Received 2 October 2022, accepted 17 October 2022, date of publication 21 October 2022, date of current version 7 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3216410

RESEARCH ARTICLE

Future-Based Persistent Spatial Data Structure for NVM-Based Manycore Machines

ABDUL SALAM[1], SAFDAR JAMIL[1], SUNGWON JUNG[1], SUNG-SOON PARK[2,3], AND YOUNGJAE KIM[1], (Member, IEEE)

[1]Department of Computer Science and Engineering, Sogang University, Seoul 04107, South Korea
[2]Gluesys Company Ltd., Anyang 951074, South Korea
[3]Department of Computer Engineering, Anyang University, Anyan-si 14028, South Korea

Corresponding author: Youngjae Kim (youkim@sogang.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2021R1A2C2014386) and by the Institute of Information Communications Technology Planning Evaluation (IITP) grants funded by the Korea government (MSIT) (No. 2020-0-00104, Development of Low-latency Storage Module for I/O Intensive Edge Data Processing) (No. 2021-0-00136, Development of Big Blockchain Data Highly Scalable Distributed Storage Technology for Increased Applications in Various Industries) (No. 2021-0-00180, Development of High-speed Analysis of Distributed Large-scale Data for Wide Usability of Various Industries).

ABSTRACT R-trees have been popular for their support of multidimensional data and high-performing queries. FBR-tree is the state-of-the-art concurrent variant of the R-tree for Intel DC Persistent Memory (DCPM). However, its adoption on manycore servers is impeded by concurrency limitations due to lengthy, lock-based synchronization, including structure modification operations, split and merge. Additionally, emerging DCPM-based machines are equipped with multiple CPU sockets, forming a non-uniform memory access (NUMA) architecture. FBR-tree's lack of NUMA-awareness induces further performance overhead from remote memory accesses. In this paper, we propose MPR-tree, a concurrent, NUMA-aware and persistent future-based R-tree for DCPM servers. MPR-tree focuses on insert operations due to their laborious nature. MPR-tree relies on per-thread local future objects and a global R-tree. To introduce NUMA-awareness and minimize remote memory accesses, MPR-tree adopts per-socket dedicated asynchronous evaluate threads to checkpoint future objects to the global R-tree. MPR-tree employs an in-memory hash table to mitigate the read overhead of key searches over the future objects. We implemented MPR-tree atop FBR-tree and evaluated its performance on a server with 40 physical cores for insert and lookup queries, and it showed that MPR-tree outperforms FBR-tree on average by 2× on log10 scale.

INDEX TERMS Futures, index data structures, non-volatile memory, manycore machines.

I. INTRODUCTION

The emergence of manycore machines with Intel DC Persistent Memory (DCPM) [1] aim to provide high performance and concurrency/scalability with persistence guarantees. DCPM offers byte addressability, high density, persistency, and close to DRAM performance. DCPM has been introduced with manycore machines with multiple CPU nodes where each CPU node can comprise tens of cores and support a high degree of parallelism [2]. These multi-CPU nodes are known as non-uniform memory access (NUMA) machines due to irregular memory access latency between CPU cores [3], [4] .[1] These DCPM-based manycore machines have been adopted by several cloud vendors [5], [6] and are also being included in high-performance computing facilities [7]. With the DCPM-based manycore machines, applications such as databases and filesystems can operate and persist data directly on the memory bus due to the aforementioned characteristics of DCPM.

DCPM-based manycore machines have become more popular, and applications such as databases and filesystems are expected to have high performance with an increased number of resources i.e., CPUs. However, several studies have

The associate editor coordinating the review of this manuscript and approving it for publication was Alba Amato.

[1]Hereafter we will refer to NUMA-based manycore machines as simply manycore machines.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

shown that even with an increased number of CPUs, such applications do not improve performance due to a high degree of concurrency and parallelism [8], [9]. Furthermore, this limited performance occurs due to a lack of support for the high degree of parallelism in data structures adopted in these applications, which rely heavily on index data structures such as B/B+-trees, hash tables, radix trees, and R-tree for fast data retrieval. Several indexing data structures have been proposed to exploit the performance characteristics of DCPM [10], [11], [12], [13], [14]. These index data structures can be classified as single-dimensional and multidimensional data structures. Single-dimensional index data structures, such as B/B+-trees, radix tree, and hash tables, have been exploited extensively for DCPM-based manycore machines. However, multidimensional index data structures such as R-tree have only been studied with single CPU node machines and have not been exploited for DCPM-based manycore machines [14].

FBR-tree is a state-of-the-art DCPM variant of R-tree. However, its adoption on manycore machines results in an application concurrency constraint. This is because FBR-tree uses legacy exclusive locking, which results in lock contention. The lock contention mainly increases for two reasons. First, the exclusive lock creates a point of contention when multiple threads try to perform insert/update operation on the same node. Second, the chain of structural modification operations (SMOs), such as split and merge, increases this contention as each thread is only able to acquire the lock when the previous thread completes its operation. This lock contention limits the performance of FBR-tree on DCPM-based manycore machines and does not fully exploit the resources and high degree of parallelism of manycore machines.

There have been several solutions to optimize R-tree on DRAM-based manycore machines [15], [16]. These techniques include shadowing and log-based approaches. However, when applied to DCPM, these approaches incur substantial performance overhead due to extra writes and cacheline flushes [3], [12]. Furthermore, traditional lock optimization techniques, such as MCS [17], FC-MCS [18], and HMCS [19], cannot overcome the inherit limitations of FBR- Tree due to SMOs and mutual exclusion. An alternate approach to such lock-based optimizations is to adopt scalable- friendly futures [20].

Futures have been proposed to increase the performance of concurrent shared data structures. Futures are the promise objects used to deliver the results of an asynchronous computation when a client/application requests it. Futures can be allocated as thread-local objects, and these objects perform the assigned task when the evaluation method is called. These thread-local future objects (TLFOs) can be applied to the shared data structure in various ways. For instance, an evaluation method can process the pending FOs in a batch or one by one. Applying TLFOs to R-tree on DCPM will help achieve concurrency for R-tree operations, as each application thread will perform operations independently. However, applying future objects on such complex data structures comes with its own challenges, such as maintaining a consistent view between the FOs and persistent R- Tree, and avoiding read performance degradation.

Furthermore, index data structures also suffer from the NUMA-effect, a trade-off between high memory bandwidth and irregular memory access latency [3], [4]. FBR-tree and other DCPM-based data structures are particularly vulnerable to performance loss from the NUMA-effect because they are not designed for NUMA machines. The NUMA-effect becomes significant since these data structures rely on a pair of FLUSH+FENCE instructions to achieve persistency by writing data from the cache to the remote DCPM node, which results in high remote memory accesses and hinders the scalability of applications. This is because, compared with local DCPM write, the peak bandwidth of remote ones is decreased to 59% over DCPM-based manycore machines.

To solve the concurrency limitations of FBR-tree, we proposed MPR-tree, a manycore-aware persistent R-tree for DCPM-based manycore machines. MPR-tree design is composed of three core components: (i) thread-local future objects (TLFOs), (ii) an in-memory hash table to optimize read operations, and (iii) a shared R-tree with fine-grained locking. To the best of our knowledge, this is the first work to address the concurrency limitation of the persistent R-tree on DCPM-based manycore machines.

The key contributions of this work are as follows:

- MPR-tree first reduces the lock contention by replacing the FBR-tree's lock with the adoption of the lock-release-lock concept, which we call fine-grained locking, whereby in less concurrent write scenarios our proposed technique achieves relatively high performance.
- MPR-tree introduces future objects (FOs) for high-volume concurrent writes to R-tree where application threads are responsible for inserting the spatial objects into thread-local future objects (TLFOs) while designated asynchronous evaluate threads checkpoint the data into R-tree. To maintain a consistent view of TLFOs in DCPM, we rely on durable linearizability for correctness condition.
- To reduce the cross-CPU communication among application and evaluate threads, we introduced NUMA-aware TLFOs and evaluate threads to minimize remote memory accesses.
- We evaluated our proposed MPR-tree against the baseline FBR-tree and showed that MPR-tree outperforms FBR-tree by 2× with an increasing number of threads on log10 scale while achieving crash consistency and comparable read performance and reducing remote memory access by 1.3× with node-local memory allocations.

Roadmap: The rest of the paper is organized as follows. Section II describes the background and motivation. Section III explains the problem definition. Section IV presents the solution and design for scalable R-tree for

DCPM-based manycore machines. Section V presents evaluation and experiment results. Finally, we discuss related work in section VI and conclude the paper in section VII.

II. BACKGROUND AND MOTIVATION

In this section, we present the background of FBR-tree, followed by how existing index data structures address the concurrency limitation and NUMA-effect on manycore machines.

