Cluster Computing
https://doi.org/10.1007/s10586-022-03766-1

Scalable NUMA-aware persistent B$^+$-tree for non-volatile memory devices

Safdar Jamil[1] · Abdul Salam[1] · Awais Khan[3] · Bernd Burgstaller[2] · Sung-Soon Park[4] · Youngjae Kim[1]

Received: 10 March 2022 / Revised: 18 July 2022 / Accepted: 19 September 2022

Abstract
Emerging manycore servers with Intel DC persistent memory (DCPM) are equipped with hundreds of CPU cores on multiple CPU sockets. Such servers are designed to guarantee high performance and scalability. Several recent studies proposed persistent fault-tolerant indexes for DCPM. Fast & Fair (F&F) is the state-of-the-art concurrent variant of the B$^+$-tree for DCPM. However, its adoption on manycore servers is hampered by scalability limitations due to lengthy, lock-based synchronization including structure modification operations. The lack of NUMA awareness induces further performance overhead from remote memory accesses. In this paper, we propose F^3-tree, a concurrent, NUMA-aware and persistent future-based B$^+$-tree for DCPM servers. F^3-tree relies on per-thread local future objects and a global B$^+$-tree. To introduce NUMA awareness and minimize remote memory accesses, F^3-tree adopts per-socket dedicated asynchronous evaluation threads to checkpoint future objects to the global B$^+$-tree. F^3-tree employs an in-memory hash table to mitigate the read overhead of key searches over the future objects. We implemented F^3-tree atop F&F and evaluated its performance on Linux using both synthetic and realistic workloads. Our evaluation shows that F^3-tree outperforms F&F on average by 3.4× and 5× without and with NUMA awareness, respectively.

Keywords Indexing data structures · B$^+$-tree · NUMA-aware architecture · Performance scalability · Persistent memory

A preliminary version of this article [Jamil et al., in Proceedings of the 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)] was presented at the 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), Washington DC, USA, September, 2021.

✉ Youngjae Kim
youkim@sogang.ac.kr

Safdar Jamil
safdar@sogang.ac.kr

Abdul Salam
abdulsalam@sogang.ac.kr

Awais Khan
bburg@yonsei.ac.kr

Bernd Burgstaller
khana@ornl.gov

Sung-Soon Park
sspark@gluesys.com

[1] Department of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea

[2] Oak Ridge National Laboratory, Oak Ridge, TN, USA

[3] Department of Computer Science, Yonsei University, Seoul, Republic of Korea

[4] GlueSys & Anyang University, Seoul, Republic of Korea

Published online: 17 November 2022

1 Introduction

Big data and cloud environments [1] are facing unprecedented challenges due to the enormous data and the number of applications that must be handled. Therefore, many cloud providers have deployed multi-socket manycore machines[1] to provide high performance and reduce the total cost of ownership (TCO) by consolidating multiple users within a single server. However, the recent inclusion of Intel DCPM in manycore machines introduces persistency at memory level [2, 3] but severely degrades application scalability. Manycore machines constitute NUMA architectures that provide high memory bandwidth at the cost of irregular memory access latencies (called the NUMA effect) [4–6]. DCPM aggravates NUMA effects because of its limited bandwidth and higher access latency compared to DRAM [7].

Applications running on cloud services include databases and file systems that manage user-generated data [8]. These applications rely heavily on index data structures such as B-trees, hash tables, radix trees, and R-trees for fast data access. Several indexing methods have been proposed for DCPM [9–12]. The B^+-tree is one of the most popular index data structures used in databases and file systems. Few studies used B^+-trees on DCPM [10, 11, 13], but none of the prior work investigated scalability on manycore machines.

F&F [11] is the state-of-the-art concurrent variant of the B^+-tree studied on DCPM. However, its adoption on manycore machines is hampered by scalability limitations. The write operation in F&F needs to obtain a lock to ensure mutual exclusion, which becomes a point of contention when multiple threads attempt to access a B^+-tree node. Structural modification operations (SMOs) such as node splitting and merging increase contention when a thread triggers a chain of SMOs from a leaf to the root of the B^+-tree. This chain requires the acquisition of per-node locks from the leaf to the root node, which is generally highly contended. Lock optimization techniques such as MCS [14], FC-MCS [15], and HMCS [16] cannot solve the inherent scalability limitations of B^+-trees (for details we refer to Sect. 3.1).

DCPM-based data structures such as F&F have not been designed for NUMA architectures and are therefore highly susceptible to performance degradation from NUMA effects. NUMA effects are further amplified because persistent data structures rely on pairs of `FLUSH+FENCE` instructions to write back data from the cache (volatile domain) to the DCPM (persistent domain). This eviction of cache lines has a profound effect on the cache coherence overhead: with *volatile* data structures, a cache line of a shared data item that is cached on one core can be directly served from that core's cache to another core that requests the data item. But with *persistent* shared data, modifications must be followed by `FLUSH+FENCE` to persist the modified data to the DCPM, and the next request (from the same or another core) is then required to fetch the modified data from the DCPM. Thereby the traffic to and from the DCPM and hence the associated NUMA effects from non-local memory accesses increase, thus, limiting the scalability of the DCPM-based data structures.

To address the scalability issue of B^+-trees, we employ scalable future objects (FOs) [17]. FOs have been proposed to improve the performance of shared data structures. FOs are data objects that promise to deliver the results of an operation once the results become available. The operations represented by FOs are applied to the shared data structure when their *evaluate* method is called by the respective thread. FOs can be maintained thread-locally, which we refer as per-thread local future objects (PTFOs). As implied in the name, with PTFOs each thread is responsible for the allocation and evaluation of its own FOs. Several approaches can be adopted to checkpoint FOs to a shared data structure. For instance, threads can accumulate pending future operations to process all PTFOs in a single batch operation on the shared data structure. Operations may cancel each other out before a batch operation takes effect. To achieve NUMA-awareness and to reduce the cross CPU node communication, NAP [7] extends the node replication technique [18] which has proven effective for DRAM-only data structures on manycore architectures. With the node replication technique, a replica of the data structure is maintained per CPU node. A shared log is adopted for lazy synchronization of the replicas. However, this approach suffers from a huge memory overhead incurred by the replicas.

Adopting PTFOs for indexing data structures such as B^+-trees can improve their insertion performance, but this comes with its own challenges:

1. Integrating DRAM-based PTFOs with a DCPM-based B^+-tree raises two consistency issues. First, the placement of PTFOs in DCPM requires a durability guarantee; Otherwise, after a crash, PTFOs may be in an inconsistent state, e.g., from missing pointers between PTFOs. Second, the *evaluate* method must incorporate operations from PTFOs to the shared B^+-tree in a crash-resilient manner. Otherwise, data loss may occur, leaving the B^+-tree in an inconsistent state.
2. PTFOs can severely degrade the read performance of a B^+-tree. The read operations have to traverse the PTFOs to search for updated keys, which incurs additional read overhead.

[1] We refer to multi-socket manycore machines as manycore machines hereafter.

3. Moreover, FOs and the *evaluate* method are not NUMA-aware by nature and incur considerable cross-CPU communication overhead when executed on a manycore machine (for details we refer to Sect. 3.2). The main culprit for the remote memory accesses is the *evaluate* method which has to access the PTFOs across CPU nodes to checkpoint to the shared data structure.

To address the aforementioned challenges, we propose F^3-tree, a NUMA-aware concurrent persistent B^+-tree for DCPM-based manycore platforms. Our F^3-tree design relies on two key properties, namely (1) DCPM-based PTFOs, and (2) a shared, global B^+-tree based on F&F [11]. This approach is inspired by the producer-consumer design principle where application threads are only allowed to update PTFOs (as producers), while dedicated asynchronous threads are privileged to perform update operations on the global tree (as consumers). We convert DRAM-based PTFOs to persistent PTFOs and rely on durable linearizability as the correctness condition to guarantee crash consistency. Although our PTFOs improve the scalability of the DCPM-based B^+-tree, scalability is still affected by NUMA-effects. To achieve NUMA-awareness, we bind the memory allocation of PTFOs to the local memory of a CPU node, and we associate asynchronous *evaluate* threads with CPU nodes to restrict checkpoint operations to local PTFOs.

