RECIPE : Converting Concurrent DRAM Indexes to Persistent-Memory Indexes

Se Kwon Lee
University of Texas at Austin

Jayashree Mohan
University of Texas at Austin

Sanidhya Kashyap
Georgia Institute of Technology

Taesoo Kim
Georgia Institute of Technology

Vijay Chidambaram
University of Texas at Austin and
VMware Research

Abstract

We present Recipe, a principled approach for converting concurrent DRAM indexes into crash-consistent indexes for persistent memory (PM). The main insight behind Recipe is that *isolation* provided by a certain class of concurrent in-memory indexes can be translated with small changes to *crash-consistency* when the same index is used in PM. We present a set of conditions that enable the identification of this class of DRAM indexes, and the actions to be taken to convert each index to be persistent. Based on these conditions and conversion actions, we modify five different DRAM indexes based on B+ trees, tries, radix trees, and hash tables to their crash-consistent PM counterparts. The effort involved in this conversion is minimal, requiring 30–200 lines of code. We evaluated the converted PM indexes on Intel DC Persistent Memory, and found that they outperform state-of-the-art, hand-crafted PM indexes in multi-threaded workloads by up-to 5.2×. For example, we built P-CLHT, our PM implementation of the CLHT hash table by modifying only 30 LOC. When running YCSB workloads, P-CLHT performs up to 2.4× better than Cacheline-Conscious Extendible Hashing (CCEH), the state-of-the-art PM hash table.

CCS Concepts • **Information systems** → **Data structures**; **Key-value stores**; **Storage class memory**; **Indexed file organization**; • **Hardware** → **Non-volatile memory**.

Keywords Persistent Memory, Data Structures, Indexing, Crash Consistency, Concurrency, Isolation

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359635

ACM Reference Format:
Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. RECIPE : Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3341301.3359635

1 Introduction

Persistent memory (PM) is an emerging class of memory technology. The first PM product, Intel DC Persistent Memory, was announced in April 2019 [35]. Intel's PM product will be attached to the memory bus and accessed like DRAM via processor loads and stores. It has a unique performance profile: read latency 3.7× that of DRAM, while read and write bandwidth are $1/3^{rd} - 1/6^{th}$ that of DRAM [38].

The low latency and durability of PM make it an attractive medium for building storage systems. Indexes are key to achieving good read performance, and are thus a crucial component of several storage systems. Researchers have designed several PM indexes such as FAST & FAIR [31], Level Hashing [78], CCEH [55], NV-Tree [75], wB+ tree [9], WOART [44], and FPTree [57]. Designing these indexes from scratch is challenging; the indexes must provide high performance and concurrency while ensuring that the index recovers correctly in the event of a power loss or a system crash. This complexity leads to subtle bugs; for example, we identify previously unknown data-loss and crash-recovery bugs in the FAST & FAIR B+ tree and CCEH hash table (§3).

While research on building concurrent, crash-consistent PM indexes has been gathering traction recently, there have been decades of research on building concurrent DRAM indexes [15, 16, 21, 26, 27, 53, 65, 66, 69]; for example, the skip list [61] was invented thirty years ago. Modern in-memory data structures are carefully designed keeping in mind cache efficiency [62], pre-fetching [8], concurrency, and parallelism via SIMD instructions [10, 74]. Concurrent data structures

Se Kwon Lee and Jayashree Mohan are supported by SOSP 2019 student travel scholarships from the National Science Foundation and ACM Special Interest Group in Operating Systems respectively.

are widely used in industry and academia; for example, latch-free BwTree in the Hekaton OLTP engine [18], Adaptive Radix Trees in the HyPer database [40], the Timeline Index in SAP HANA [39], and Masstree in the Silo database [68]. In this work, we seek to leverage the research on concurrent DRAM indexes to build persistent PM indexes.

We present Recipe, a principled, practical approach for converting concurrent DRAM indexes into their persistent, crash-consistent counterparts. If the source DRAM index that is converted by using the Recipe approach is correct, the resulting PM index will be correct as well. We call the index *correct*, if no previously inserted key is lost and a search returns the latest value of the key. Recipe can only be applied to DRAM indexes meeting specific conditions, and the conversion process differs based on the matching conditions. Therefore, we introduce a set of conditions that specify which DRAM indexes can be converted using the Recipe approach; DRAM indexes meeting these conditions can be converted to their PM counterparts with minimal changes. We convert five popular DRAM indexes into their PM counterparts; all conversions required less than 200 LOC (1–9% of the codebase). Each converted index uses a different data structure: a hash table, a trie, a B+ tree, a radix tree, and a combination of tries and B+ trees. The converted PM indexes offer good performance and scalability, outperforming hand-crafted state-of-the-art PM indexes on many workloads.