A. FBR-TREE: DCPM-BASED R-TREE

R-tree is one of the most popular spatial index data structures used in databases to store multidimensional data objects. R-tree is different from other single-dimensional data structures because it stores a set of minimum bounding rectangles (MBR) instead of sorted array of keys and value pointers at tree nodes. Each MBR contains all the MBRs of the corresponding sub-tree, whereby leaf nodes store the MBR pointing to the actual spatial object, i.e., data object. FBR-tree is a DCPM variant of R-tree. FBR-tree introduces a configurable size bitmap metadata update operation. Since the DCPM store instruction is bounded to 8-Byte, FBR-tree updates 8-byte metadata atomically, whereas in the case of metadata larger than 8 bytes, it carefully flushes the data to DCPM by relying on hardware transactional memory. For structural modification operations (SMOs), such as split and merge, FBR-tree introduces in-place rebalancing algorithms with metadata-only logging. FBR-tree also provides lock-free read operations in parallel to insert operations.

When the MBR of a new spatial object is inserted, FBR-tree performs a recursive search from the root to a leaf node to identify the candidate MBR of the leaf node using the least enlargement heuristics algorithm [21]. Once the candidate MBR is selected, two cases need to be taken into account. First, when the selected MBR does not overlap the new MBR, the existing MBR will be enlarged to accommodate the new MBR. After enlarging the MBR, FBR-tree calls FENCE+FLUSH to persist the enlarged MBR. Second, if the candidate MBR overlaps the new MBR, FBR-tree will update MBRs as needed on the way down to a leaf node, ensuring that all ancestor nodes get the updated MBR. Whenever it finds a leaf node, it will seek a vacant space by checking the bitmap in the leaf node and storing the object's spatial coordinates and call FENCE+FLUSH to persist the new object. Finally, it will increment the version number and update the bitmap to validate an MBR newly inserted into the FBR-tree.

If a leaf node is overflown, FBR-tree will call a split operation. First, it will allocate memory for a new sibling node and copy half of the content to the new node. Second, it will update the version number to 0, which tells other threads that this particular node is splitting and the content might not be updated in the parent node yet. In the next step, it will add the corresponding new node MBR to the parent node and update the bitmap of the parent node to reflect the changes. Suppose a system fails before the bitmap is updated. In that case, the written MBR is ignored and treated as an empty space when the system recovers. Therefore, there is no need for a recovery procedure.

B. CONCURRENCY LIMITATIONS OF FBR-TREE

FBR-tree adoption on DCPM-based manycore machines results in an application concurrency constraint. This is because, to avoid mutual inclusion during the write operation, FBR-tree uses legacy exclusive locking, which results in lock contention. To investigate the concurrency limitation of FBR-tree, we performed concurrency experiments with an increasing number of threads. Our evaluation platform consists of four CPU nodes with 10 cores per CPU. We used the synthetic data used by [14] and inserted 500K spatial objects into FBR-tree with 100% put (write) requests (details of the experiment setup are mentioned in Section V)

Figure 1 depicts the throughput and performance breakdown analysis of the FBR-tree. In Figure 1(a) FBR-tree shows adequate performance with a single thread. As we accumulate more threads, the overall throughput of the FBR-tree starts decreasing right from start and becomes stable after a certain number of threads. This leads us to further investigate what is precipitating this throughput performance drop. Therefore, we further analyzed the performance breakdown within the FBR-tree and identified that one potential cause could be lock contention among threads, because in FBR-tree the write request acquires locks on different parts of the R-tree. Figure 1(b) shows the result of the lock contention among threads. A thread has to wait to acquire the lock on a particular node. As depicted in Figure 1(b), with an increasing number of threads, the lock contention increases, which limits the performance of FBR-tree on manycore machines (Section III presents a working example of it).

C. OPPORTUNITIES TO ADOPT FUTURES IN INDEX DATA STRUCTURES

With the emergence of manycore machines, the use of manycore parallel programming is essential. Asynchronous calls and better reactive system program structures have increased to use these manycore machines. A future is a language construct that enables programmers to quickly and easily reveal parallelism in programming languages. As an alternative to low-level constructs such as locks and threads, the choice of futures has several advantages [22]. In comparison to more lower-level parallel models, a parallel futures library in C++ would offer all the advantages of safety, maintainability, and programmability while guaranteeing performance and considerable scalability with low overhead. Futures provide a way to manage return values from asynchronous calls. The method caller receives a "placeholder" future object representing the return value as soon as an asynchronous call is made, and the call itself continues to run simultaneously. The method caller can obtain the call's return value in the future after it has been processed. If the value is requested before the call has finished, the future suspends the requesting thread's execution until the value of the future becomes available. The principal design rationale behind futures is that *"the*

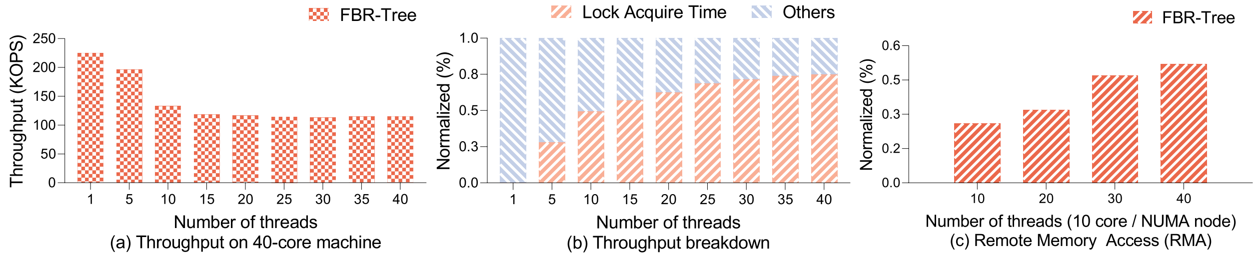


FIGURE 1. Motivational experiments performed on FBR-tree [14]. KOPS represent Kilo Operations per Second.

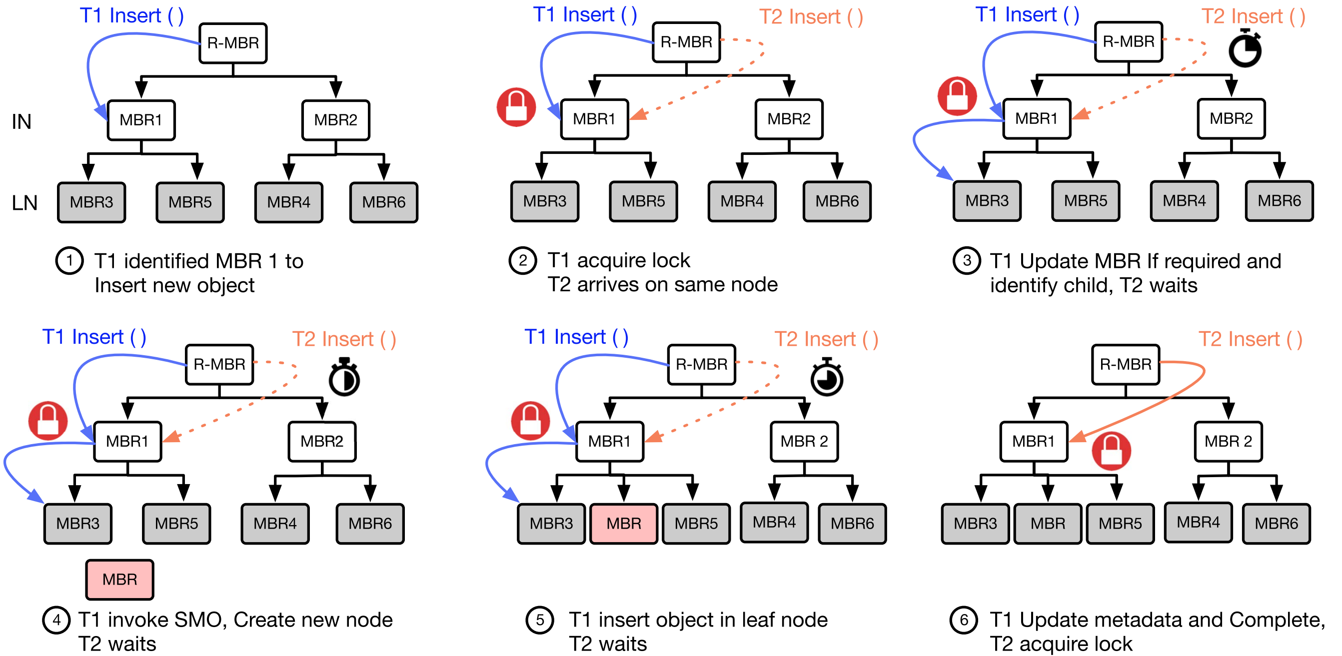


FIGURE 2. Use case for multi-thread insertion to illustrate the scalability limitation in FBR-tree [14].

programmer takes on the burden of identifying what can be computed safely in parallel, leaving the decision of exactly how the division [of work] will take place to the runtime system" [23].