This work makes the following specific contributions:

- We evaluate the state-of-the-art F&F B^+-tree on DCPM-based manycore machines and identify scalability limitations such as increased lock contention when multiple threads wait to acquire the lock of a particular leaf node.
- We propose F^3-tree which adopts scalable future objects to overcome the lock contention limitation of F&F for application threads. We design a per-thread linked list data structure that accommodates the insert operations from the application and evaluates (or checkpoints) those operations to the global B^+-tree through dedicated asynchronous threads.
- We further improve the F^3-tree by incorporating NUMA-awareness for our per-thread linked list and bind the dedicated evaluate threads to only checkpoint the CPU node local linked list to the global B^+-tree.
- To achieve correctness and crash resiliency for the thread-local linked list, we adopt durable linearizability.
- We adopt a hash table for the per-thread local linked list to avoid the linear traversal of linked list nodes.
- We evaluate the scalability of F^3-tree against the state-of-the-art F&F using a set of synthetic benchmarks on a 40-core platform. We employ one million 8-byte key-value pairs with sequential and random key distributions. Our evaluation shows that F^3-tree outperforms F&F on average by a factor of 3.4× and 5× without and with NUMA-awareness, respectively. Our NUMA-aware node-local memory allocation and CPU-bound asynchronous *evaluate* threads reduce cross CPU node memory accesses by 50%.

The remainder of this paper is organized as follows. Section 2 introduces the relevant background on F&F and the state-of-the-art NUMA-awareness techniques. Section 3 presents our investigations on the scalability and NUMA awareness of DCPM-based data structures. Section 4 presents the design and implementation details of our proposed system. Section 5 provides the experimental results. Section 6 discusses the related work and Sect. 7 draws our conclusions.

2 Background

In this section, we present the required background on the F&F B^+-tree, followed by a detailed discussion of state-of-the-art NUMA-awareness techniques adopted for DRAM and DCPM-based index data structures.

2.1 Fast & Fair: B^+-tree

A recent study, F&F, proposed a DCPM-based durable and concurrent B^+-tree that provides lock-free reads [11]. F&F avoids the logging overhead by transforming a B^+-tree to another consistent state or a transient inconsistent state that readers can tolerate. Readers detect and tolerate inconsistencies such as duplicated elements in a sorted list. Writers hold a lock for mutual exclusion. When write operations detect inconsistencies, they attempt to fix them.

F&F is composed of two algorithms, failure atomic shift (Fast) and failure atomic in-place rebalance (Fair). Fast is used to insert the keys within a node of the B^+-tree by performing atomic shift operations to maintain the sorted order of the keys. Because a B^+-tree node is an array of entries, the shift operation is a sequence of load and store instructions in cascading order, and it maintains the total store order. This helps in avoiding excessive use of `FLUSH+FENCE` instructions, as the updated entries within the B^+-tree node can be flushed together. Maintaining a consistent view of the tree-based indexing data structure during the structural modification operations is challenging with respect to logging. Logging constitutes additional overhead as it duplicates the number of pages, increases the write traffic, and blocks the concurrent access to tree nodes. The Fair algorithm avoids the use of logging by

maintaining sibling pointers in the B$^+$-tree nodes, and by creating a B-link tree [19].

2.2 NUMA-aware data structures

There have been several attempts to achieve NUMA awareness for DRAM-based data structures [16, 18, 20–22]. Node replication (NR [18]) is a state-of-the-art approach with ideas from distributed computing. NR replicates the data structure across multiple CPU nodes and performs lazy synchronization using a single shared log. NrOS [22] increased the scalability of NR by using multiple shared logs and multiple per-node combiners. A recent study, NAP [7], explores the NR technique for DCPM-based data structures and reports two major limitations. First, NR does not consider failure atomicity, which is crucial for DCPM. Second, NR suffers from severe space overhead due to per-node replicas and shared global logs. NAP, therefore, proposes a block box approach to convert DCPM indexes into NUMA-aware index structures. NAP introduces a NUMA-aware layer (NAL) that contains hot items with a non-synchronized partial and crash-consistent per-node view in DCPM for insert, update and delete operations. It maintains a global and volatile view in DRAM to serve lookup operations. However, NAP suffers from space overhead because it maintains the hot items in the per-node partial and crash-consistent view.

3 Motivation

In this section, we first investigate the scalability limitations of F&F on manycore machines and elaborate on them in detail. We then dive into the necessity and details of NUMA-awareness on manycore platforms.

3.1 Scalability limitations of F&F

F&F faces scalability limitations with highly parallel write scenarios on manycore machines. For instance, the critical section of F&F is composed of several sub-operations, i.e., linear search, lock acquisition, shift, and SMOs.

Figure 1 depicts a concurrent write operation in F&F, where two threads, T_1 and T_2, perform an insert operation. Both threads look up the candidate node for key insertion, as shown in step ① of Fig. 1. Note that both threads select node N1 as their candidate for key insertion. However, F&F employs locks for mutual exclusion. Hence, only one thread can acquire the lock and proceed, while the other thread has to wait. In the scenario depicted in step ② of Fig. 1, thread T_1 wins and acquires the lock of node N1, while thread T_2 is now blocked on that same lock. As thread T_1 proceeds with the insert operation, it checks the capacity of candidate node N1 and triggers an SMO because it finds the capacity of the candidate node to be exhausted. During the SMO, thread T_1 allocates a new node (N5), migrates half of the entries from node N1 to N5, and inserts the key into the corresponding node, as depicted in steps ③ and ④. In step ⑤, thread T_1 has relinquished the lock of node N1 and acquired the lock of the parent node PN to update its links accordingly. Meanwhile, thread T_2 is finally able to acquire the lock of node N1, only to find out that node N1 has been split, and that the key needs to be inserted in the new candidate node, i.e., node N5. So, in step ⑥, thread T_2 relinquishes the lock of node N1 and acquires the lock of the new candidate node N5 to perform its insert operation.

The previous example shows that F&F's blocking synchronization in conjunction with long-running SMOs drastically limits its scalability on manycore machines, where hundreds of application threads are contending to perform insert operations. To address the aforementioned

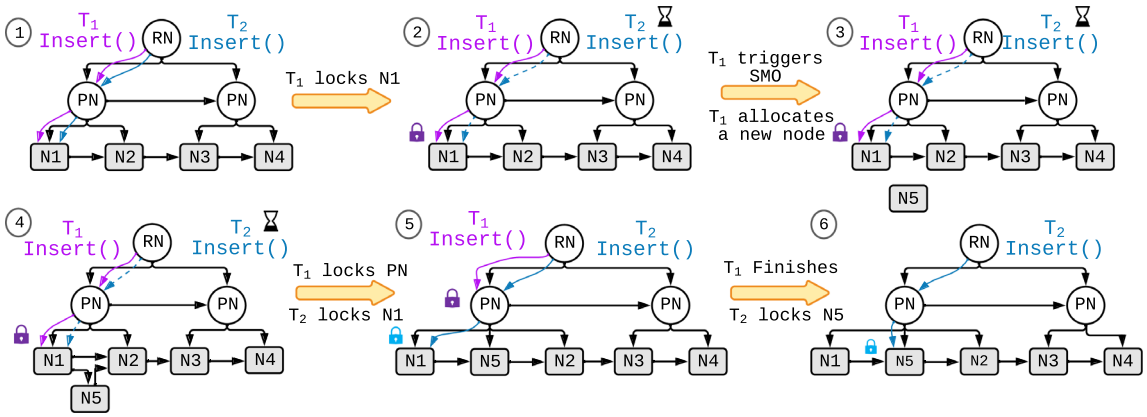


Fig. 1 A multi-thread insertion use case to demonstrate the scalability limitation in F&F [11]

limitation of F&F, we propose F^3-tree, a concurrent persistent future-based B$^+$-tree.

3.2 The lack of NUMA awareness with state-of-the-art DCPM-based data structures

F&F has not been designed for NUMA architectures and hence suffers from NUMA-effects on contemporary manycore platforms, where each CPU node has attached its own local memory [3, 7, 23]. The NUMA-effect refers to the differences in memory latencies between the local and remote memory of a CPU node. These irregular memory latencies are mainly caused by cross CPU node communication, cache coherence protocol overhead, and remote memory accesses.

To investigate these effects, we perform several experiments on F^3-tree [1] for an increasing number of threads within and across the CPU nodes of a system. Our evaluation platform consists of 4 CPU nodes with 10 cores per CPU (for the details of the experimental setup we refer to Sect. 5.1). To emulate DCPM for manycore machines, we employ two memory allocation policies using the *numactl* tool [24] to cover the *interleaved* and *non-interleaved* modes of DCPM [3, 25]. In interleaved mode, the pages of a memory allocation are assigned round-robin, i.e., interleaved, across all the memory devices of a system. Interleaving is conducted uniformly and irrespective of the requesting CPU core. Thus in interleaved mode, each thread of an application is able to allocate memory pages across all memory devices in the system. In contrast, the non-interleaved mode restricts CPU cores—and the application threads executing on them—to allocate memory from the local memory of the CPU.