The basic insight behind Recipe is that isolation in concurrent DRAM indexes is closely related to crash consistency in persistent indexes. Isolation ensures that reads return correct values and writes result in consistent states irrespective of other active reads and writes. We can view crash consistency similarly: reads after a crash return correct values, and writes after a crash lead to consistent states (perhaps by fixing inconsistencies). To increase performance, many DRAM indexes employ non-blocking synchronization. They allow reads and writes to see inconsistent states; the reads and writes have the ability to detect inconsistencies, and either tolerate or fix them. This is the exact set of features required to recover correctly after a crash; such DRAM indexes basically have crash-recovery logic woven into their reads and writes. We make the observation that converting such DRAM indexes into their PM indexes is much more straight-forward than designing PM indexes from scratch; since PM and DRAM are both accessed via the same load and store instructions, a developer only has to ensure stores are correctly ordered using memory fences and flushed from volatile caches to persistent media. No new crash recovery algorithms (which tend to be complex) are required to be added to the converted PM index, as the reads and writes can detect and tolerate or fix inconsistencies already.

The challenge in developing the Recipe approach is carefully reasoning about which DRAM indexes can be converted, and how to convert compatible indexes. For example, read operations in some lock-free DRAM indexes, on finding an

DRAM	Data	RECIPE	Lines of Code		
Index	Structure	Condition	Orig	Core	Modified
CLHT	Hash Table	#1	12.6K	2.8K	30 (1%)
HOT	Trie	#1	36K	2K	38 (2%)
Bw Tree	B+ Tree	#2	13K	5.2K	85 (1.6%)
ART	Radix Tree	#3	4.5K	1.5K	52 (3.4%)
Masstree	B+ Tree & Trie	#3	25K	2.2K	200 (9%)

Table 1. Categorizing common DRAM indexes. The table enumerates popular DRAM indexes, the RECIPE condition they satisfy, and the effort required to convert them to their PM versions. The lines of code in the core codebase is calculated by excluding tests and helper libraries.

inconsistency based on version numbers, simply back-off and retry, assuming the inconsistency is transient. This approach does not work if a crash has left the index in a permanently inconsistent state. We present three conditions and conversion actions that precisely capture the properties the source DRAM index should have, and how to convert the source DRAM index. The guidance provided by Recipe is not at the source level, and thus cannot be easily automated. However, we found that the conditions are broadly applicable; we were able to convert five different DRAM indexes by modifying fewer than 200 lines of code.

Building a PM index using the Recipe approach offers several benefits. First, it drastically lowers the complexity of building a PM index; the developer simply chooses an appropriate DRAM index and modifies it as indicated by our approach. The developer does not have to worry about crash recovery, even in the presence of concurrent writes. Second, if the developer converts a DRAM index that has high performance and scalability, the converted PM index also offers good performance without any further optimization.

We present our experience with converting five DRAM indexes using Recipe, each based on a different data structure: Adaptive Radix Tree (ART) [46, 47], Height Optimized Trie (HOT) [4], BwTree [48], Cache-Line Hash Table (CLHT) [15] and Masstree [52]. Table 1 categorizes these DRAM indexes based on the condition they satisfy, and the effort involved in converting them to be persistent (as a percentage of change to the core codebase that excludes tests and helper libraries). Masstree was by far the most complicated of these indexes, as it combines tries and B+ trees. By applying Recipe, we were able to convert all indexes to their PM counterparts with 1–9 % change to the core source code.

To test the correctness of our converted PM indexes, we introduce a new methodology for testing crash recovery. We take advantage of the fact that insert and structure-modification operations (such as a node split in a B+ tree) in non-blocking indexes are comprised of a small number of ordered atomic steps. We instrument the code to simulate a crash in between these atomic steps. Simulating a crash involves returning from an insert or structure-modification operation mid-way without cleaning up any state, leaving the index in a partially modified state. We then continue reading and writing to the index using multiple threads, testing that the reads return expected values and writes complete successfully. We also trace all dynamic memory allocations, stores, and cache line flushes using PIN [51], and check that all dirtied cache lines in allocated memory ranges are flushed to PM. Though this method is not exhaustive, it is powerful, allowing us to find data-loss and crash-recovery bugs in CCEH and FAST & FAIR. Testing did not reveal any bugs in the PM indexes we converted.

To test the performance of our converted PM indexes, we use the YCSB [14] benchmark to perform multi-threaded insertions, point queries, and range queries on Intel DC Persistent Memory. We compare the converted PM indexes against state-of-the-art manually-designed PM indexes. We find that our converted PM indexes outperform the state-of-the-art by up-to 5.2× in multi-threaded YCSB workloads. The main performance gain for Recipe-converted indexes comes from the fact that the DRAM indexes we convert are already optimized for concurrency and cache efficiency; the high read latency of PM makes cache efficiency even more important.