Adopting futures for the scalability of index data structures on DCPM-based manycore machines is a relatively new topic and has recently been explored for B^+-tree, a well-known but simple tree-based data structure as F^3-tree [8]. B^+-tree is a single-dimensional data structure that stores simple key values and a sorted data structure. On the other hand, R-tree stores spatial data in terms of minimum bounding rectangles (MBR) and are composed of considerably more complicated multidimensional workloads. F^3-tree shows that adopting futures for DCPM-based manycore machines could be advantageous to achieve high scalability but lacks a clear statement about whether future objects are favorable to utilize in complex data structures such as R-tree. In this work, we explored futures for complex and multidimensional data structures and investigated futures to apply to multidimensional data structures for concurrency.

D. NUMA EFFECT ON FBR-TREE

FBR-tree design does not consider NUMA architectures and suffers from the NUMA effect on manycore machines. The difference in the latencies on the CPU's local memory and remote memory is known as the NUMA effect and is usually caused by cross-CPU node communication [3], [24]. To investigate the NUMA effect on FBR-tree, we performed experiments with an increasing number of threads within and across the NUMA node of our machines. Our evaluation platform consisted of four NUMA nodes where each node has 10 physical cores (for the details of the experiment setup, please refer to Section V). Figure 1(c) shows the normalized percentage of remote memory access (RMA) of the FBR-tree. *Perf tool* [25] was used to investigate the RMA. It can be seen in Figure 1(c) that the FBR-tree performs RMAs even for 10 threads, which is within the NUMA boundary of our testbed. This is because in FBR-tree, the threads are not bonded to any CPU cores and are being assigned across the system. RMA increases considerably as the number of NUMA nodes increase. However, the central thought from

Figure 1(c) is straightforward: the DCPM-based data structure is severely affected by the lack of NUMA-awareness, as exemplified by the large percentage of remote memory accesses.

III. PROBLEM DEFINITION

In this section, we provide details on the limitations of concurrency in FBR-tree and form the basis for the problem we are targeting in this work.

A. CONCURRENT INSERT OPERATIONS (INCLUDING UPDATE)

FBR-tree faces a performance constraint when executed in a high degree of concurrency, and this is due to its design and development choices. For example, a critical section of FBR-tree is composed of several sub-operations including acquiring mutex lock, updating MBR if required, and SMO operations including split or merge, if required.

Figure 2 shows a concurrent write operation in FBR-tree, where two threads, Thread 1 ($T1$) and Thread 2 ($T2$) perform an insert operation. $T1$ identifies the candidate node where it needs to insert a new spatial object (*MBR1* in this case). Note that we represent the MBR in the figure to make it simple instead of the node that contains multiple MBRs, as shown in step ① of Figure 2. In step ②, $T1$ acquires the lock on candidate node MBR1 while $T2$ also has to perform the insert operation at the same node based on heuristic algorithms [21]. Since $T1$ has acquired the lock of node MBR1, $T2$ has to wait to perform its operation of the same node. Note that both threads select MBR3 (leaf node) as their candidate for key insertion as shown in step ③.

Now the $T1$ will identify the leaf node where the actual insertion is to be made (i.e., MBR3 as shown in step ③). As $T1$ proceeds, it first checks the capacity of the candidate leaf node (MBR3) and if it is overflown then $T1$ triggers an SMO; otherwise a new MBR is simply inserted and $T1$ releases the lock. However, in step ④ of Figure 2, an SMO is triggered by $T1$. During the SMO, $T1$ allocates a new node (denoted as MBR in step ④), and migrates half of the entries of MBR3 to MBR. The pointer of the MBR is updated in the parent node (MBR1) and the corresponding new insert operation proceeds on either one of the nodes, MBR or MBR3, and $T1$ releases the lock as shown in step ⑤. Now, $T2$ is finally able to acquire the lock on MBR1, and it finds out that MBR3 has been split. $T2$ will again call the heuristic algorithm and identify the node where the entry needs to be inserted.

The simplest technique to ensure correctness in index data structures is to obtain a lock on the shared data and allow only one thread to write at a time. This serializes the operations and reduces the performance of the data structure with an increasing number of threads. In FBR-tree, we identified that a thread needs to acquire a lock on a sub-tree to perform its write operation, as shown in Figure 2, which eventually leads to whole tree being locked. For instance, if the root node is locked, the whole tree will be locked until that lock is released by the corresponding thread.

B. CONCURRENT LOOKUP OPERATIONS

FBR-tree supports lock-free read operation and relies on atomic metadata update operations for the correctness of the data structure. With lock-free read operation, FBR-tree can allow multiple threads to concurrently perform a read operation. In FBR-tree, a read thread will first read the metadata of the leaf node, as the spatial objects are stored at the leaf nodes of the tree. Once the metadata is read, the read thread will access the spatial object and verify the state of the spatial object by accessing the metadata once again. If the state of metadata is consistent, the spatial object will be returned to the application. Otherwise, it will perform the read operation once again. This optimistic approach always returns consistent data to the application thread and thus supports concurrent read operations.

C. CONCURRENT DELETE OPERATIONS

The delete operation in FBR-tree takes advantage of the lock-free read operations to identify the leaf node where the entry to be deleted can be found without any concurrency limitations. A delete thread will simply flip the valid bit of the deleting entry in the bitmap of the leaf node atomically without acquiring any lock. The invalid entry in the leaf node of FBR-tree is updated using an in-place update operation with a new entry. In addition, delete operations do not deal with SMOs in FBR-tree since SMOs are only entertained by the insert operations. Thus, delete operations have a small critical section and do not cause concurrency limitations.

This blocking mechanism limits scalability, i.e., concurrent writes accessing the FBR-tree on DCPM-based manycore machines where hundreds or thousands of application threads are withstood to perform the write operation. To address the aforementioned limitation of FBR-tree, we proposed MPR-tree, and our work focused on designing a scalable multidimensional index data structure for DCPM-based manycore machines. To the best of our knowledge there has been no prior attempt to address this problem, and our work is unique in performing a scalability study for multidimensional indexing such as R-tree on DCPM-based manycore machines.

IV. MPR-TREE DESIGN

In this section, we provide the design overview of our proposed MPR-tree followed by operational flow, correctness guarantee and NUMA-awarenes.

A. SYSTEM OVERVIEW

The design objectives of MPR-tree are to achieve concurrency, crash resilience, and NUMA-awareness while maintaining read performance. Our design approach was inspired by an asynchronous computing design idea that is a producer and consumer model. The producers are responsible for serving application requests, while the consumers manipulate the shared index data structure. Figure 3 shows a design

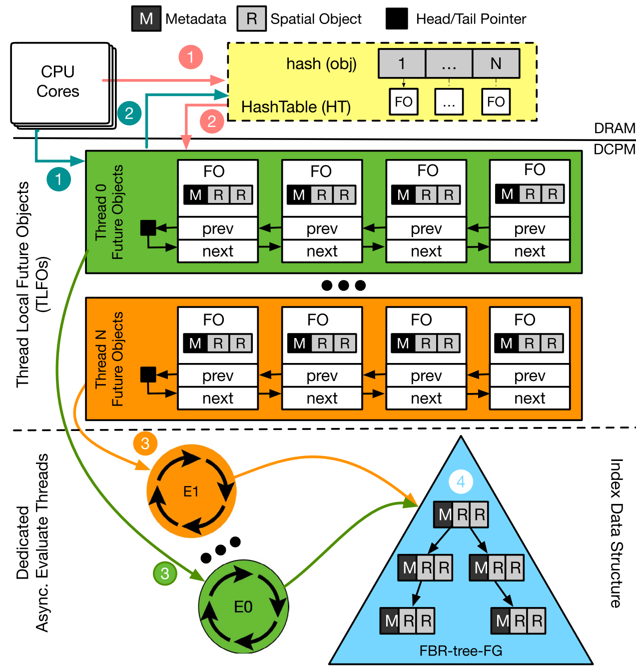


FIGURE 3. Design overview and the operational flow of our proposed MPR-tree (manycore-aware persistent R-tree). The three-layered design of MPR-tree composed of index data structure, thread-local future objects and in-memory hash table.

overview of our proposed MPR-tree. Our solution is composed of three components divided into DRAM and DCPM. The DCPM-based components include thread-local future objects (TLFO) and a modified version of FBR-tree while we adopted a DRAM-based in-memory hash table to maintain the read performance. TLFOs are designed as lock-free doubly linked-list and rely on durable linearizability [26] to achieve crash consistency. Further, we modified the FBR-tree to adopt a fine-grained locking mechanism by changing it to acquire a per-node lock instead of sub-tree, and we refer to it as FBR-tree-FG. Lastly, to avoid the linear traversal of TLFOs, we adopted an in-memory hash table to provide direct access to spatial objects stored at TLFOs. To achieve NUMA-awareness, we bound the memory allocations of TLFOs to the CPU's local memory modules and introduced node-local consumers.

B. OPERATION FLOW

Figure 3 shows the insert, search, and delete operation flow of our proposed MPR-tree. The green arrows in Figure 3 along with dedicated asynchronous *evaluate* threads show the insert operation flow. During the insert operation, the application thread will first insert the spatial object to the thread-local linked-list as shown in step ❶. Once the object is written to the TLFO, an entry will be made in the in-memory hash table for lookup as shown in step ❷. Algorithm 1 shows the pseudo-code of the insert operation at TLFOs. In our design, the delete and search operation use a distinct path (shown in red arrows) as compared to the insert operation.