Figure 2 depicts the percentages of non-local memory accesses for three configurations of F^3-tree executed on our evaluation platform with 10, 20, 30, and 40 threads: Configuration `F 3-tree` does not pin the producer and consumer threads to physical CPU cores and uses the default memory allocation policy, i.e., local allocation. Configurations `F 3-tree(I)` and `F 3-tree(NI)` pin the threads to physical CPU cores and use the interleaved and non-interleaved policies, respectively. Our pinning scheme populates CPU nodes consecutively, from the first node onward. For all configurations, we limit the number of *evaluate* (consumer) threads to 2, regardless of the number of producers.

It can be observed from Fig. 2 that the `F 3-tree` configuration performs remote memory accesses even for 10 threads, which is within the NUMA boundary of our evaluation platform. This is due to the fact that with the `F 3-tree` configuration, threads are not pinned to physical CPU cores and hence the OS schedules threads across all CPUs. The default (local) memory allocation policy allocates the PTFOs from the thread's local memory, which will be the memory on the CPU the respective thread is executing. However, configurations `F 3-tree(I)` and `F 3-tree(NI)` with 10 threads exhibit little cross socket communication because threads are pinned to the physical CPU cores of the first CPU and all threads perform local memory allocations for the PTFOs.

On the other hand, remote memory accesses increase considerably for 20, 30, and 40 threads because these thread configurations exceed the NUMA boundary of our evaluation platform. The remote memory accesses are mainly due to the checkpointing of PTFOs to the global B$^+$-tree, conducted by the *evaluate* threads. Configuration `F 3-tree(NI)` suffers largely from remote memory accesses when the number of CPU nodes increases because the *evaluate* threads are always pinned to the first CPU node and to checkpoint the PTFOs, *evaluate* threads have to perform remote memory accesses. Although configuration `F 3-tree(I)` has the *evaluate* threads pinned to the first CPU node, it nevertheless employs the interleaved memory allocation policy which to some degree mitigates

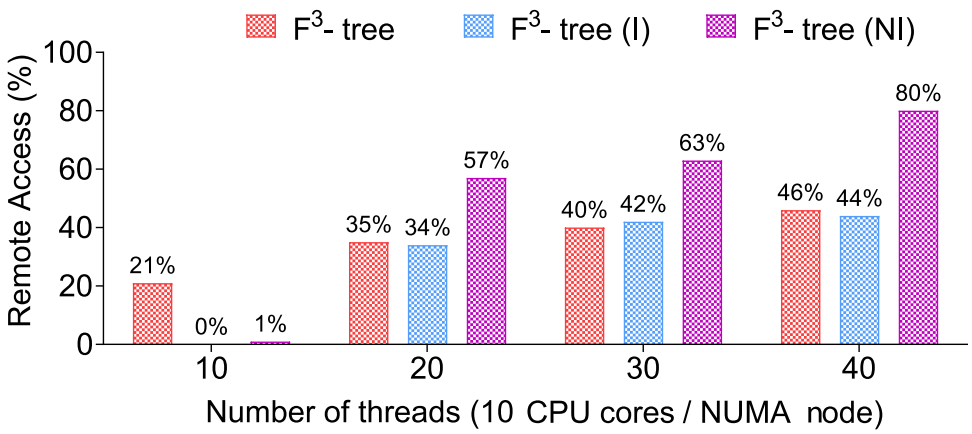


Fig. 2 F^3-tree cross socket memory accesses (lower is better) on the 4-socket evaluation platform

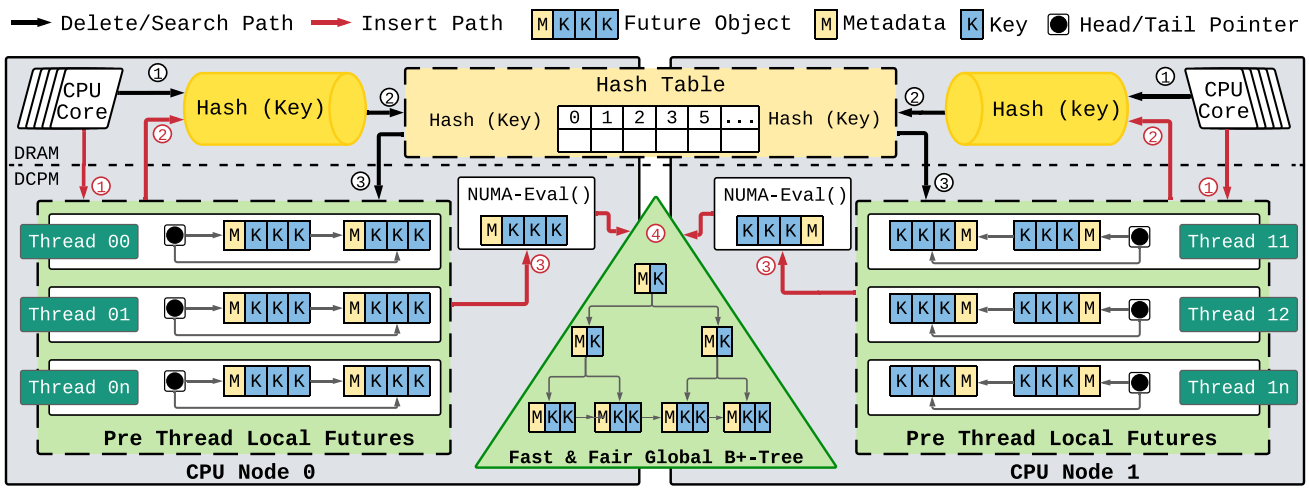


Fig. 3 Design overview and operation flow of proposed NUMA-aware F^3-tree. The red arrows show the insert operation flow while black arrows present delete and search operations

the lack of NUMA awareness. Thus, `F 3-tree(I)` experiences fewer remote memory accesses than `F 3-tree(NI)`. However, the general message from Fig. 2 is clear: irrespective of the particular configuration, this DCPM-based data structure is severely affected by its lack of NUMA awareness, as exemplified by the large percentage of remote memory accesses.

4 F^3-tree: design and implementation

In this section, we present our design overview, operational flow, and the space and time complexity of our proposed design.

4.1 Design overview

The key design goals of F^3-tree include scalability, NUMA-awareness, crash resilience, and maintaining the read performance. We achieve the first three goals by extending future objects [17] and the last goal by using an in-memory hash table, as shown in Fig. 3. Our design is inspired by the asynchronous computation design principle, i.e., the producer-consumer model. Producers only update the FOs while consumers perform the operations on the shared data structure. FOs can be allocated thread-locally and by binding their memory allocations to the local memory nodes, we achieve scalability and NUMA-awareness. Converting DRAM-based FOs to be persistent requires a durability guarantee in case of a power failure or full system crash. We achieve crash resilience by relying on durable linearizability as the correctness condition. To minimize the search overhead for keys residing in thread-local FOs, we employ an in-memory hash table.

4.2 A scalable and NUMA-aware F^3-tree

Figure 3 shows the design overview of our proposed F^3-tree. Our design is comprised of three major components: per-thread-local future objects (PTFOs), a shared global B$^+$-tree, and a DRAM-based hash table. PTFOs act as thread-local buffers and allow application threads (as producers) to perform write operations locally. We maintain FOs as thread-local doubly linked-lists.[2] Because PTFOs are thread-local, application threads do not require any locking mechanism for mutual exclusion and thereby achieve higher scalability.

Concurrently to the producer's thread-local work, PTFOs are checkpointed to the shared global B$^+$-tree through the *evaluate* function, where a dedicated work pool of asynchronous threads serves as consumers. The global B$^+$-tree is the base design of F&F [11] with no modifications. To achieve NUMA awareness, we bind the memory allocations of the PTFOs to the local memory of CPU nodes of the application threads, as shown with the per CPU node PTFOs in Fig. 3. We bind the asynchronous *evaluate* threads to only perform checkpointing for the local CPU node PTFOs and thereby avoid accessing cross CPU node PTFOs, as depicted by the per CPU node `NUMA-Eval()` method in Fig. 3.

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 PTFOs of the application threads from remote CPU nodes. With controlled memory allocation and CPU-bound asynchronous *evaluate* threads, our design reduces the amount of remote memory

[2] For simplicity, from here onward we refer to thread-local doubly-linked lists of FOs as PTFOs.

