPM, guaranteeing durability is an additional requirement that a data structure need to provide to ensure the persistency and crash resiliency. A durably linearizable concurrent data structure satisfies the properties of linearization. Furthermore, a durable linearizable data structure should be able to retain a consistent state in case of the full system crash or partial system failure. The state of the concurrent data structure must reflect a consistent sub-history, a sequence of method invocation and their responses.

We call the point where a concurrent data structure becomes durable as a durability point in the execution history E, i.e., its effects are visible to all the corresponding application threads and the data is persistent. After a durability point is acquired, if we execute a recovery mechanism, it will result in a consistent sate of the data structure. For our proposed design, we achieve durability point for TLFOs as shown in Figure 3. We do not consider the same of the global data structure as the global data structure can follow any consistency guaranteeing mechanism including lock-based synchronization.

Figure 3(a) shows an example where we achieve the durability point within a single future object of TLFO by atomically updating the data objects. Steps ① to ④ in Figure 3(a) show the write operation within an FO. The lines labeled DLP represent the durable linearizability points achieved by the insert operation by calling FLUSH+FENCE instructions. In Figure 3(a), there are two durably linearizability points 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 doubly linked list. Steps ② to ④ show the allocation of a new future object in the thread-local doubly linked list. In this scenario, the DLP is achieved once the next pointer of the newly allocated future object 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 future object so that if a crash happens after the DLP, we can 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 [18] and the optimistic access scheme [19].

3.4 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 TLFO or in the global B+-tree. Also, the in-memory hash table is placed in DRAM and not DCPM, so we do not consider its space overhead for DCPM. Though, for DRAM the hash table space overhead is O (M), where M is the number of keys stored in the TLFO 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 TLFO and second where a thread only looks for the key in the global B+-tree. In addition, the read operation must go through the in-memory hash table. Now, if a thread is looking for a key it must first search the hash of the key in the hash table, which has constant time complexity O (1). If the key is found in the hash table, then the thread will traverse the TLFO 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 hash table, 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.

4. EVALUATION

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 emulate the latency of Intel DCPM as presented in [20]. We implemented the proposed

solution for two state-of-the-art index data structure solutions i.e., B+-tree and R-tree.

Our solution for B+-tree, we call it F3-tree, is implemented on top of F&F [8]. We used a synthetic benchmark with one million 8-byte key-value pairs with sequential key distribution. We bind the threads to the first CPU node using the numactl command and then move to other nodes. We compared the proposed approach, F3-tree, to two different variants, i.e., key-based (F3-K) and node-based (F3-N). In key-based approaches asynchronous evaluate threads perform checkpointing based on each key-value pair stored at future object in TLFO. Meanwhile, in the node-based approach asynchronous evaluate threads checkpoint whole future object to global index data structure.

For R-tree solution, we call it MPR-tree, is implemented on top of FBR-tree [10]. We implemented our proposed MPR-tree using a Persistent Memory Development Kit (PMDK) [21]. To evaluate MPR-tree, we allocated a single memory pool for index and call pmemobj_alloc() for each tree node inside the pool. For evaluation, we used a time series multidimensional taxi service trajectory workload that has millions of polylines with a total of nine attributes[1]. A polyline contains a list of GPS coordinates (i.e., WGS84 format). To check the concurrency performance, we performed experiments by increasing the number of concurrent threads for insert operations with a 500K queries 3D workload into MPR-tree.

4.1 FUTURE-BASED B+-TREE

4.1.1 INSERT PERFORMANCE

Figure 4(a) shows the performance for F3-tree in comparison to state-of-the-art F&F. We can 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 TLFO 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 on average 2.4x due to its TLFO 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 saturates with threads crossing the single CPU node boundary due to high remote memory accesses. Although the F3-tree writes to TLFO still the asynchronous threads read the TLFO 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.