Algorithm 1 Insert FO to TLFOs

1: SO: Spatial Object
2: T_{id}: Thread ID for TLFO
3: $TLFO_{T_{id}}$: Thread Local Future Objects of Thread T_{id}

4: **procedure** insert_futureObj(SO, T_{id})
5: Identify the corresponding TLFO by T_{id}
6: **if** FO_1 not full **then**
7: Insert SO to FO_1 of $TLFO_{T_{id}}$
8: // FO_1 represents the 2nd node of $TLFO_{T_{id}}$
9: Update metadata
10: // Update number of entries in the FO_1
11: Insert SO entry to Hash Table
12: **else**
13: Allocate new FO
14: Update Next node pointer of Head node
15: Update Next node pointer of new FO to point to previous FO_1
16: //Now new FO becomes FO_1
17: Call `FLUSH+FENCE`
18: Update the previous pointer of new FO_1 to Head node
19: Update the previous pointer of old FO_1 to new FO_1
20: Insert SO to new FO_1 of $TLFO_{T_{id}}$
21: Update metadata
22: // Update number of entries in the FO_1
23: Insert SO entry to Hash Table
24: **end if**
25: **if** SizeOf(FO_1)>=CACHELINE **then**
26: Call `FLUSH+FENCE`
27: **end if**
28: **end procedure**

Our design supports hierarchical spatial object lookup operations for both search and delete. To perform lookup, a spatial object is first identified in the hash table (step ❶ in Figure 3) and if an entry is found, the corresponding TLFO is accessed directly and returns the spatial object, as shown in step ❷. Otherwise, FBR-tree-FG is traversed in a traditional look-free manner. On the other hand, the dedicated asynchronous *evaluate* threads are responsible to checkpoint the spatial objects from TLFOs to global FBR-tree-FG, steps ③ of Figure 3. Each *evaluate* thread is responsible for specific TLFOs and only checkpoint the data from assigned linked-lists. The evaluation of TLFOs can be done based on several options; data in TLFOs can be written to FBR-tree-FG based on the amount of time or size of the TLFOs. Algorithm 2 shows the pseudo-code of the *evaluate* operation.

C. FINE-GRAINED LOCKING OF FBR-TREE-FG

As discussed in section III, when acquiring the lock on the internal node, FBR-tree completely locks the sub-tree. In tree-based index data structures, every locking action

Algorithm 2 Evaluate TLFOs to FBR-tree-FG

1: **procedure** Evaluate(*TID*)
2: Identify the corresponding *TLFO* by T_{id}
3: Traverse to the Nth FO of $TLFO_{T_{id}}$
4: **while** ($TLFO_{T_{id}}$ head pointer != $TLFO_{T_{id}}$ tail pointer) **do**
5: **while** $FO_{count}! = 0$ **do**
6: Call rtree_insert() with *SO* from FO_N of $TLFO_{T_{id}}$
7: // *rtree_insert* is same as FBR-tree
8: update metadata
9: Remove the *SO* entry from Hash Table
10: **end while**
11: Update $TLFO_{T_{id}}$ tail to FO_{N-1}
12: De-allocate FO_N
13: **end while**
14: **end procedure**

begins at the root of the tree and progresses down to the leaf node, locking each node along the way. If it comes across a locked node that isn't the root, the locking procedure is either aborted (failure) or set to wait and try again (busy wait).

Consider the same example from Figure 2 in section III. Let's say two threads, thread $T1$ and thread $T2$, identify the MBR where they need to perform the insert operation and only one thread will be able to continue and the other thread has to wait. As shown in Figure 4(1), $T1$ wants to acquire a lock on MBR3 and $T2$ wants to perform an operation on the same parent node. In FBR-tree, if $T1$ wins to acquire the lock, $T2$ will have to wait until $T1$ finishes its operation, which is inserting a new MBR into the leaf node. This causes $T2$ to wait for a long time and increase the lock acquire time, which eventually degrades the performance of the application. To mitigate the performance drop of the locking mechanism of FBR-tree, we adopted FBR-tree-FG, a fine-grained lock-release-lock mechanism atop FBR-tree.

Figure 4 shows the lock-release-lock operation flow. In FBR-tree-FG, after $T1$ wins to acquire the lock, it will perform the internal/parent node operation as shown in Figure 4 step ①, while $T2$ has to wait to acquire the lock. Once $T1$ starts the insert operation after identifying MBR3 to insert a new object, if the parent of MBR3 (MBR1 in this case) is needed to extend, i.e., the existing MBR1 does not have the capacity to accommodate the new spatial object, MBR1 will be extended to accommodate the new spatial object. However, step ②, $T1$ will release the lock of the parent/internal node and acquire the lock on the desired child node (MBR3 in this case) and will repeat the same on the tree until the leaf node. Since the lock on MBR1 is released by $T1$, $T2$ now has the opportunity to grab the lock on MBR1 and perform its operation on MBR5.

The lock-release-lock mechanism for FBR-tree reduces the lock contention among threads where few threads try to perform the operation on the same sub-tree of the parent/internal node. This will give threads the opportunity to acquire the lock and concurrently perform operations on the sub-tree of the targeted node. However, the performance of the overall application does not scale due the inherit lock contention, where the insert operation cannot be performed without acquiring a lock and hundreds of threads are waiting to acquire the lock on the same node. In FBR-tree-FG, the lock-release-lock strategy gives some extra space to achieve better performance and reduce the lock contention time among threads when the number of threads are small.

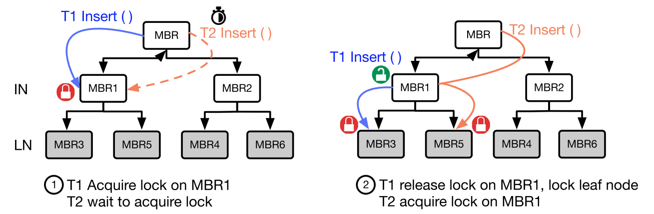


FIGURE 4. Lock-release-lock mechanism for FBR-tree, i.e. FBR-tree-FG.

D. THREAD LOCAL FUTURE OBJECT

A future is a data item that promises to provide an operation's results when it is ready [20]. In our design, a future object constitutes an array of spatial objects, an entry count, and the previous and next FO pointers, as depicted in Figure 3. Thread local future objects are stitched together by a doubly linked-list for each thread.

We used a doubly linked-list for two main reasons. First, the doubly linked-list reduces contention between applications and asynchronous evaluate threads by allowing application threads to modify the linked-list only from the head pointer. Meanwhile, asynchronous evaluate threads checkpoint the future objects from the tail pointer, as shown in Figure 3. Second, to maintain a consistent view of the doubly linked-list in case of failure. For instance, if a crash happens in between updating the head pointing to the newly allocated FO, as shown in step ② and step ④ of Figure 6, the recovery mechanism will still be able to access the allocated FOs by traversing through the tail pointer of the head node. With DCPM, a consistent view of TLFOs is a critical problem. Note that we do not rely on any existing locking mechanism while consuming data from TLFOs. Therefore, we adopt durable linearizability to ensure that each operation takes effect in sequential order.

E. PERSISTENCY ACHIEVED IN TLFOS

In concurrent data structures, if each operation takes effect in between the method's invocation and response [27], that concurrent data structure is known as being linearizable. In DCPM, a durability guarantee is additionally required for crash consistency so that the data will be persistent and crash resistant.

A *durably linearizable* concurrent data structure satisfies the linearization property. In addition, after a full-system

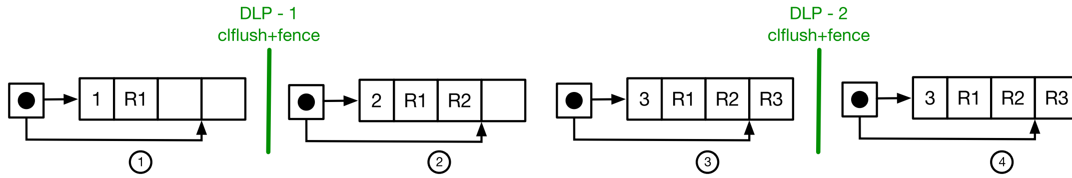


FIGURE 5. Single future object attaining a durable linerizability point with a single linked-list.