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 global B$^+$-tree, which suffers from inherent scalability limitations (as discussed in Sect. 3.1). We did not bound the memory allocations of the global B$^+$-tree as we opted not to modify the base design of F&F.

Figure 3 shows how we achieve NUMA awareness in our design for a two-socket architecture (CPU Node 0 and CPU Node 1). The circled numbers represent the steps performed for write operations by application threads and the asynchronous *evaluate* threads (the detailed operational flow follows in Sect. 4.3). In step ①, threads from CPU Node 0 and Node 1 perform write operations to their corresponding PTFOs and once the data is written durably, entries are updated in the hash table in step ②. Step ③ pertains to the `NUMA-Eval()` method where dedicated asynchronous *evaluate* threads checkpoint the entries from the local CPU nodes' PTFOs to the global B$^+$-tree.

4.3 Operation flow

We show the operation flow of F^3-tree for the insert, search, and delete operations in Fig. 3. The red encircled steps show the insert operation flow. The insert operation will first insert the key-value pair into the thread-local buffers, i.e., into the thread's PTFOs. Once the key is successfully written, it is pushed to the hash table. Algorithm 1 shows the pseudo code of the insert operation at PFTOs from Line 4 to Line 20. Note that the delete and search operations use a different path compared to insert. Both delete and search perform a hierarchical key lookup, i.e., (i) a lookup in the hash table; If the key is not found, then (ii) a lookup is performed in the corresponding thread-local linked list followed by (iii) the global B$^+$-tree. Note that the lookup complexity increases as the search operation progresses to higher hierarchies because of the increased search space.

The *evaluate* operation works as follows: each *evaluate* thread is responsible for a particular thread-local linked list and it checkpoints each linked list's PTFOs to the global B$^+$-tree, once it meets a given threshold. We define two thresholds for the checkpoint of PTFOs, (i) time-based and (ii) based on the number of KV pairs. A PTFO is accessed using the tail pointer of the linked list by the *evaluate* threads when either one of the thresholds is met. Once a PTFO is checkpointed, the tail pointer is updated atomically to the previous node. Lines 21 to 31 of Algorithm 1 show the pseudo code of the *evaluate* operation.

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

During the recovery phase after a system crash, the asynchronous *evaluate* threads will checkpoint the PTFOs from before the crash to the global tree. In addition, for application threads, new PTFOs and a hash table will be allocated in DCPM and DRAM, respectively. This will allow the application to resume its execution without waiting for the recovery process to finish.

Algorithm 1 Future Insert and Evaluate Methods

1: KV : input key & value pair
2: TID : Thread ID for PTFO
3: $PTFO_{TID}$: Thread-local future linked list of Thread TID

4: **procedure** INSERT_FO(KV, TID)
5: Identify the corresponding PTFO by TID
6: **if** FO_1 not full **then**
7: Insert Key & value to FO_1 of $PTFO_{TID}$ ▷ FO_1 represents the 2^{nd} node of $PTFO_{TID}$
8: Update metadata
9: **else**
10: Allocate new FO
11: Update Next node pointer of Head node
12: Update Next node pointer of new FO to point to previous FO_1 ▷ Now new FO becomes FO_1
13: Call $FLUS0_{with_FENCE}$
14: Insert Key and value to new FO
15: Update metadata
16: **end if**
17: **if** $SizeOf(FO_1) >= CACHELINE$ **then**
18: Call $FLUSH_{with_FENCE}$
19: **end if**
20: **end procedure**

21: **procedure** EVALUATE(TID)
22: Identify the corresponding $PTFO$ by TID
23: Traverse to the N^{th} FO of PFO_{TID}
24: **while** $HeadPTFO_{TID}! = TailPTFO_{TID}$ **do**
25: **while** $FO_{key\_count}! = 0$ **do**
26: Call $btree\_insert$ with KV from FO_N ▷ $btree\_insert$ is same as F&F
27: **end while**
28: Update $TailPTFO_{TID}$ to previous node of FO_N
29: De-allocate FO_N
30: **end while**
31: **end procedure**

4.4 Thread-local future objects

A future is a data object that promises to deliver the results of an operation when ready [17]. In this work, a future object constitutes an array of keys, an entry count, and the next and previous FO pointers, as depicted in Fig. 3. Every thread maintains its thread-local FOs stitched together by a doubly-linked list.

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

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

4.5 Durable linearizability (DL)

A concurrent data structure is linearizable if each operation takes effect in between the method's invocation and response [26]. With DCPM, a durability guarantee is additionally required because the data will be persistent and need to be crash resilient. A *durably linearizable* concurrent data structure satisfies the linearization property. In addition, after a full-system crash (i.e., all threads crash), the state of the data structure must reflect a consistent sub-history of operations that includes all operations completed by the time of the crash. We use the term *durability point* as the point in the execution history where an operation becomes durable, i.e., its effects are visible to other threads and persistent. After a durability point, if we execute the recovery mechanism, then the data structure will be in a consistent state. The durability order of execution can be elaborated in terms of durability points, where each durability point implies an order on the operations.

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

Figure 4b shows an example where a new FO is added to the thread-local linked list. Steps ② to ④ show the allocation of a new FO in the thread-local linked list. In this scenario, the DLP is achieved once the next-pointer of the newly allocated FO is updated atomically, followed by the `FLUSH+FENCE` instructions. We call the `FLUSH+FENCE` instruction pair right after updating the next pointer of the new FO so that if a crash happens after the DLP, we are

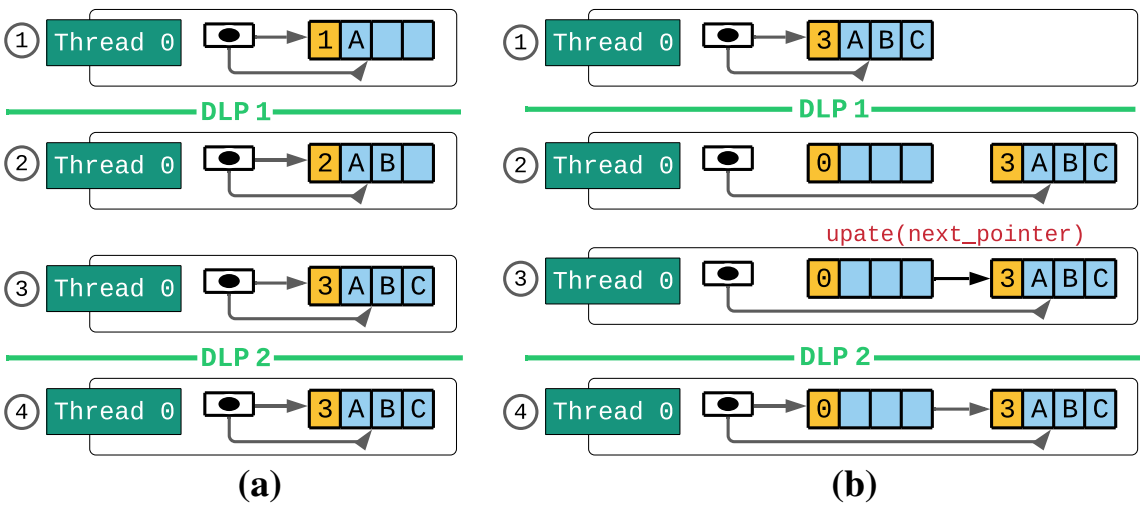


Fig. 4 Durable linearizability examples. DLP represents the DL point achieved. (a) shows a single FO achieving DLP while (b) shows a multi-FO DLP

able to access the new FO by a backward traversal. If a crash happens before DLP 2, our recovery mechanism will be able to achieve the consistent state of the thread-local linked list up to DLP 1. There is a potential memory leak that needs to be addressed if a crash happens in between steps ②and ③. Several mechanisms can be adopted for memory leaks such as hazard eras [27] and the optimistic access scheme [28].

4.6 Space and time complexity

The space overhead of our proposed design is similar to the traditional B$^+$-tree, i.e., $O(N)$, because a key is either in the PTFO or in the global B$^+$-tree. Also, the in-memory 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 PTFO entries at a particular point of time.