The Recipe approach has a number of limitations. Recipe cannot be applied to any DRAM index that does not match one of its three conditions. For instance, Recipe cannot be applied to indexes with blocking reads or non-blocking reads with version-based retry. Recipe assumes that the locks used in an index are not persistent, and are reinitialized to prevent deadlock when an index recovers from a crash. Recipe assumes garbage collection is employed for the persistent-memory allocator to reclaim unreachable objects. Recipe assumes the original DRAM index is correct; if it has a bug, the converted PM index will also have a bug. Finally, the main focus of this approach is the correct and principled conversion of DRAM indexes into PM indexes; there are usually opportunities for further optimization.

In summary, this paper makes the following contributions:

- Recipe, a principled approach to convert DRAM indexes into PM indexes (§4).
- An efficient method for testing crash recovery of PM indexes (§5).
- A case study of converting five DRAM indexes based on different data structures to PM indexes using the Recipe approach (§6). Recipe-converted indexes are available at https://github.com/utsaslab/RECIPE.
- Experimental evidence showing Recipe-converted PM indexes recover correctly from crashes, and achieve performance and scalability competitive with state-of-the-art hand-crafted PM indexes (§7).

2 Background

We begin by describing DRAM indexes and their interfaces, persistent indexes, and how indexes achieve concurrency and scalability. We then motivate why a principled approach is required for building PM indexes.

2.1 DRAM Indexes

DRAM indexes are used to efficiently lookup data items in databases, file systems, and other storage systems. Their interface involves five main operations:

insert(key, value) inserts the pair of `key` and `value` into the index. `value` is usually the location in the storage system where `key` can be found.

update(key, value) update `key` with `value` in the index. Some key-value stores use `insert` for both insertions and updates, while other key-value stores will fail insertions if the key already exists.

lookup(key) returns the `value` associated with `key` in the index.

range_query(key1, key2) returns all key-value pairs where the keys are within the specified range. Range queries are sometimes implemented using an iterator: a cursor that can be incremented to the next key in the sequence.

delete(key) removes the specified key from the index.

Structural Modification Operations (SMOs). SMOs are operations internal to the data structure, that are required either to ensure that the invariants of the data structure holds, or to improve performance. For instance, when the nodes in a B-tree overflow (during insertion) or underflow (during deletion), node splits or merges are required to re-establish the invariants of a B-tree. In other data structures like hash tables, SMOs like re-hashing are necessary to keep constant average cost per operation.

Performance. DRAM indexes take special care to have high lookup and insertion performance, as these are often performed in the critical path. Lookup and insertion performance depend on the number of processor loads and store required, along with aspects like whether the layout is cache-friendly and prefetcher-friendly.

Correctness. A DRAM index should return the latest inserted `value` for any given `key`. Unless the key is explicitly deleted, an inserted key should never be lost.

2.2 Concurrency and Isolation

DRAM indexes use multiple threads to increase throughput on multi-core machines. However, since all threads operate on the same shared index, additional mechanisms are required to ensure correctness. Concurrent DRAM indexes need to provide *isolation*: ensuring that even if multiple writers are modifying the index at the same time, the final index state corresponds to the insertions or updates happening in some sequential order. The index also needs to ensure that reads do not reflect the result of a partial or incomplete insertion or update operation.

Blocking operations. The easiest way to ensure correctness in a concurrent index is to obtain a lock on the index, and only allow threads with lock to read or write. This serializes all operations and decreases throughput to that of a single thread. To increase performance, reader-writer locks are often used [54, 63, 78]; readers can get a shared lock, all writers have to contend on a single lock, and there is mutual exclusion between readers and the writer.

Non-blocking operations. Non-blocking operations [67] are employed to fully exploit the parallelism offered by modern hardware. Non-blocking operations guarantee progress of some or all remaining threads regardless of the suspension, termination, or crash failure of one of the threads [22, 25]. They provide consistency and correctness by carefully ordering load and store instructions using memory fence (`mfence`) [1, 28], while avoiding the use of mutual exclusion and expensive synchronization primitives.

Non-blocking operations can be categorized into lock-free and wait-free, based on their progress guarantee. Lock-free operations allow multiple threads to simultaneously access a shared object, while guaranteeing that at least one of these operations finish after a finite number of steps [25]. Wait-free operations are a subset of lock-free operations, with the additional condition that every thread finishes the operation in a finite number of steps [25].

Non-blocking operations are built using hardware-atomic primitives such as compare and swap (CAS) or test and set. If every update is performed via a single atomic store, correctness is implicitly guaranteed. If updates consist of a sequence of atomic stores, then the readers can either make progress by reasoning about the deterministic order of stores, or can use additional techniques such as version-based retry [6, 20].