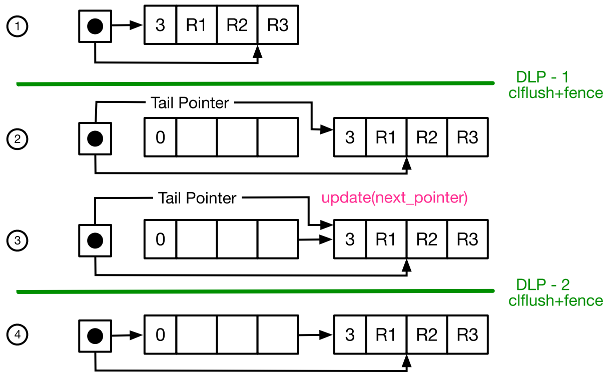


FIGURE 6. Multi future object achieving durable linerizability point.

crash, the data structure's state must reflect a consistent sub-history of activities that includes all operations completed by the time of the crash. We use the term *durability point* as the point in the execution history where an operation becomes durable, i.e., its effects are visible to other threads and persistent. If we run the recovery process after a durability point, the data structure will be in a consistent state and it will be persistent. The order of execution can be expressed in terms of durability points, with each point implying a different sequence of operations. We do not consider durability as the same for the global FBR-tree-FG because it follows the design principle of FBR-tree. We achieve *durable linearizability* (DL) for single and multiple TLFOs as shown in Figure 5 and 6, respectively. Figure 5 and 6 show two examples for achieving a DL point.

Figure 5 shows an example where we achieve the DL within a single FO by atomically updating the spatial object. Steps ① to ④ in Figure 5 show the write operation within an TLFO. The lines labeled DLP represent the durable linearizability points achieved by the insert operation by calling `FLUSH+FENCE` instructions. In Figure 5, two DLPs, DLP 1 and DLP 2, are achieved. If a crash happens between DLP 1 and DLP 2 (i.e., during step ② or step ③), the recovery mechanism will be able to achieve a consistent view of the thread-local linked-list up to DLP 1.

Figure 6 represents the use-case where we achieve the DL between multiple FOs by atomically updating the next pointer. At step ①, we present a consistent and fully utilized FO of a particular TLFO with durability being achieved. At step ②, a new FO is allocated as new spatial objects are being inserted at the TLFO. Once a new FO is allocated, we update the next pointer atomically as shown in step ③ and achieve the second durability point as shown in Figure 6. We consider the next pointer update for newly allocated FOs as the durability point because if a system crashes, we can still be able to traverse the newly allocated FOs by backward traversal. A system failure after step ② will result in potential memory leak problem which need to be addressed. Several mechanisms can be adopted for memory leaks such as hazard eras [28] and the optimistic access scheme [29].

F. SPACE AND TIME COMPLEXITY

The space overhead of our proposed design is similar to the traditional R-tree i.e., $O(N)$ because an object is either in the TLFO or in the fine-grained FBR-tree. Also, since the in-memory hash table is placed in DRAM, 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 objects stored in the TLFO entries at a time, T. The time complexity for the search operation is composed of two cases, one where a thread is required to traverse the TLFO and the second where a thread only looks for the object in the fine-grained FBR-tree. In addition, the read operation needs to traverse through the in-memory hash table. Now, if a thread is looking for an object R it first needs to search the hash of the object R in the hash table, which causes the best time complexity $O(1)$. If the object R is found in the hash table, the thread will traverse the particular TLFO linearly and return once the object R is found. The time complexity (T) for this case is $O(1) + O(M)$. For the second scenario, if the object is not found in the in-memory hash table, the thread will directly look for the object in the fine-grained FBR-tree. Since our fine-grained FBR-tree is atop FBR-tree it offers lock-free read operations.

G. NUMA-AWARENESS IN MPR-TREE

The NUMA effect on DCPM index data structures is significant and plays an important role in the overall performance of the applications. As discussed in Section II-D, FBR-tree causes additional remote memory access, that is, threads allocate memory pages across the nodes. It has been observed in previous studies that compared to local node memory allocation for DCPM write, the peak bandwidth of the remote node memory decreased to 59% [24]. Figure 7 shows how we achieve NUMA-awareness in our design for a two-socket architecture (CPU Node 0 and CPU Node N). The key idea here is to make the design of MPR-tree NUMA-aware.

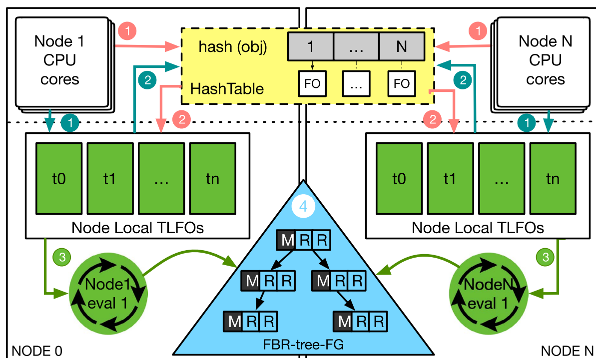


FIGURE 7. NUMA-awareness in MPR-tree design.

We achieve NUMA-awareness in our design by bounding the memory allocation of the TLFOs to the local memory of CPU nodes of the application threads, as shown with the per CPU node TLFOs in Figure 7. To further achieve NUMA-awareness for MPR-tree, we bind the asynchronous *evaluate* threads to only perform checkpointing of the local CPU node's TLFOs and thereby avoid accessing cross-CPU node TLFOs, as depicted by the node-specific *evaluate* thread in Figure 7.

With CPU node-local *evaluate* threads, we avoid unnecessary cross-CPU node communication. In contrast, global *evaluate* threads would have to perform remote memory accesses to checkpoint the TLFOs of the application threads from remote CPU nodes. With controlled memory allocation and CPU-bounded asynchronous *evaluate* threads, our design reduces the amount of remote memory allocations and cross-CPU node communication and improves scalability. However, the asynchronous *evaluate* threads still perform remote memory accesses because of the memory allocation scheme of the raw index data structure, which suffers from inherent concurrency limitations (as discussed in Section II-D). We did not bound the memory allocations of the FBR-tree-FG because we opted not to modify the base design of FBR-tree.

H. DISCUSSION

In this work, we mainly focused on the performance of insert operations of state-of-the-art FBR-tree, as these operations are responsible for all the maintenance of the FBR-tree. Thus, we identified the limitations of the insert operation in FBR-tree and proposed solutions to overcome these limitations. First, FBR-tree is composed of a huge critical section that limits its performance when the degree of concurrency is high. We broke down the critical section with a top-down approach and employed a fine-grained lock-release-lock approach, FBR-tree-FG, and showed that our fine-grained approach outperforms the baseline FBR-tree. For further performance enhancement, we adopted the future-based per-thread local doubly linked-list (TLFOs) atop FBR-tree-FG and showed that it can support a high degree of parallelism. Furthermore, we identified the limitation of TLFOs on NUMA machines and proposed CPU node local TLFOs and evaluate threads. For lookup operations, we employed in-memory hash tables to directly access data objects stored at TLFOs that rely on lock-free search on the FBR-tree-FG. To address the delete operation, we relied on the baseline mechanism proposed in FBR-tree, as delete operations do not have a large critical section.

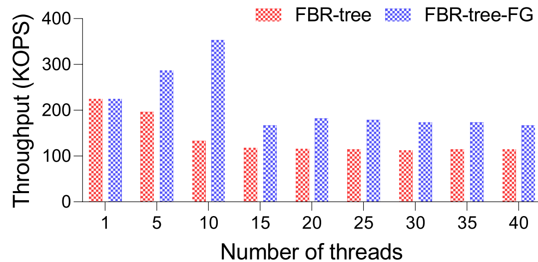


FIGURE 8. Concurrency analysis of FBR-tree-FG with FBR-tree.

V. EVALUATION

In this section, we first show the experimental setup. Then, we present the evaluation results of MPR-tree for insert and search queries. Next, we show the effect on performance by varying the size of the TLFOs, and finally we show the performance of the NUMA-aware MPR-tree.

A. TEST-BED SETUP

We performed the experiments on a Linux machine (kernel version 5.4.0) equipped with four 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 emulated the latency of Intel DCPM as presented in [3]. We implemented our proposed MPR-tree using a Persistent Memory Development Kit (PMDK) [30]. 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.[2] 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 and lookup operations with a 500K queries 3D workload into MPR-tree.

B. CONCURRENCY ANALYSIS

1) FBR-TREE vs FBR-TREE-FG

As discussed in section IV-C, we adopted lock-release-lock approach for FBR-tree that we call fine-grained FBR-tree i.e. FBR-tree-FG. As shown in Figure 8, it can be observed that FBR-tree-FG performs 0.6× on average compared to the baseline FBR-tree. This is because the lock-release-lock mechanism reduces the lock contention among threads. In FBR-tree-FG, a thread only acquires the lock on a sub-tree

[2]The data set can be found here: https://archive.ics.uci.edu/ml/index.php

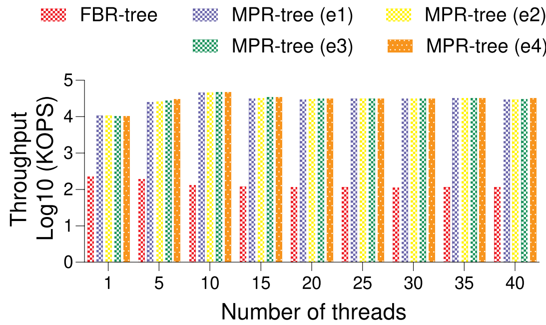


FIGURE 9. Scalability analysis of MPR-tree on manycore machines. In MPR-tree, e_i represents the number of asynchronous evaluate threads.