4.1.2 READ PERFORMANCE

We perform read experiments to show the overhead of TLFO in Figure 5(a). We compare F&F with our proposed F3-tree, which includes in-memory HT. For this experiment, we place 20% key-value pairs at the TLFO while 80% key-value pairs are placed at the global F&F tree. It can be observed that F3-tree shows equivalent performance to F&F with negligible overhead to check the HT first and then looking for the key in the corresponding TLFOs.

4.2 FUTURE BASED R-TREES

4.2.1 INSERT PERFORMANCE

Figure 4(b) depicts the results of our concurrency analysis of MPR-tree in comparison to state-of-the-art FBR-tree. We varied the asynchronous evaluate threads in this experiment to show the impact on performance. The e2 and e4 present the number of asynchronous evaluate threads in Figure 4(b). We can clearly observe that MPR-tree outperforms FBR-tree ranging from 1.7x to 2.1x by varying the number of threads from 1 to 80 on Log10 scale. It can also be seen that MPR-tree's performance saturates as the number of threads increases, i.e., after 10 threads. The reason is twofold. First, since our experiments were run on a NUMA machine, threads tend to perform memory allocations to their local CPU nodes, which leads to remote memory accesses. Second, with an increasing number of

1) The data set can be found here: https://archive.ics.uci.edu/ml/index.ph

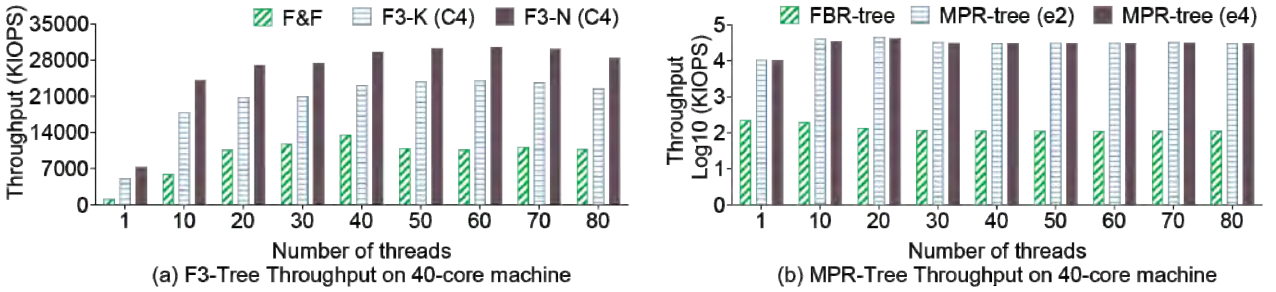

그림 4 Scalability analysis of F3−tree and MPR−tree on manycore machines.

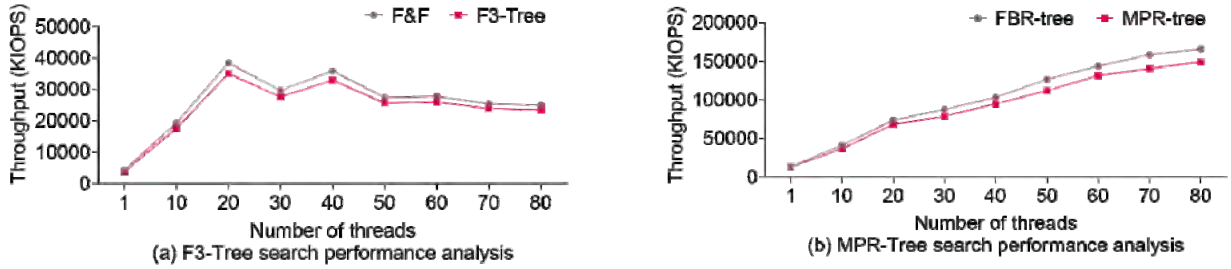

그림 5 Search performance analysis of F3−tree and MPR−tree on manycore machine.

threads, the pressure on evaluate threads creates a contention and thus leads to limited performance. Notably, all different numbers of evaluate threads showed a scalable trend for threads within a single CPU node. Due to the overhead of remote memory accesses, performance suffers when threads cross the CPU-node barrier. The dominant factor in performance degradation is remote memory accesses with the asynchronous evaluate threads because evaluate threads frequently read the future objects from remote memory and write them to global FBR-tree-FG.