While non-blocking operations are known to provide high performance and scalability, high contention to the shared resource reduces performance and could lead to starvation [19]. For example, if a lock-free write is interrupted by the scheduler, it might need to retry the operation after being rescheduled if the shared state has changed. To protect against starvation, many indexes use non-blocking reads and blocking, lock-protected writes [4, 15, 47, 52].

2.3 Persistent Memory

Persistent Memory (PM) bridges the gap between DRAM and storage by offering DRAM-like latency with storage-like persistence. Writes to the PM are issued in 8-byte failure-atomic units, which are first written to the volatile CPU cache. These cache lines can be written back to the Persistent Memory Controller in an arbitrary order. Intel x86 architecture provides the `mfence` instruction to prevent such memory reordering [33]; if a store instruction is followed by a `mfence`, then it is guaranteed to be visible before any other stores that follow the `mfence`. Additionally, to explicitly flush a cache line to the persistent controller, x86 architecture provides `clflush`, `clwb` and `clflushopt` instructions. Our work uses `clwb` and `mfence` to guarantee persistence.

2.4 Crash-Consistent PM Indexes

Building PM indexes is attractive for two reasons. First, the larger capacity of PM at close-to-DRAM latencies allows using larger indexes than possible with just DRAM. Second, DRAM indexes need to be reconstructed after a crash; for large indexes, reconstruction could take several minutes or hours. In contrast, a PM index is instantly available. This has motivated a number of researchers to design efficient indexes on PM; we count fifteen PM indexes published in top systems and database conferences since 2015. The PM indexes include variants of B+ trees [2, 9, 11, 31, 41, 57, 70, 73, 75], radix trees [44], and hash tables [55, 64, 72, 77, 78].

Crash Recovery. One of the main differences between a DRAM index and a PM index is that the PM index has to ensure that it can correctly recover in the case of power loss or kernel crash. This requires carefully ordering stores to PM using `mfence` instructions and then flushing the dirty data from volatile caches to persistent media using cache line flush instructions (`clflush`, `clwb`, or `clflushopt`) [58]. If the write is larger than eight bytes, a crash could lead to a torn write where the data is partially updated; techniques such as logging [24] and copy-on-write [29] are used to provide atomicity.

3 Motivation

Hand-crafted PM indexes employ non-blocking operations to increase scalability [31, 55]. However, while non-blocking operations offer high performance and scalability, their complexity makes it challenging to develop, test, and debug indexes with non-blocking operations. Persistence makes the problem even harder, since developers have to ensure that crash recovery and concurrency mechanisms interact correctly. We analyze two state-of-the-art PM indexes: the FAST & FAIR B+ tree [31], and the CCEH hash table [55].

FAST & FAIR. FAST & FAIR is a PM B+ tree that provides lock-free reads. The reads detect and tolerate inconsistencies such as duplicated elements in a sorted list. Writers hold a lock for mutual exclusion. The writes detect inconsistencies such as duplicated elements, and try to fix them. However, we found that concurrent writes could lead to loss of a successfully written key.

Consider the following scenario. Two threads try to insert keys to the same internal node concurrently; one thread gets a lock and performs a node split. When the other thread gets the lock, it does not realize the node has been changed, and inserts the key into the wrong node. The insert is successful, but a reader would never be able to find the inserted value.

We confirmed this design-level bug with the FAST & FAIR authors. The solution is to add metadata about the high-key to B+ tree nodes, as done by prior works [6, 48, 52]. Please refer to our bug report for more details [43].

We also found an implementation bug. According to its design, FAST & FAIR recovers correctly from crashes at any point, not losing any inserted keys. However, when we crashed FAST & FAIR consecutively in the middle of split and merge operations on two nodes, keys present in the right node were lost. This is a testament to the complexity of these indexes; a correct design is not always translated properly to a correct implementation.

Finally, we found that incorrect crash recovery can result in poor performance. If FAST & FAIR crashes in the middle of splits, although the recovered structure is correct, it is not efficient. A series of such crashes transforms the B+ tree into a linked list, leading to poor read and write performance.

In summary, our investigation of FAST & FAIR revealed a design-level bug that lost data, an implementation-level bug that lost data, and that crashes can lead to poor performance. The design-level bug resulted from not leveraging prior research on concurrency, where the high-key problem and its solution is well-known.

CCEH. We discovered that the CCEH PM hash table [55] has two bugs: one in its directory doubling code, and one in crash recovery code. Directory doubling is similar to rehashing the hash table. There are three pieces of metadata that CCEH has to atomically update in correct order during directory doubling: the pointer to the directory, the directory width, and the global depth. If a crash happens before the global depth is updated, insertion operations loop infinitely. If a crash happens after the pointer to the directory is swapped, the crash recovery algorithm goes into an infinite loop. The authors of CCEH have acknowledged both bugs.