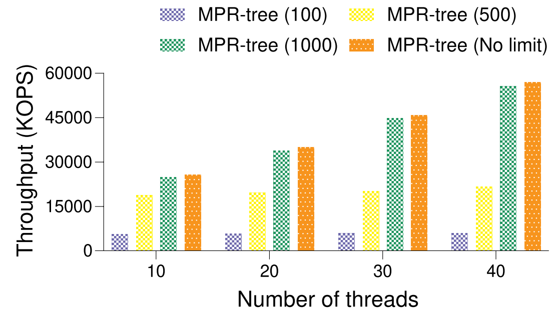


FIGURE 10. Impact of Thread-local future object (TLFO) size on performance.

until it enlarges the parent node (if it needs to) and checks the corresponding child node, where the new spatial object will be inserted. If the child node requires an SMO within the parent node's lock, the thread will perform an SMO, and after the split/merge is done, the thread will release the lock on the parent node and acquire the lock on the child node to complete the insert operation. On the other hand, if the corresponding child node does not require an SMO, the thread will release the lock on parent nodes and acquire the lock on the child node and complete the insert operation. This gives other threads the opportunity to concurrently acquire the lock and perform operations on the sub-tree. However, the performance of FBR-tree-FG does not scale after crossing the NUMA boundary, as shown in Figure 8. This is due to two reasons; first, the inherent lock contention as shown in Figure 1(b). Even though FBR-tree-FG adopted the fine-grained locking mechanism, it still suffers from lock contention. Second, the threads perform remote memory accesses, which sharply drops the memory bandwidth. We can observe that FBR-tree-FG gives some extra space to achieve better performance than baseline FBR-tree and reduces the lock acquire time among threads when the number of threads is small.

2) FBR-TREE vs MPR-TREE ATOP FBR-TREE-FG

Figure 9 depicts the results of our concurrency analysis. We varied the asynchronous *evaluate* threads in this experiment from 1 to 4 to show the impact on performance. We can clearly observe that MPR-tree outperforms FBR-tree ranging from 1.7× to 2.1× by varying the number of threads from 1 to 40 on Log10 scale. It can also be seen that MPR-tree's performance saturates as the number of threads increases in CPU node, 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 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.

C. TLFOS PERFORMANCE IMPACT

TLFOs are the main component of our design, and play a significant role in application performance. We showed the performance impact of TLFOs by limiting the number of TLFOs. Figure 10 shows the performance impact of varying the size of TLFOs. In this experiment, we limited the number of future objects, i.e., the numbers of nodes in linked-list to 100, 500, and 1000 shown as MPR-tree(100), MPR- tree(500) and MPR-tree(1000), respectively. We only used four asynchronous *evaluate* threads for this experiment. We can clearly observe in Figure 10 that the degree of concurrency of our proposed system is limited by the number of future objects, i.e., the size of TLFOs. This is because the application threads (producers) spend most of their time waiting for the *evaluate* threads (consumers) to checkpoint the data from TLFOs to the FBR-tree-FG. Furthermore, we can also observe that MPR-tree with No Limits on TLFOs, where we do not limit the number of future objects, has the best overall performance. FBR-tree(1000) shows nearly comparable performance to FBR-tree(No limit), as the number of TLFOs is less than the threshold of 1000 future objects, and it does not block the producers during the entire execution of our application.

D. MPR-TREE WITH NUMA CONSIDERATION

In this subsection, we present the results of evaluating MPR-tree NUMA-aware compared to FBR-tree and other variants of MPR-tree. For a fair investigation of NUMA's impact on MPR-tree, we evaluated FBR-tree performance with three variants of MPR-tree based on different memory allocations and physical core pinning policies. For the baseline, we used FBR-tree without pinning the application threads to the physical CPU core and let the compiler decide the thread and memory allocation. We denoted it as simple FBR-tree, and the same policy was applied to MPR-tree. Next, we bound the application threads to only perform memory allocation to the memory devices of local CPUs, that is, a thread running

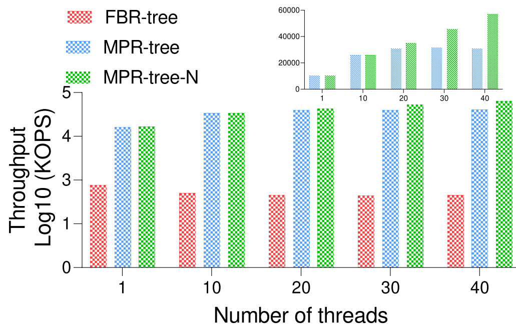


FIGURE 11. MPR-tree-N performance comparison with base FBR-Tree and MPR-tree on manycore machine. We fixed the evaluate threads to four and threads were bound to CPU cores. The subplot shows the zoom in a pattern of log10 results.

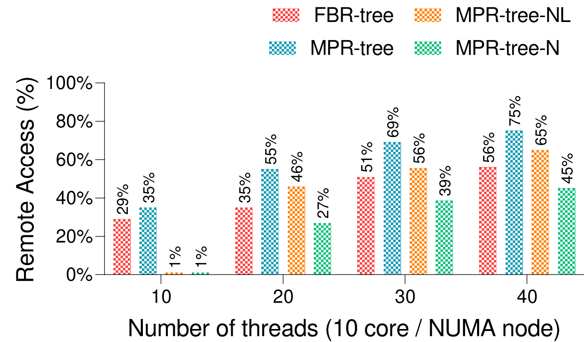


FIGURE 12. Comparison of cross CPU node memory accesses between FBR-tree, MPR-tree, MPR-tree-NL and MPR-tree-N.

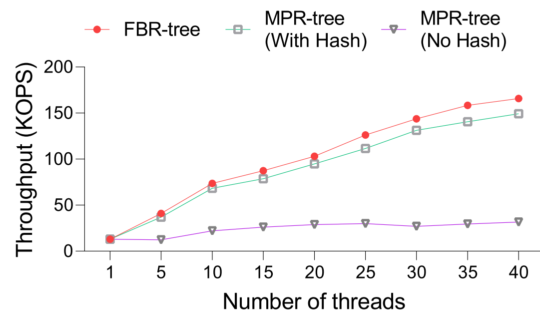


FIGURE 13. Search performance analysis on manycore machines.

on CPU 1 will only perform allocations at CPU 1's memory modules. We refer to it as FBR-tree (NL). Lastly, the memory allocation policy is the same as that of MPR-tree-NL and we additionally introduced NUMA-awareness to our evaluate threads as described in section IV-G. We bound these memory allocations using numactl [31].

Figure 11 depicts the performance for the NUMA-aware MPR-tree (MPR-tree-N). We performed the experiment by increasing the number of concurrent NUMA boundary threads with insert operations. On average, MPR-tree-N outperforms FBR-tree by $2\times$ on log10 scale and MPR-tree by $2\times$. That is because MPR-tree-N performs node local memory allocations, while MPR-tree relies on the compiler's memory allocation policy. Additionally, evaluate threads in MPR-tree-N are only bounded to checkpoint CPU local TLPFOs. This minimizes cross-CPU node communication and reduces the cache coherence overhead (cache ping-pong [8], [32]) of shared data. Furthermore, Figure 11 also shows a zoom-in comparison of MPR-tree and MPR-tree-N for throughput in KIOPS. It can be observed that MPR-tree-N outperforms MPR-tree, as the number of threads increase for MPR-tree-N.

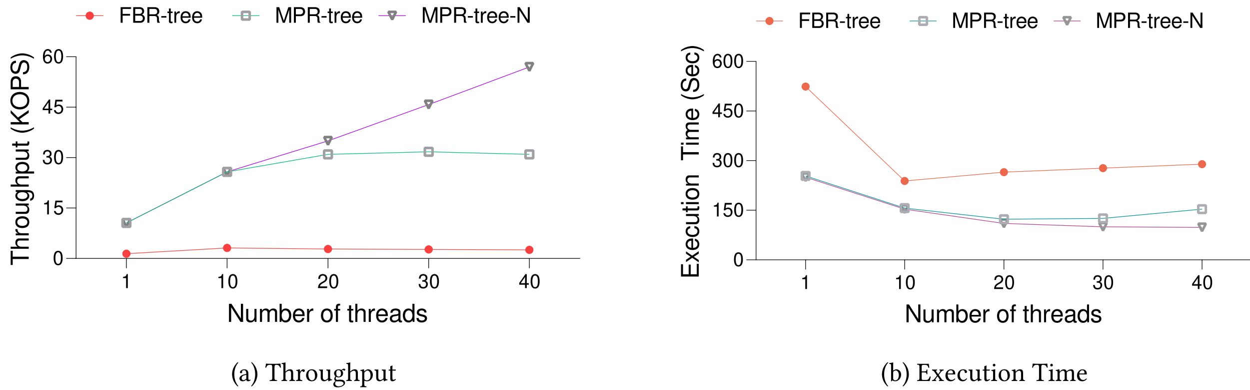
Figure 12 represents the percentage of remote memory accesses for the NUMA-aware MPR-tree. We compared the remote memory accesses with FBR-tree and MPR-tree variants explained above. For all MPR-tree variants, we limited the number of evaluate threads to four. It can be seen in Figure 12 that MPR-tree-N exhibits the lowest percentage of remote memory accesses due to its node-local memory allocation. However, it can also be observed that MPR-tree-N and MPR-tree-NL still perform remote memory accesses due to all the allocations being performed in the reserved memory pool as described in section V-A. We think user-level applications cannot do much to prevent NUMA issues in the interleave mode since it is beyond the application's scope to control the memory pool.