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

5 Evaluation

5.1 Testbed setup

We performed our experiments on a Linux machine (kernel version 5.4.0) equipped with 4 Intel Xeon(R) E5-4640 v2 CPUs @ 2.20GHz with 10 physical cores per node, 80MiB last-level cache, and 256GiB DDR3 DRAM. We enabled hyper-threading to increase threads for the scalability evaluation. We emulated the latency of Intel DCPM as presented in [3]. We implemented the F^3-tree atop F&F in C++11. All source code was compiled with g++ version 9.3.0 using optimization level `-O3`. We employed the numactl utility [24] to pin threads to CPU cores, populating CPU nodes consecutively from the first node onward. In our evaluation, we compared the following implementations:

- F&F: The baseline concurrent variant of the DCPM-based B$^+$-tree. We utilized the public implementation of F&F available at [29].
- F^3-tree: The variant of F^3-tree without NUMA-awareness [1]. We evaluated two different versions of F^3-tree without NUMA-awareness, i.e., key-based (F^3-K) and node-based (F^3-N). With F^3-K, the *evaluate* thread consumes the keys from the PTFOs in sequential order and checkpoints them one-by-one to the global B$^+$-tree. In contrast, F^3-N benefits from batching, i.e., a single PTFO consists of multiple keys and the *evaluate* thread checkpoints the entire FO to the global B$^+$-tree in a single operation.
- F^3-tree (N): The NUMA-aware variant of F^3-tree where we bound the memory allocations to the local CPU node and the *evaluate* threads to checkpoint per CPU node PTFOs.

Table 1 provides the details of the benchmarks used in our evaluation. We adopted the synthetic benchmark to evaluate our proposed solution which is the same from F&F approach and their work. Furthermore, the synthetic benchmark simulates the workload pattern of a standard benchmark, such as Yahoo Cloud Serving Benchmark [30]. It generates workloads that can be of either uniform distribution or random/skewed distribution.

5.2 Evaluation results of F^3-tree

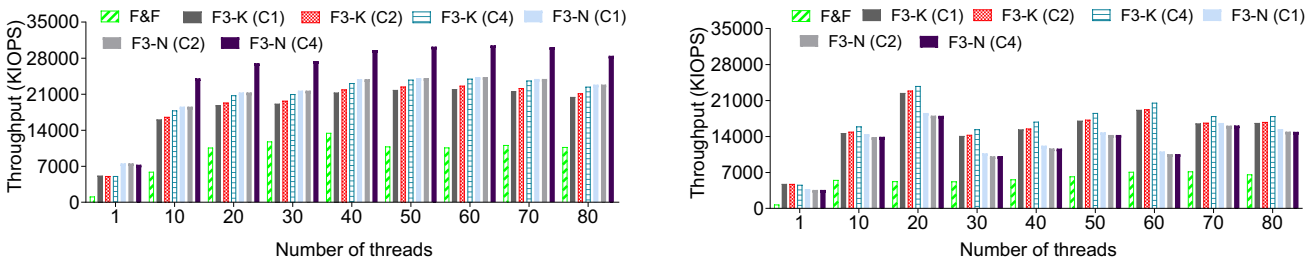
In this subsection, we present the evaluation results of F^3-tree using the synthetic benchmark with sequential and random key distributions, the effect on performance when varying the size of the PTFOs, and the evaluation of F^3-tree using a simulated realistic workload.

5.2.1 Sequential workload analysis

Figure 5a depicts the performance for sequential workloads. We observe that on average F^3-N outperforms F^3-K by a factor of 1.3×, and F&F by a factor of 3.3×. The reasons are manifold. First, F^3-N benefits from the sequential order of keys within the workload and does not

Table 1 Benchmark and workload description

Benchmark	Workload	Workload size
Synthetic	W1: write only	1 M 8B keys
	W2: read only	
Realistic	Composite	1 M write (8B keys)
		12.5 K Delete (8B keys)
		2 M Search Ops



(a) Sequential insert workload

(b) Random insert workload

Fig. 5 Scalability analysis of F^3-tree on the manycore machine. In F^3-K (C_i) and F^3-N (C_i), C_i represents the number of asynchronous *evaluate* threads. We used workload W1 of the synthetic benchmark from Table 1

explicitly perform sorting. Second, F^3-N checkpoints the whole PTFO to the global F&F tree, which leads to fewer shift operations and SMOs on the global tree. Third, F^3-N incurs less thread synchronization and communication overhead with foreground threads. Similarly, F^3-K outperforms the F&F tree on average by a factor of 2.4× due to its PTFO design. We 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 the sequential key order. Moreover, the workload is equally distributed among threads, reducing the contention for individual B$^+$-tree nodes.

Notably, we observed a scalable trend by all approaches for threads within a single CPU node. However, performance degrades with threads crossing the CPU-node boundary due to the overhead of remote memory accesses. Although the F^3-tree writes to PTFOs still the asynchronous threads read the PTFOs and checkpoint them to the global tree and thus suffer from remote memory accesses and performance saturation. We address this issue with F^3-tree(N), a NUMA-aware solution for B$^+$-trees (see Sect. 5.3.1). Figure 5a shows a similar trend for throughput when the number of consumer (*evaluate*) threads varies.

5.2.2 Random workload analysis

Figure 5b shows the results of the random workload. We observed that F^3-K outperforms F^3-N because of two major reasons. First, F^3-N traverses the whole global tree to check if the key-value pairs within the local future object overlap with the key-value pairs of any of the global tree nodes. Second, F^3-N performs a double shift operation, i.e., on the PTFO and the global tree.

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

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

5.2.3 Varying future objects

Figure 6 shows the performance impact of varying the PTFO size. We limit the number of future objects to 100, 500, and 1000, shown as F3-K (100), F3-K (500), and F3-K (1000). We limit the number of *evaluate* threads to only four and compared them with the performance of F&F with four threads only. We observe in Fig. 6 that the scalability of our proposed system is limited by the number of future objects. Meanwhile, the application threads (producers) spend most of their time waiting for the *evaluate* threads (consumers) to checkpoint the data from PTFOs to the global B$^+$-tree. Furthermore, we observe that F3-K (No Limit), where we do not limit the number of future objects, has the best overall performance. F3-K (1000) shows equivalent performance to F3-K (No Limit) for 40 threads because the number of PTFOs is less than the threshold of 1000 future objects, and so the producers are not blocked during the entire execution.

5.2.4 Realistic workload analysis

We simulate a realistic workload scenario where an application performs a mixed workload of read and write

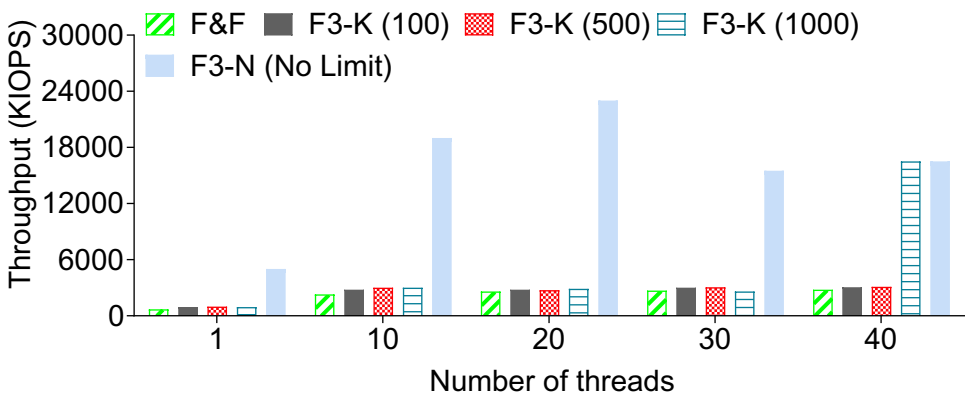


Fig. 6 PTFO size impact on performance with the synthetic benchmark using workload W1

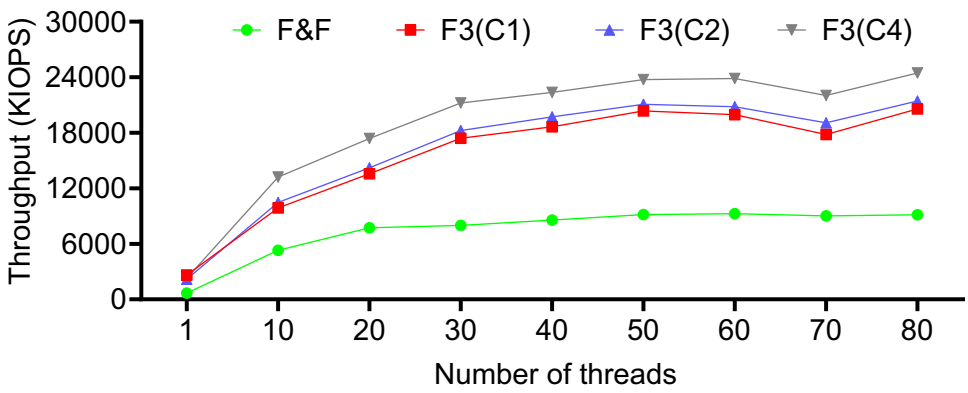


Fig. 7 Realistic workload analysis (synthetic benchmark with 2M search, 1M insert, and 12.5K delete operations). C_i shows the number of *evaluate* threads