Summary. We find the ad-hoc design of concurrent, crash-consistent PM indexes makes it hard to reason about behavior during concurrent writes and crashes, leading to bugs. There is a need both for principled design of PM indexes, and testing whether PM indexes correctly recover from crashes.

4 The Recipe Approach

We present Recipe, a principled approach for converting a specific class of DRAM indexes to their crash-consistent PM counterparts. The converted PM index inherits correctness and scalability from the DRAM index. The Recipe approach guarantees that the converted PM index will recover from crashes correctly. Thus, if the developer uses the Recipe approach to convert an appropriate DRAM index, the resulting PM index will be correct, concurrent, and crash-consistent.

Recipe identifies three categories of DRAM indexes to guide this conversion. Each category is accompanied by a condition and conversion action of the form: "*if the DRAM index satisfies these conditions, then convert it to a PM index using these conversion actions*". We first present the intuition behind the Recipe approach, and then describe each category.

4.1 Overall Intuition

We observe that some DRAM indexes use non-blocking reads (such as lock-free reads) to improve performance. These non-blocking reads may observe inconsistent states since writes may be underway at the time of read; the read operations can then *tolerate* such inconsistencies, returning a consistent answer to the user. For example, the read operation may see duplicate records and only return a single record to user [23, 31]. Similarly, write operations may also see an inconsistent state and *fix* the inconsistency; write operations in BwTree perform such fixes [48]. Prior theoretical work has termed this a *helping mechanism*, where an operation started by one thread which fails is later completed by another thread [5].

The Recipe approach is based on the following insight: if reads can tolerate inconsistencies, and writes can fix them, a separate crash-recovery algorithm is not required. DRAM data structures that have such read and write operations are *inherently crash-consistent*. If such data structures are stored on PM instead of DRAM, they would be crash-consistent with minimal modifications; the developer would only need to ensure that all data dirtied by store operations are persisted to PM in the right order. We refine this observation through three conditions with corresponding conversion actions that help a developer convert a DRAM index into a crash-consistent, concurrent PM index.

4.2 Assumptions and Limitations

Recipe assumes that the locks used in the index are non-persistent, and that the locks are re-initialized after a crash (to prevent deadlock). Recipe also assumes that unreachable PM objects will be garbage collected, as a failed update operation may result in an allocated but unreachable object. Finally, Recipe also assumes that the DRAM index operates correctly in the face of concurrent writes.

Recipe can only be applied to DRAM indexes that match one of the three conditions. For example, DRAM indexes which employ blocking reads or non-blocking reads with retry mechanisms cannot be converted using Recipe. The three conditions which follow specify precisely which DRAM indexes can be converted by Recipe.

4.3 Condition #1: Updates via single atomic store

Reads must be non-blocking, while writes may be blocking or non-blocking. The index makes write operations visible to other threads using a single hardware-atomic store.

Conversion Action. Insert cache line flush and memory fence instructions after each `store`.

Fig 1 illustrates the scenario covered by Condition #1. The index moves in a single atomic step from its initial state to its final state. A crash at any point leaves the index consistent, so crash recovery is not required.

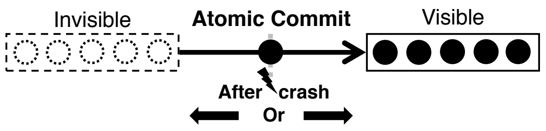


Figure 1. Condition 1. When a crash occurs in the middle of an update operation that completes in a single hardware atomic step, there is no recovery required. The state after the crash is either the initial state or the final state.

Converting an index that fits these conditions is straightforward; each store instruction must be followed by a cache line flush and a memory fence instruction. This ensures that all dirty data is flushed to PM, and that the order in which the writes happen in CPU cache is the same order in which they are persisted to PM. Performance can be increased by allowing stores preceding the final critical store to be reordered [58]. Instead of putting a fence after each store, we would need fences only surrounding the final atomic store.

Examples. We converted two indexes, the Height Optimized Trie (HOT), and the Cache-Line Hash Table (CLHT) based on Condition #1. These indexes employ copy-on-write for updates and failure-atomically make them visible to other threads via atomic pointer swap. Thus, their conversions just require adding cache line flushes and memory fences after each store.

4.4 Condition #2: Writers fix inconsistencies

Reads and writes must be both non-blocking. The index performs write operations using a sequence of ordered hardware-atomic stores. If the reads observe an inconsistent state, they detect and *tolerate* the inconsistency without retrying. If writes detect an inconsistency, they have a *helping mechanism* which allows them to fix the inconsistency.

Conversion Action. Insert cache line flush and memory fence instructions after each `store`.