E. LOOK UP 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 13. We compared FBR-tree with our proposed MPR-tree (with Hash), which includes the in-memory hash table and MPR-tree (No Hash) which does not contain the in-memory hash table. For this experiment, we placed the initial 80% of the loaded 1M objects to the global FBR-tree-FG while left the remaining 20% to the TLFOs. MPR-tree with ''No Hash'' had the worst read performance because every read operation has to traverse linearly over the TLFOs first, and if the object is found it will return and if the object is not found, it then searches the global tree. In contrast, 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 has to check the hash table and then look for the object in the corresponding TLFOs

F. REALISTIC WORKLOAD

We simulate a realistic workload scenario where an application performs a mixed workload of insert, delete and search operations. For this experiment, we used time series multi-dimensional taxi service trajectory polylines workload containing GPS coordinates. During the experiments, threads were divided into 500K insert operations with 50% warm-up data, 250K lookup and 12K delete operations performed on MPR-tree. Figure 14 (a) shows that the MPR-tree-N with NUMA outperforms MPR-tree and FBR-tree and improved performance around $1.6\times$ and $13\times$, respectively when the number of threads increase. This is due to high-performing



(a) Throughput

(b) Execution Time

FIGURE 14. **Realistic workload analysis on manycore machines with 500K insert, 250K lookup, and 12K delete operations.**

insert operations of the MPR-tree where the application threads only write to the TLFOs and the in-memory hash table improves the performance for search operations. Furthermore, Figure 14 (b) shows the execution time of the FBR-tree, MPR-tree and MPR-tree-N. It can be observed that MPR-tree-N outperforms FBR-tree by a considerable margin while it manages to achieve faster execution times as compare to MPR-tree.

VI. RELATED WORK

A. R-TREE INDEXING DATA STRUCTURES

R-tree indexing data structures have been extensively studied [15], [33], [34]. The R-tree [35] is a dynamic index suited for multi-dimensional data objects. Guttman [35] has proposed three methods to split a R-tree node: (1) the exhaustive method, (2) the quadratic method, and (3) the linear method. In the quadratic algorithm, two initial objects are first identified. For each of the remaining objects, compute the difference of the increase in the area of the covering rectangles if that object were to be added to each of the two partitions. The object with the most noticeable difference is picked out and allotted to the partition with the smaller rectangle enlargement. In the Linear R-tree algorithm, two seed objects are identified, and the remaining objects are assigned to each partition with a minimum increase in the area of the bounding rectangle.

One drawback of the above algorithms is their lack of optimizing the size of the overlapping regions among the split partitions. Thus, if a query range intersects any of these overlapping regions, multiple sub-trees have to be descended to answer the query. The overlap has been reduced or perhaps completely eliminated using algorithms. There are several studies that introduce new R-tree variations and the accompanying algorithms, e.g., [15], [33], and [34].

The R*-tree adopts a combined optimization that minimizes both the areas and the overlap between the enclosing rectangles [16]. Upon a split, the R*-tree generates several candidate distributions and computes three goodness measures: the area, the margin, and the overlap. By optimizing these measures, the R*-tree reduces the number of paths to traverse during a search. By re-inserting the object that is furthest from the bounding rectangle's center into another node inside the same tree level, the R*-tree delays a node split.

The Revised R*-tree (RR*-tree) [16] enhances the R*-tree in several ways. The reinsertion policy of the R*-tree is abandoned. Also, the R*-tree has a relatively expensive overlap optimization that is only performed in the lowest non-leaf level (the one above the leaf nodes). With a redesigned algorithm, overlap optimization can be applied to all non-leaf levels in the RR*-tree. Also, the balance of the splitting pages is added as an optimization criterion, and another improvement is in high-dimensional data. Since it is possible that bounding boxes with zero volume occur when using high-dimensional data, volume-based optimization becomes less effective. In such situations, a lower dimension perimeter-based optimization is used.

Since the adoption of manycore machines in various facilities, index data structures, adopted in various system-level applications such as spatial databases, are expected to have capabilities to exploit the performance characteristics of these manycore machines. However, most of the tree-based multi-dimensional index data structures suffer from inherent concurrency issue, such as lock overhead. To overcome this limitation, we proposed MPR-Tree which exploits the performance characteristics of manycore machines by adopting thread-local future objects and fine-grained lock-release-lock mechanism for state-of-the-art FBR-Tree. Our proposed solution can be adopted to spatial database and filesystems that deal with multi-dimensional data.

Further, it has been in discussion by the Intel that they will be winding down their Optane Memory [36]. However, Intel has also mentioned that they will keep supporting their already in use Intel Optane Machines and server families. Since this announcement, there have been several major updates announced by the Intel to their Persistent Memory Development Kit. Additionally, our proposed solution can be adopted to any high performance memory device such as CXL supported memory devices [37].

B. NUMA-AWARE SYSTEM APPLICATIONS AND DATA STRUCTURES

There have been several studies to introduce NUMA-awareness to file systems and system-level applications. nCache [38] investigated the adoption of range locks on shared files in manycore servers to execute concurrently and proposed a novel file metadata cache framework by ensuring consistent updates. Several studies have been conducted to identify the NUMA impact on system-level software and data structures with DCPM [4], [9], [24]. All these studies conclude that there is a need to design NUMA-aware data structures that can use NUMA-based manycore machines efficiently. Several recent prominent studies, including [3], [39], highlighted the importance of NUMA's impact. Daase et. al [40] investigated OLAP-related workload interactions across NUMA regions. June-Hyung Kim et al. [4] proposed fine-grained range-based locks to improve the scalability of NOVA [41] with NUMA architectures. NAP [24] established black-box NUMA-aware counterparts for the index data structure for DCPM.

VII. CONCLUSION

In this paper, we presented MPR-tree, a highly concurrent persistent R-tree for DCPM-based manycore machines. To achieve concurrency, we adopted fine-grained locking over the state-of-the-art FBR-tree, an R-tree variant for DCPM. Furthermore, MPR-tree achieves scalability and high write concurrency by adopting thread-local future objects (TLFOs) atop FBR-tree-FG. TLFOs are checkpointed to the global FBR-tree-FG in an asynchronous manner based on a tunable time and size-based threshold. Search queries are optimized by employing a volatile in-memory hash table. We introduced NUMA-awareness to our MPR-tree by bounding the memory allocations of TLFOs to local DCPM nodes and evaluate threads to CPU node. We showed experimental proof that MPR-tree has relatively better performance for its counterpart FBR-tree with 2× improvement in scalability on a log10 scale, and by carefully adopting a memory allocation policy we reduced 1.3× remote memory access in MPR-tree.

ACKNOWLEDGMENT

(Abdul Salam and Safdar Jamil contributed equally to this work.)

REFERENCES

[1] *3D Xpoint™: Intel™ OptaNet Persistent Memory*. Accessed: Jun. 6, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html

[2] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *Proc. 50th Annu. Design Autom. Conf. (DAC)*, 2013, pp. 1–10.

[3] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th USENIX Conf. File Storage Technol. (FAST)*, Feb. 2020, pp. 169–182.

[4] J.-H. Kim, Y. Kim, S. Jamil, and S. Park, "A NUMA-aware NVM file system design for manycore server applications," in *Proc. 28th Int. Symp. Model., Anal., Simul. Comput. Telecommun. Syst. (MASCOTS)*, Nov. 2020, pp. 1–5.

[5] M. Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M. Sharique, S. Seifert, S. Vishnoi, D. Booss, T. Peh, I. Schreter, W. Thesing, M. Wagle, and T. Willhalm, "Sap hana adoption of non-volatile memory," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1754–1765, Aug. 2017, doi: 10.14778/3137765.3137780.

[6] *Google Cloud™: Google Cloud: Intel Optane Dc Persistent Memory*. Accessed: Jul. 11, 2022. [Online]. Available: https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory

[7] *Frontier™: Advancing Computing and Data Capabilities for Scientific Discovery*. Accessed: Jun. 28, 2022. [Online]. Available: https://www.olcf.ornl.gov/frontier/

[8] S. Jamil, A. Khan, B. Burastaller, and Y. Kim, "Towards scalable manycore-aware persistent B+− trees for efficient indexing in cloud environments," in *Proc. IEEE Int. Conf. Autonomic Comput. Self-Organizing Syst. Companion (ACSOS-C)*, Sep. 2021, pp. 44–49.