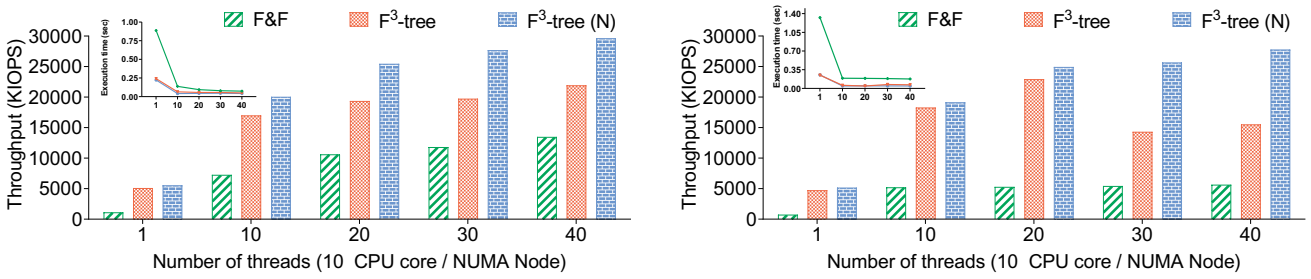
operations. For this experiment, we used a random key distribution and the key-based operation mode of the F^3-tree. During this experiment, each thread alternates between four insert queries, 16 search queries, and one delete query, as performed in F&F. Figure 7 shows that the F^3-tree outperforms F&F and gains a speedup of about 2.6× when the number of threads increases. This is due to high performing insert operations of the F^3-tree and the in-memory hash table for search operations. Supporting range queries for the F^3-tree is challenging because range queries fetch multiple key-value pairs. The range of keys in the F^3-tree can overlap between PTFOs and the global B^+-tree. Within PTFOs, range query performance degrades even after using the in-memory hash table because it does not support range queries.

5.3 Results of NUMA-aware F^3-tree

In this subsection, we present the evaluation results of F^3-tree (N) against F^3-tree. We present the results with sequential and random key distributions and show the reduced cross CPU node communication with F^3-tree (N).

5.3.1 Write performance

In this subsection, we present the evaluation results of our NUMA-aware F^3-tree in comparison to the F^3-tree without NUMA-awareness and F&F. We employ workload W1 of the synthetic benchmark from Table 1 unless stated otherwise. We only utilize the key-based approach for the *evaluate* threads to checkpoint the PTFOs. We limit the number of *evaluate* threads in F^3-tree to two whereas the number of F^3-tree (N) *evaluate* threads increases by two with each CPU node (i.e., we employ two *evaluate* threads per CPU node). We restrict the number of threads to the number of physical CPU cores. As F^3-tree (N) bound the *evaluate* threads to only checkpoint the local PTFOs and our system is based on four physical CPU nodes with 10 cores each, we performed our experiments for up to 40 threads only. Figure 8a depicts the performance for the sequential distribution of the keys. On average, F^3-tree (N) outperforms F^3-tree by 1.3× and F&F by 5×. The reasons are manifold. First, F^3-tree (N) does not allocate non-local memory (i.e., all allocations are conducted on the local CPU node), and the *evaluate* threads only checkpoint the local CPU nodes' PTFOs. This minimizes the cross CPU node communication and reduces the cache coherence overhead (cache ping-pong [31]) of shared data. Second,



(a) Sequential insert workload

(b) Random insert workload

Fig. 8 F^3-tree (N) performance comparison with base F^3-tree and F&F on the manycore machine. We limit the *evaluate* threads for F^3-tree to two and pin the threads to physical CPU cores only. The subplot shows the execution time

F^3-tree and F^3-tree (N) both benefit from the sequential order of keys within the workload and do not explicitly perform sorting.

Figure 8b shows the performance under a random distribution of the keys. F^3-tree (N) outperforms F^3-tree due to its local memory allocations and the CPU node-bound *evaluate* threads. F^3-tree performance drops once the threads cross the NUMA boundary due to high cross-CPU node communication overhead caused by the *evaluate* threads. However, with F^3-tree (N), the *evaluate* threads are only allowed to checkpoint their local PTFOs, which reduces the cross CPU node communication overhead and enables our F^3-tree (N) to scale beyond NUMA boundaries.

Furthermore, Fig. 8 shows the execution times of the F&F, F^3-tree and F^3-tree (N). It can be observed that F^3-tree (N) outperforms F&F by a significant margin while it manages to achieve faster execution times in comparison to F^3-tree.

Figure 9 depicts the percentages of remote memory accesses for F^3-tree (N). We compare the remote memory accesses with F^3-tree for an increasing number of consumer (*evaluate*) threads per CPU node. With F^3-tree (N), the number of *evaluate* threads increases by two with each CPU node, while with F^3-tree (N2) we limit the number of *evaluate* threads to two. Moreover, F^3-tree (NI) represents the non-interleaved memory allocation policy as explained in Sect. 3.2. It can be observed from Fig. 9 that F^3-tree (N) exhibits the lowest percentage of remote memory accesses due to its node-local memory allocation and *evaluate* threads. However, it still performs remote memory accesses due to the global B^+-tree as we did not modify the memory allocation scheme of the global B^+-tree. Overall, we can observe that 50% of the remote memory accesses are reduced in comparison to F^3-tree (NI).

5.3.2 Read performance

We perform experiments with read workloads to determine the overhead of PTFOs, as depicted in Fig. 10. We compare F&F with our proposed F^3-tree, which includes the in-memory hash table, F^3-tree without the in-memory hash table, and F^3-tree (N). We use the synthetic benchmark with workload W2 detailed in Table 1. For this experiment, we place 20% key-value pairs in PTFOs while 80% key-value pairs are placed in the global F&F tree. F^3-tree "No Hash" has the worst read performance because every read operation has to go over the PTFOs first and if the key

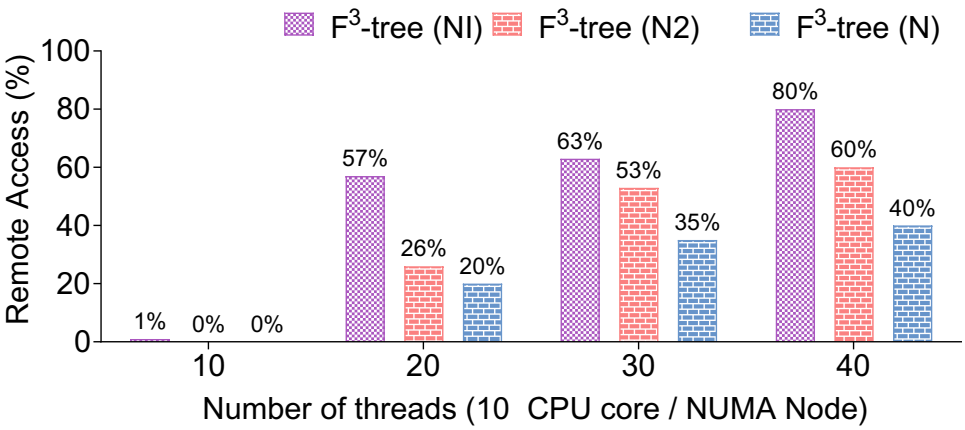


Fig. 9 Comparison of cross CPU node memory accesses between F^3-tree (NI) (non-interleaved memory allocation), F^3-tree (N), and F^3-tree (N2). F^3-tree (NI) and F^3-tree (N2) have two *evaluate* threads, whereas, in F^3-tree (N), the number of *evaluate* threads increases by the order of two

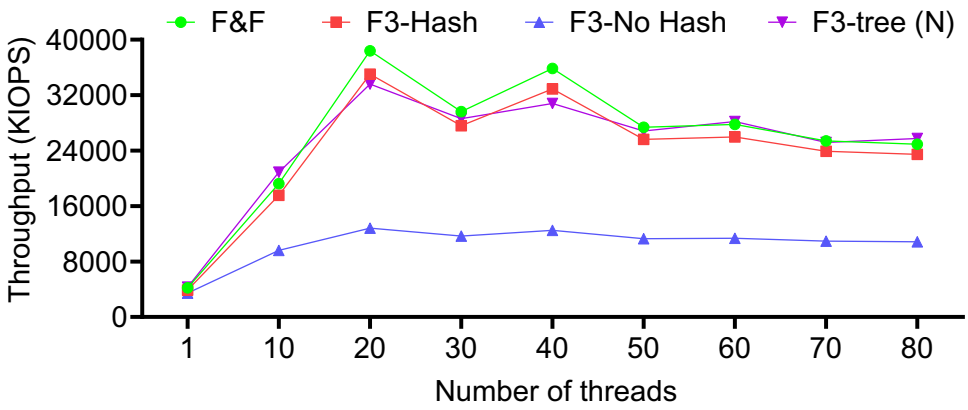


Fig. 10 Search performance analysis. We used the synthetic benchmark with Workload 2 as shown in Table 1

is not found, it then searches the global tree. In constrast, F^3-tree with its hash table shows equivalent performance to F&F with negligible overhead to check the hash table first and then look for the key in the corresponding PTFOs. Additionally, we observe that F^3-tree (N) maintains the read performance of F^3-tree.