Figure 2 illustrates the scenario for Condition #2. A crash leaves Thread 1's write partially completed. Thread 2 is able to detect this; since the write operation comprises of a small sequence of deterministic steps, Thread 2 can identify where the crash happened. Thread 2 then proceeds to complete the operation, and then proceed with its own write. This restores the index back to its consistent state. Any read observing these actions is able to tolerate the inconsistency and return a consistent value back to the user.

Note that in general, it is hard after a crash to identify what happened before the crash if extra information is not logged. Indexes meeting Condition #2 are able to do this because write operations in such indexes are comprised of a small number (typically fewer than five) of ordered store operations which mutate the index in a deterministic fashion. Thus, after a crash, the write operation can always deduce what happened before a crash.

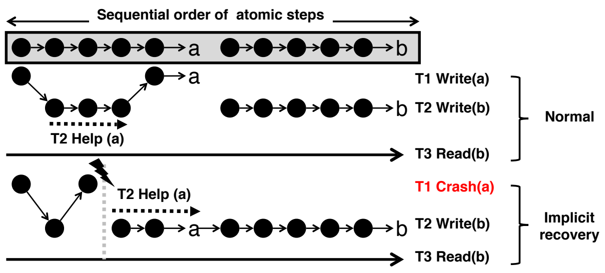


Figure 2. Condition 2. A crash occurs in the middle of Thread 1's write operation. Thread 2 detects this, completes Thread 1's write operation using its helper mechanism, and then proceeds with its own write operation.

Indexes matching Condition #2 do not need any explicit crash-recovery code because *implicit crash recovery* is already part of the read and write operations. The first writer that tries to update the index after a crash and detects the inconsistency is responsible for the recovery of the part of the index the writer deals with. As a result, the same conversion action from Condition #1 still applies; every store instruction should be followed by a cache line flush and a memory fence instruction.

Example. The BwTree has non-blocking read and write operations. It uses a sequence of ordered atomic stores to perform Structural Modification Operations (SMO) like node splits and merges. BwTree write operations have helper mechanisms which complete and commit any intermediate SMO state encountered, before proceeding with their own write. Thus, BwTree fits into Condition #2, and we converted it to its persistent version simply by adding cache line flushes and memory fences.

4.5 Condition #3: Writers don't fix inconsistencies

Reads must be non-blocking, while writes must be blocking. Write operations involve a sequence of the ordered atomic steps similar to Condition #2, but they are protected by write exclusion (locks). Reads can detect and tolerate inconsistencies. Writes can detect inconsistencies; however, they lack the helper mechanisms needed to fix the inconsistency.

Conversion Action. Add mechanism to allow writes to detect permanent inconsistencies. Add helper mechanism to allow writes to fix inconsistencies. Insert cache line flush and memory fence instructions after each `store`.

Indexes conforming to Condition #3 are the hardest to convert, as they require multiple steps. The root of the problem is that Condition #3 indexes do not have helper mechanisms in their write operations. Therefore while reads and writes tolerate inconsistencies, the permanent inconsistency will never get fixed.

First, the write operation must distinguish between a transient inconsistency due to another on-going write or a permanent inconsistency due to a crash. It differentiates these

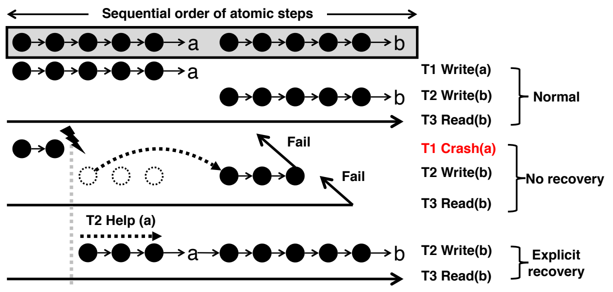


Figure 3. Condition #3. Condition #3 indexes lack the helper mechanism which allows them to resume an interrupted write operation. We explicitly add the helper mechanism which identifies that Thread 1's write operation was interrupted, and finishes the write operation before proceeding with Thread 2's write operation.

scenarios by trying to acquire the write lock; if it is successful, there are no other writes happening concurrently, so an inconsistent state must be due to a crash.

Second, a helper mechanism must be added to finish an interrupted write operation. We find that helper mechanism can be built using code from the write path. The helper mechanism must first identify what was happening at the point of the crash (similar to Condition #2); it must then complete the interrupted write operation. Figure 3 illustrates that explicit recovery code must be added into the writer for Condition #3 indexes.

Adding the helper mechanism to write operation is correct since it re-uses code from the write path; reads can already tolerate the inconsistencies due to on-going writes. Adding the helper mechanism converts a Condition #3 index into a Condition #2 index. At this point, only adding cache line flushes and memory fences after each store are required to produce a crash-consistent, concurrent PM index.