4.2.2 SEARCH PERFORMANCE

We performed experiments for search queries where 1M objects were inserted into the global FBR-tree and then 500K objects were searched to determine the overhead of TLFOs, as depicted in Figure 5(b). We compared FBR-tree with our proposed MPR-tree, which includes the in-memory hash table. For this experiment, we placed 20% objects in TLFOs while 80% objects were placed in the global tree. MPR-tree with its in-memory hash table showed very negligible performance overhead as compared to FBR-tree. This is because MPR-tree with hash first must check the hash table and then look for the object in the corresponding TLFOs.

5. CONCLUSION

In this article, we introduce future for index data structure, where a highly concurrent persistent B+-tree and R-tree was implemented for DCPM-based manycore machines. Our future-based solutions achieve scalability and high write concurrency by adopting Thread Local Future Objects (TLFOs). The per-thread future objects are later check-pointed to the global tree-based index data structures in an asynchronous manner. The search queries are optimized by employing a volatile in-memory hash table. We evaluate our proposed design F3-tree based on B+-tree and FBR-tree based on R-tree on a manycore Linux machine with emulated DCPM. The results show that future based solutions achieve high scalability compared to baseline F&F and FBR-tree.

References

[1] A. Papdoupoulos and D. Katsaros, “A-tree: Distributed indexing of multidimensional data for cloud computing environments,” in Proceedings of the 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] "3D xpoint™: A breakthrough in non-volatile memory technology..", https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html

[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 Proceedings of the 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, (New York, NY, USA), p. 371-386,Association for Computing Machinery, 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] S. Cho, W. Kim, S. Oh, C. Kim, K. Koh, and B. Nam, "Failure-atomic byte-addressable r-tree for persistent memory," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 3, pp. 601-614, 2021.

[11] M. L. Scott, Shared-Memory Synchronization. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2013.

[12] D. Dice, V. J. Marathe, and N. Shavit, "Flat-combining NUMA locks," in Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, 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," in Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, p. 215-226, Association for ComputingMachinery, 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] S. Jamil, A. Khan, B. Burastaller, and Y. Kim, "Towards scalable manycore-aware persistent b+- trees for efficient indexing in cloud environments," in 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), pp. 44-49, 2021.

[16] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank, "A persistent lock-free queue for non-volatile memory," in Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '18, p. 28-40, 2018.

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

[18] 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.

[19] 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.

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

[21] "Defining the future of in-memory database computing." https://pmem.io/vmem/libvmmalloc/

약 력

Safdar Jamil

2016 BE in Computer Systems and Engineering, Mehran University of Engineering and Technology, Pakistan
2018-Present PhD Course, Sogang University
Research Interest : Database systems, data structures and algorithms, parallel processing, and distributed systems
Email : safdar@sogang.ac.kr

Abdul Salam

2016 BS in Computer Systems Engineering, MUET Pakistan
2017~2018 Jr. Software Engineer, Osmani & Company pvt. ltd.
2018~2020 Program & Training Officer, SEF Pakistan
2022 MS Computer Science and Engineering, Sogang University
2022 Present Memory Optimization Engineer, Dubai.
Research Interests : Data structures and algorithms, parallel processing, and database systems
Email : abdulsalam.arain@gmail.com

Youngjae Kim

2001 BS in Computer Science, Sogang University
2003 MS in Computer Science, KAIST
2009 PhD in Computer Science and Engineering, Pennsylvania State University, USA
2003~2004 ETRI (Researcher)
2009~2015 Oak Ridge National Laboratory, USA (R&D Staff)
2015~2016 Ajou University (Assistant Professor)
2016-Present Sogang University (Associate Professor)
Research Interest: File and storage system, database system, concurrency, parallel processing, distributed system, computer security
Email: youkim@sogang.ac.kr