5.4 Real-world applications of F^3-tree

As our proposed F^3-tree is an index data structure we argue that our proposed solution can be adopted for all system-level applications where index data structures play a vital role. Additionally, we argue that our proposed solution would have the best performance where applications deal with write intensive workload patterns. An example application of our proposed F^3-tree would be key-value stores.

6 Related work

Data structures based on persistent memory: Existing PM-based KV-stores can be broadly classified into hash and tree-based data structures. HiKV [32] creates a hybrid index including DRAM and PM to efficiently support a range of key-value operations. Using the persistent hash index structure allows fast index searches and for range queries it utilizes a volatile B+-tree. The tree-based CDDS B+-tree [33] was one of the early persistent B+-tree data structures. For failure consistency, CDDS B+-tree adopted the global versioning control technique but the scalability suffers with increasing number threads trying to change the global version number. RNTree [34] takes advantage of hardware transactional memory (HTM). To reduce the sorting overhead it uses a new slot array approach. There have been various other studies to provide optimal persistent data structures such as B+-trees [9, 10, 35], a hashing scheme [36], and a radix tree [37]. Recent work [2] proposed group-split-merge (GSM), a persistent index data structure atop of PMEMKV for scientific indexing and querying. Many of those propose write optimal techniques while delivering consistency of the data structures with 8-byte failure atomic writes in persistent memory. Fast and Fair [11] proposed a B-link-tree for persistent memory using failure-atomic in-place operations, shifting and re-balancing. Consistency is achieved using the previous sorted shift node entries. Further, it performs split node operations without logging by taking advantage of the B-link-tree sibling pointers.

6.1 B+-tree indexing data structures

Recent studies on B+-tree indexing data structures can be classified into hybrid (DRAM-PM) B+-trees [10, 13, 38] and PM-only B+-trees [9, 11, 33]. With PM-only variants, data resides entirely in PM, allowing nearly instant recovery. For hybrid indexes, DRAM is used for auxiliary data that is rebuilt on recovery. DRAM has lower latency than PM, and this scheme usually results in improved performance at the cost of longer recovery time. But most of these studies have either compromised the basic design of the B+-tree for the sake of performance, such as allowing unsorted entries within B+-tree nodes [33], or lack concurrency support [9]. F&F is the state-of-the-art persistent B+-tree variant that maintains the basic properties of B+-trees while supporting concurrent operations.

However, none of the existing PM-based B+-tree write operations scale on manycore machines with hundreds of threads. In this work, we proposed F^3-tree, a highly concurrent persistent B^+-tree for DCPM-based manycore machines. We adopted future-based data structures for asynchronous computations [17] over F&F and achieve higher performance on manycore machines. Futures have not yet been adopted for indexing data structures, and this is the first work that has adopted future-based data structures for B+-tree indexing data structures.

6.2 NUMA-aware system applications and data structures

There have been several studies to introduce NUMA awareness to file systems and system-level applications. nCache [39] 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, 7, 40]. All these works conclude that there is a need to design NUMA-aware data structures that can efficiently utilize NUMA-based manycore machines. Several recent prominent studies, including [3, 41], highlighted the importance of NUMA impact. Daase et al. [42] investigated OLAP-related workload interaction across NUMA regions. Junehyung Kim et al. [4] proposed fine-grained range-based locks to improve the scalability of NOVA [43] with NUMA architectures. NAP[7] established black-box NUMA-aware counterparts for index data structure for DCPM. All those studies explore NUMA awareness in data structures but none of them is directly applicable to FOs. For instance, the NAP approach focuses on crash consistency and failure atomicity inspired by node replication [18] but suffers from space amplification as it maintains hot items in each node. In the proposed work, we focus on achieving scalability and NUMA awareness for persistent FOs without space overhead.

7 Conclusion

In this paper, we presented F^3-tree, a highly concurrent persistent B^+-tree for DCPM-based manycore machines. F^3-tree achieves scalability and high write concurrency by adopting thread-local future objects (PTFOs). PTFOs are checkpointed to the global B^+-tree 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 evaluated F^3-tree on a manycore Linux machine with emulated DCPM. The results show that F^3-tree achieves high scalability (3.4× on average) compared to F&F. We introduced NUMA-awareness with our F^3-tree design and observed significant performance improvements and a reduction in remote memory accesses with the NUMA-aware variant of F^3-tree. Experimental results show that our NUMA-aware F^3-tree improves the performance over F^3-tree without NUMA-awareness on average by a factor of 1.3× for workloads with both sequential and random key distributions. Our NUMA-aware F^3-tree reduces remote memory accesses by 50% and outperforms F&F on average by 5× and 3× for workloads with random and sequential key distributions, respectively.

Acknowledgements The authors would like to thank the reviewers for providing precious comments to improve our work.

Author Contributions SJ did the idea development and most of the implementation and evaluation. AS and AK contributed to the technical discussion as well as contributed to the manuscript writing and proofreading along with BB and S-SP. As the corresponding author, YK supervised the entire process from idea development to implementation, experimentation and evaluation, and paper writing.

Funding 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. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan). This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Data availibility None.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

Informed consent All authors are informed and have consent on the manuscript.

References

1. Jamil, S., Khan, A., Burgstaller, B., Kim, Y.: Towards scalable manycore-aware persistent B+- trees for efficient indexing in cloud environments. In: Proceedings of the 2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), pp. 44–49 (2021)
2. Khan, A., Sim, H., Vazhkudai, S. S., Ma, J., Oh, M.-H., Kim, Y.: Persistent memory object storage and indexing for scientific computing. In: Proceedings of the 2020 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC), pp. 1–9 (2020)
3. Yang, J., Kim, J., Hoseinzadeh, M., Izraelevitz, J., Swanson, S.: 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 (2020)
4. Kim, J.-H., Kim, Y., Jamil, S., Park, S.: A NUMA-aware NVM file system design for manycore server applications. In:

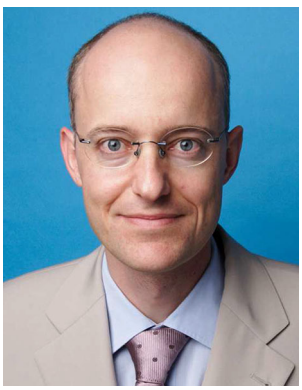
Proceedings of the 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 1–5 (2020)
5. Kim, T., Khan, A., Kim, Y., Kasu, P., Atchley, S.: NUMA-aware thread scheduling for big data transfers over terabits network infrastructure. Sci. Program. **2018** 4120561 (2018)
6. Kim, J., Kim, Y., Khan, A., Park, S.: Understanding the performance of storage class memory file systems in the NUMA architecture. Clust. Comput. **22**(2), 347–360 (2019)
7. Wang, Q., Lu, Y., Li, J., Shu, J.: Nap: a black-box approach to NUMA-aware persistent memory indexes. In: Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), pp. 93–111. USENIX Association, Berkeley (2021)
8. Khan, A., Lee, C.-G., Hamandawana, P., Park, S., Kim, Y.: 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)
9. Chen, S., Jin, Q.: Persistent B+-trees in non-volatile main memory. Proc. VLDB Endow. **8**, 786–797 (2015)
10. Oukid, I., Lasperas, J., Nica, A.,Willhalm, T.,Lehner, W.: 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), pp. 371–386. Association for Computing Machinery, New York (2016)
11. Hwang, D., Kim, W.-H., Won, Y., Nam, B.: Endurable transient inconsistency in byte-addressable persistent B+-tree. In: Proceedings of the 16th USENIX Conference on File and Storage Technologies, FAST'18, pp. 187–200 (2018)
12. Khan, A., Sim, H., Vazhkudai, S.S., Kim, Y.: MOSIQS: Persistent memory object storage with metadata indexing and querying for scientific computing. IEEE Access **9**, 85217–85231 (2021)
13. Yang, J.,Wei, Q., Chen, C., Wang, C., Yong, K. L., He, B.: NV-Tree: reducing consistency cost for NVM-based single level systems. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15), pp. 167–181 (2015)
14. Scott, M.L.: Shared-Memory Synchronization. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, San Francisco (2013)
15. Dice, D., Marathe, V. J., Shavit, N.: Flat-combining NUMA locks. In: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '11, pp. 65–74. Association for Computing Machinery, New York (2011)
16. Chabbi, M., Fagan, M., Mellor-Crummey, J.: 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, pp. 215–226. Association for Computing Machinery, New York (2015)
17. Kogan, A., Herlihy, M.: The future(s) of shared data structures. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, PODC '14, pp. 30–39 (2014)
18. Calciu, I., Sen, S.,Balakrishnan, M., Aguilera, M.K.: Black-box concurrent data structures for NUMA architectures. In: Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17 (New York, NY, USA), pp. 207–221. Association for Computing Machinery, New York (2017)
19. Lehman, P.L., Yao, S.B.: Efficient locking for concurrent operations on B-trees. ACM Trans. Database Syst. **6**, 650–670 (1981)
20. Yi, Z.,Yao, Y., Chen, K.: A universal construction to implement concurrent data structure for NUMA-multicore. In: Proceedings of the 50th International Conference on Parallel Processing (New York, NY, USA), Association for Computing Machinery, New York (2021)
21. Calciu, I., Gottschlich, J., Herlihy, M.: Using elimination and delegation to implement a scalable NUMA-friendly stack. In: Proceedings of the 5th USENIX Workshop on Hot Topics in Parallelism (HotPar 13) (San Jose, CA), USENIX Association, Berkeley (2013)
22. Bhardwaj, A., Kulkarni, C., Achermann, R., Calciu, I., Kashyap, S., Stutsman, R., Tai, A., Zellweger, G.: NrOS: effective replication and sharing in an operating system. In: Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21), pp. 295–312. USENIX Association, Berkeley (2021)
23. Lee, S. K., Mohan, J., Kashyap, S., Kim, T., Chidambaram,V.: Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19, pp. 462–477 (2019)
24. Numactl. https://linux.die.net/man/8/numactl. Accessed: 2021-04-06
25. Defining the future of in-memory database computing. https://pmem.io/vmem/libvmmalloc/. Accessed 1 Dec 2021
26. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, San Francisco (2012)
27. Ramalhete, P., Correia, A.: Brief announcement: hazard eras—non-blocking memory reclamation. In: Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '17, pp. 367–369. Association for Computing Machinery, New York (2017)
28. Cohen, N., Petrank, E.: 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, pp. 254–263. Association for Computing Machinery, New York (2015)
29. Fast &fair B^{+}-tree. https://github.com/DICL/FAST_FAIR. Accessed 07 Feb 2022
30. Yahoo cloud serving benchmark. https://github.com/brianfrankcooper/YCSB/. Accessed 21 Jan 2022
31. Lu, M., Zhixi Fang, J.: A solution of the cache ping-pong problem in multiprocessor systems. J. Parallel Distrib. Comput. **16**(2), 158–171 (1992)
32. Xia, F., Jiang, D., Xiong, J., Sun, N.: HiKV: a hybrid index key-value store for DRAM-NVM memory systems. In: Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 17) (Santa Clara, CA), pp. 349–362. USENIX Association, Berkeley (2017)
33. Venkataraman, S., Tolia, N., Ranganathan, P., Campbell, R. H.: Consistent and durable data structures for non-volatile byte-addressable memory. In: Proceedings of the 9th USENIX Conference on File and Stroage Technologies, FAST'11, p. 5 (2011)
34. Liu, M., Xing, J., Chen, K., Wu, Y.: Building scalable NVM-based B+tree with HTM. In: Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019 (New York, NY, USA). Association for Computing Machinery, New York (2019)
35. Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K. L., He, B.: NV-Tree: Reducing consistency cost for NVM-based single level systems. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15) (Santa Clara, CA), pp. 167–181. USENIX Association, Berkeley (2015)
36. Zuo, P., Hua, Y.: A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. IEEE Trans. Parallel Distrib. Syst. **29**(5), 985–998 (2017)
37. Lee, S.K., Lim, K.H., Song, H., Nam, B., Noh, S.H.: WORT: Write optimal radix tree for persistent memory storage systems. In: Proceedings of the 15th USENIX Conference on File and

Storage Technologies (FAST 17) (Santa Clara, CA), pp. 257–270. USENIX Association, Berkeley (2017)
38. Zhou, X., Shou, L., Chen, K., Hu, W., Chen, G.: DPTree: Differential indexing for persistent memory. Proc. VLDB Endow. **13**, 421–434 (2019)
39. Lee, C.-G., Noh, S., Kang, H., Hwang, S., Kim, Y.: Concurrent file metadata structure using readers-writer lock. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing (New York, NY, USA), pp. 1172–1181. Association for Computing Machinery, New York (2021)
40. Kim, J.-H., Kim, Y., Jamil, S., Lee, C.-G., Park, S.: Parallelizing shared file I/O operations of NVM file system for manycore servers. IEEE Access **9**, 24570–24585 (2021)
41. Peng, I.B., Gokhale, M.B., Green, E.W.: System evaluation of the Intel Optane byte-addressable NVM. In: Proceedings of the International Symposium on Memory Systems, MEMSYS '19 (New York, NY, USA), pp. 304–315. Association for Computing Machinery, New York (2019)
42. Daase, B., Bollmeier, L.J., Benson, L., Rabl, T.: Maximizing persistent memory bandwidth utilization for OLAP workloads. In: Proceedings of the 2021 International Conference on Management of Data, SIGMOD/PODS '21 (New York, NY, USA), pp. 339–351. Association for Computing Machinery, New York (2021)
43. Xu, J., Swanson, S.: NOVA: a log-structured file system for hybrid Volatile/Non-volatile main memories. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16) (Santa Clara, CA), pp. 323–338. USENIX Association, Berkeley (2016)

Safdar Jamil received the B.E. degree in computer systems engineering from Mehran University of Engineering and Technology (MUET), Jamshoro, Pakistan, in 2017. He is currently pursuing an Integrated MS-Ph.D. degree with Sogang University, Seoul, South Korea. His research interests include data structures and algorithms, persistent memory, database systems, and system energy optimization.

Abdul Salam received the B.E. degree in computer systems engineering from Mehran University of Engineering and Technology (MUET), Jamshoro, Pakistan, in 2017. He is currently pursuing a M.S. degree with Sogang University, Korea. His research interests include persistent memory and index data structures for big data processing systems.

Awais Khan received his Ph.D. degree in Computer Science and Engineering from Sogang University, Seoul, South Korea in 2021. He received his B.S. degree in Bioinformatics from Mohammad Ali Jinnah University, Islamabad, Pakistan. He worked in Digital Research Laboratories as software engineer from 2012 to 2015. Currently, he is working as a Postdoctoral Research Associate in Oak Ridge National laboratory, TN, USA. His research interests include service optimizations for AI/ML applications, memory-centric computing and HPC, data management services in HPC, object storage systems, cluster-scale deduplication, parallel and distributed file systems.



Bernd Burgstaller received the Ph.D. degree from the Vienna University of Technology in 2005. He was a postdoctoral researcher at the University of Sydney until 2007. He is currently an associate professor in the Department of Computer Science at Yonsei University. His research interests include programming languages, parallel computing on multicore architectures, and embedded systems. Before pursuing an academic career, he spent three years as a software engineer and architect at Philips Consumer Electronics, Vienna.

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 Full Time Lecturer 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 a professor of the Department of Computer Science and Engineering at Anyang University and CEO of Gluesys Co. LTD. His research areas include network storage systems and cloud computing.

Youngjae Kim received his 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 at Sogang University, Seoul, Republic of Korea. Before joining Sogang University, he was a staff scientist in the US Department of Energy's Oak Ridge National Laboratory (2009-2015) and an assistant professor in Ajou University, Suwon, Republic of Korea (2015-2016). He received the B.S. degree in Computer Science from Sogang University in 2001, and the M.S. degree in Computer Science from KAIST in 2003. His research interests include operating system, file and storage system, database system, system security, and distributed system.