Example. The Adaptive Radix Tree (ART) falls into the category of Condition #3. The writes in ART do not have the helper mechanism, so they just tolerate inconsistencies, when encountering an intermediate state of Structural Modification Operations (SMO). Fortunately, ART's SMO consist of exactly two ordered steps; after a crash, the helper mechanism only needs to identify if step one or two has occurred. We modified ART to introduce permanent inconsistency detection and helper mechanisms, along with adding cache line flushes and memory fences.

5 Testing Crash Recovery of PM Indexes

We introduce a novel method to test whether PM indexes recover correctly after crashes. Testing crash recovery involves testing two things: whether the PM index recovers to a consistent state, and whether the PM index loses any data successfully persisted before the crash. Consistency for a PM index involves reads and range queries of all previously inserted keys returning the correct values, and further writes completing successfully.

The main challenge in testing crash recovery is deciding where to crash in each workload. A crash could happen after each 8-byte atomic store in a workload; this makes the total space of crashes in a reasonable workload prohibitively large. We address this challenge by observing that most operations in PM indexes are comprised of a small number of atomic stores; it is enough to simulate a crash after each atomic store. For each operation in a PM index, we simulate a crash after all its atomic stores. This is feasible since PM indexes have few operations, and each operation has few atomic steps. Structure modifications operations and insertions have less than five atomic steps in all the PM indexes we tested. Thus, crashing only after atomic stores drastically reduces the search space. While there are existing tools like PM-Inspector [34], `pmreorder` [60], and `yat` [42] to simulate crashes, these tools still pick crash points in a random or exhaustive manner; our targeted crashing strategy is powerful, revealing bugs with limited testing.

Testing consistency. We test for consistency using three steps. First, we run a write-heavy workload, and probabilistically simulate a crash after an atomic store in either insertion or a structure modification operation like a node split. A crash is simulated by returning from the operation without any clean-up activities, leaving a partially modified state. Next, we explicitly call the recovery function if the PM index has one. We perform a number of read and write operations using multiple threads, keeping track of all successfully inserted keys. Finally, we read back all successfully inserted keys and check that they have the right values. Note that this approach does not require actual PM; we are able to emulate crashes using DRAM.

Testing durability. Testing durability involves checking that all cache lines which were dirtied during the workload are flushed to PM. This ensures that data written to the PM index is not lost if there is a crash. To test durability, we use the Pin [51] tool to trace all allocations made using `malloc`, `posix_memalign`, and `new`. We then trace all store instructions to these allocated regions, and verify that all dirtied cache lines are safely flushed to PM. We perform this testing using two phases: a load phase and a test phase. We first load the index with enough keys such that future insertions will trigger node splits and other structure modification operations. In the test phase, we perform the insertion while tracing allocation, stores, and cache line flushes. For each insertion, we verify that all dirtied cache lines were persisted.

6 Case Studies

We describe how we modified five concurrent DRAM indexes to their PM counterparts. For each index, we discuss and modify the main write operations of the indexes in accordance with the proposed conversion actions. The operations we

DRAM	Synchronization		Conditions	
Index	Reader	Writer	Non-SMO	SMO
CLHT	Non-blocking	Blocking	#1	#1
HOT	Non-blocking	Blocking	#1	#1
BwTree	Non-blocking	Non-blocking	#1	#2
ART	Non-blocking	Blocking	#1	#3
Masstree	Non-blocking	Blocking	#1	#3

Table 2. Categorizing convertion actions. The table lists the converted DRAM indexes with their category and synchronization properties.

modify are classified into Structural Modification Operations (SMOs) and Non-SMOs (Inserts and Deletes). Non-SMOs affect a single node (in tree based indexes) or a single bucket (in hash tables), whereas SMOs require changes to multiple nodes or buckets. Table 2 lists the converted indexes along with their categories and properties.

Lock initialization. Some of the converted indexes use locks for write exclusion. These locks are embedded into the node or bucket structure and are persisted along with the node. However, locks are required only to provide concurrency; persisting them can result in deadlocks if a system crash occurs. We re-initialize locks on startup for all indexes converted using Recipe. We statically allocate a lock table that holds pointers to each node's lock. This lock table is initialized when the PM index is restarted after crash.

Crash detection. When a converted PM index detects an inconsistency during path traversal, it tries to acquire the lock for the node using `try lock`. If it fails to acquire the lock, either the inconsistency is transient due to a concurrent write, or another write operation is in the process of fixing the inconsistency. If the write operation acquires the lock, it fixes the inconsistency using the helper mechanism.

6.1 Trie: Height Optimized Trie (HOT)

The Height Optimized Trie (HOT) is a lookup and space-optimized variant of a trie, where the children of each node in the search tree share a prefix of the key. HOT achieves cache efficiency, dynamically varying the number of prefix bits mapped by a node to maintain consistent high fanout. The layout is designed for compactness and fast lookup using SIMD instructions.