[9] J.-H. Kim, Y. Kim, S. Jamil, C.-G. Lee, and S. Park, "Parallelizing shared file I/O operations of NVM file system for manycore servers," *IEEE Access*, vol. 9, pp. 24570–24585, 2021.

[10] S. Chen and Q. Jin, "Persistent B+−trees in non-volatile main memory," *Proc. VLDB Endowment*, vol. 8, no. 7, pp. 786–797, Feb. 2015.

[11] 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 *Proc. Int. Conf. Manage. Data*. New York, NY, USA: Association for Computing Machinery, Jun. 2016, pp. 371–386, doi: 10.1145/2882903.2915251.

[12] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent B+−tree," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 187–200.

[13] 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.

[14] S. Cho, W. Kim, S. Oh, C. Kim, K. Koh, and B. Nam, "Failure-atomic byte-addressable R-tree for persistent memory," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 601–614, Mar. 2021.

[15] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: Association for Computing Machinery, 1990, pp. 322–331, doi: 10.1145/93597.98741.

[16] N. Beckmann and B. Seeger, "A revised R*-tree in comparison with related index structures," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2009, pp. 799–812, doi: 10.1145/1559845.1559929.

[17] M. L. Scott, *Shared-Memory ynchronization* (Synthesis Lectures on Computer Architecture). San Rafael, CA, USA: Morgan & Claypool, 2013, doi: 10.2200/S00499ED1V01Y201304CAC023.

[18] D. Dice, V. J. Marathe, and N. Shavit, "Flat-combining NUMA locks," in *Proc. 23rd ACM Symp. Parallelism Algorithms architectures (SPAA)*, New York, NY, USA: Association for Computing Machinery, 2011, pp. 65–74.

[19] M. Chabbi, M. Fagan, and J. Mellor-Crummey, "High performance locks for multi-level NUMA systems," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 215–226, Dec. 2015, doi: 10.1145/2858788.2688503.

[20] A. Kogan and M. Herlihy, "The future(S) of shared data structures," in *Proc. ACM Symp. Princ. Distrib. Comput. (PODC)*, 2014, pp. 30–39.

[21] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, Jun. 1984.

[22] J. Sharman, "Exploring tradeoffs in parallel implementations of C++ using futures," M.S. thesis, Rice Univ., 2017. [Online]. Available: https://hdl.handle.net/1911/105498

[23] R. H. Halstead, "MULTILISP: A language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 4, pp. 501–538, Oct. 1985, doi: 10.1145/4472.4478.

[24] Q. Wang, Y. Lu, J. Li, and J. Shu, "Nap: A black-box approach to NUMA-aware persistent memory indexes," in *Proc. 15th USENIX Symp. Operating Syst. Design Implement. (OSDI)*. Berkeley, CA, USA: USENIX Association, Jul. 2021, pp. 93–111. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/wang-qing

[25] B. Gred, "Linux performance analysis and tools," Joyent, San Francisco, CA, USA, Tech. Rep., Feb. 2013. Accessed: Oct. 30, 2021. [Online]. Available: https://brendangregg.com/perf.html

[26] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank, "A persistent lock-free queue for non-volatile memory," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Feb. 2018, pp. 28–40.

[27] M. Herlihy and N. Shavit, *The Art Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann, 2012.
[28] P. Ramalhete and A. Correia, "Brief announcement: Hazard eras–non-blocking memory reclamation," in *Proc. 29th ACM Symp. Parallelism Algorithms Archit.*, Jul. 2017, pp. 367–369, doi: 10.1145/3087556.3087588.
[29] N. Cohen and E. Petrank, "Efficient memory management for lock-free data structures with optimistic access," in *Proc. 27th ACM Symp. Parallelism Algorithms Archit.*, Jun. 2015, pp. 254–263, doi: 10.1145/2755573.2755579.
[30] *Persistent Memory Development Kit*. Accessed: Dec. 1, 2021. [Online]. Available: https://pmem.io/vmem/libvmmalloc/
[31] *Numactl*. Accessed: Apr. 6, 2021. [Online]. Available: https://linux.die.net/man/8/numactl
[32] M. Lu and J. Zhixi Fang, "A solution of the cache ping-pong problem in multiprocessor systems," *J. Parallel Distrib. Comput.*, vol. 16, no. 2, pp. 158–171, Oct. 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/074373159290030Q
[33] S. Berchtold, D. A. Keim, and H. P. Kriegel, "The X-tree: An index structure for high-dimensional data," in *Proc. Int. Conf. Very Large Data Bases*. Burlington, MA, USA: Morgan Kaufmann, 1996, pp. 28–39.
[34] D. Greene, "An implementation and performance analysis of spatial data access methods," in *Proc. 5th Int. Conf. Data Eng.*, 1989, pp. 606–615.
[35] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: Association for Computing Machinery, 1984, pp. 47–57, doi: 10.1145/602259.602266.
[36] Intel. *Compute Express Link*. Accessed: Oct. 1, 2022. [Online]. Available: https://www.computeexpresslink.org,note
[37] Intel. *Intel Winding Down Its Optane Memory Business*. Accessed: Oct. 1, 2022. [Online]. Available: https://www.forbes.com/sites/tomcoughlin/2022/07/28/intel-winding-down-its-optane-memory-business/?sh=1b9c13ec45b8
[38] C.-G. Lee, S. Noh, H. Kang, S. Hwang, and Y. Kim, "Concurrent file metadata structure using readers-writer lock," in *Proc. 36th Annu. ACM Symp. Appl. Comput.* New York, NY, USA: Association for Computing Machinery, Mar. 2021, pp. 1172–1181, doi: 10.1145/3412841.3441992.
[39] I. B. Peng, M. B. Gokhale, and E. W. Green, "System evaluation of the Intel optane byte-addressable NVM," in *Proc. Int. Symp. Memory Syst.*, Sep. 2019, pp. 304–315, doi: 10.1145/3357526.3357568.
[40] B. Daase, L. J. Bollmeier, L. Benson, and T. Rabl, "Maximizing persistent memory bandwidth utilization for OLAP workloads," in *Proc. Int. Conf. Manage. Data*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 339–351, doi: 10.1145/3448016.3457292.
[41] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*. Santa Clara, CA, USA: USENIX Association, Feb. 2016, pp. 323–338. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

ABDUL SALAM received the B.E. degree in computer systems engineering from the Mehran University of Engineering and Technology (MUET), Jamshoro, Pakistan, in 2017. He is currently pursuing the M.S. degree with the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. He is currently a member of the Distributed Computing and Operating Systems Laboratory, Department of Computer Science and Engineering, Sogang University. His research interests include persistent memory and index data structures for big data processing systems.

SAFDAR JAMIL received the B.E. degree in computer systems engineering from the Mehran University of Engineering and Technology (MUET), Jamshoro, Pakistan, in 2017. He is currently pursuing the integrated M.S. and Ph.D degrees in the Department of Computer Science and Engineering, Sogang University, Seoul, South Korea. He is currently a member of the Distributed Computing and Operating Systems Laboratory, Department of Computer Science and Engineering, Sogang University. His research interests include scalable indexing data structures and algorithms, NoSQL database, data deduplication, and high performance computing.

SUNGWON JUNG received the B.S. degree in computer science from Sogang University, Seoul, South Korea, in 1988, and the M.S. and Ph.D. degrees in computer science from Michigan State University, East Lansing, MI, USA, in 1990 and 1995, respectively. He is currently a Professor with the Computer Science and Engineering Department, Sogang University. His research interests include data mining, spatial and temporal databases, blockchain databases, and multimedia databases.

SUNG-SOON PARK received the B.S. degree in computer science from Hongik University, in 1984, the master's degree in computer science and statistics from Seoul National University, in 1987, and the Ph.D. degree in computer science from Korea University, in 1994. He worked as a Lecturer (full-time) at Korea Air Force Academy, from 1988 to 1990. He also worked as a Postdoctoral Researcher at Northwestern University, from 1997 to 1998. He is currently a Professor with the Department of Computer Science and Engineering, Anyang University, and also the CEO of Gluesys Company Ltd. His research interests include network storage systems and cloud computing.

YOUNGJAE KIM (Member, IEEE) received the B.S. degree in computer science from Sogang University, Seoul, South Korea, in 2001, the M.S. degree in computer science from KAIST, in 2003, and the Ph.D. degree in computer science and engineering from Pennsylvania State University, University Park, PA, USA, in 2009. He is currently an Associate Professor with the Department of Computer Science and Engineering, Sogang University. Before joining Sogang University, he was a Research and Development Staff Scientist at the Oak Ridge National Laboratory, U.S. Department of Energy, from 2009 to 2015, and an Assistant Professor at Ajou University, Suwon, South Korea, from 2015 to 2016. His research interests include operating systems, file and storage systems, database systems, parallel and distributed systems, and computer systems security.