Non-SMOs. HOT uses copy-on-write and commits an insert or delete operation by atomically swapping the single parent pointer per operation. It uses non-blocking read and exclusive write to prevent the updates from getting lost due to competing pointer swap operations.

SMOs. SMOs in HOT occur when prefix bits are mismatched. If SMOs are required during insertion and deletion, HOT first identifies the set of nodes to be modified, locks them bottom up to avoid deadlock, performs the update using copy-on-write and then unlocks them top down.

Conversion to PM. HOT abides by Condition #1 because every update to the index is installed through an atomic pointer swap. Therefore, as long as the `store` instructions are correctly ordered and flushed, crashes will not result in inconsistencies. Conversion to P-HOT required adding 38 LOC (<2% of the 2K LOC in HOT core).

6.2 Hash Table: Cache-Line Hash Table (CLHT)

CLHT is a cache-friendly hash table that restricts each bucket to be of the size of a cache line (64 bytes). At most three key-value pairs, whose keys and values are 8 byte each, fits into one bucket. The design aims at addressing the cache-coherence problem by ensuring that each update to the hash table requires one cache line access in the common case. To ensure that a non-blocking reader finds the correct value, CLHT uses atomic snapshots of key-value pairs [15, 17]

Non-SMOs. CLHT installs any update to the hashtable by locking the appropriate bucket, performing the update in-place and then unlocking it. CLHT installs the insert and delete operation using a single atomic commit point, ordered by memory fences: writing the correct value first prior to updating 8 byte key (for insertion) and writing 0 to the key (for deletion).

SMOs. If the inserts extend the number of buckets per hash beyond a threshold, CLHT performs re-hashing using copy-on-write. The old hash table is first locked for write. The entries in each bucket are then copied over to the new hash table, and finally, the old hash table is atomically swapped with the new one.

Conversion to PM. CLHT abides by Condition #1 because the inserts, deletes, and re-hashing are effected via a single atomic store. Similar to HOT, we insert cache line flushes and memory fences after appropriate `store` instructions to build P-CLHT. Common-case non-SMOs (inserts and deletes), except for re-hashing, require only one cache line flush per update. Conversion involved 30 LOC (CLHT lock-based implementation is 2.8K LOC).

6.3 B+ TREE: BwTree

BwTree is a variant of B+ tree that provides non-blocking reads and writes. It increases concurrency by prepending delta records (describing the update) to nodes. It uses a mapping table that enables atomically installing delta updates using a single Compare-And-Swap (`CAS`) operation. Subsequent reads or writes to this node replay these delta records to obtain the current state of the node.

Non-SMOs. Insert and delete operations prepend the delta record to the appropriate node, and update the mapping table using `CAS`. If a `CAS` to the mapping table fails because of another concurrent update, the thread simply aborts its operation and restarts from the root.

SMOs. When the base node in BwTree overflows (or underflows), a node split (or merge) is necessary. BwTree uses a helper mechanism [48] to co-operatively perform concurrent updates in the presence of structural modifications due to node splits and merges. Any subsequent writer thread that observes an ongoing split or merge operation first tries to complete it, before going forward with its own operation.

Splits and merges first post a special delta record to the node to indicate that a modification is in progress. It then uses the two-step atomic split mechanism of B-link trees [45] to create a new sibling node in the first step and later update the split key in the parent node. For node merges, the left sibling of the node to be merged is updated with a physical pointer to this node and then the merge key in the parent is removed.

Conversion to PM. BwTree's non-SMOs are completed by preprending new delta node with a single `CAS`, so they fit into Condition #1. We perform a cache line flush if the `CAS` succeeds. BwTree's node split and merge mechanisms expose intermediate states to other readers and writers. While readers never restart in the original design of the BwTree, the open-source implementation of BwTree allows reads to restart if a node merge is in progress [71]. We address this issue by modifying the reader to avoid retry using the inconsistency detection and fix algorithm already present in the write path of BwTree.

Using their helper mechanism, the writers in BwTree detect and fix any partially completed operation. As a result, SMOs of BwTree (after modifications to the read operation) fits into Condition #2. We build P-BwTree by simply adding cache line flushes and memory fences after every `store` operation to the nodes and mapping table. Building P-BwTree involves modifying 85 LOC, as compared to 5.2K LOC in the core BwTree index.

6.4 Radix Tree: Adaptive Radix Tree (ART)

ART is a radix tree variant that reduces space consumption by adaptively varying node sizes based on the valid key entries. 8-bit prefix (one byte) is indexed by each node. The 8-byte header of each node in ART compresses some part of common prefix and the length of it. The level field in each node represents the full length of common prefix shared at this node and is never modified after its creation. As in HOT, synchronization is provided using non-blocking read and exclusive write [46, 47].

Non-SMOs. For an insertion, a new key-value pair is appended into the end of the entries in a node and is atomically made visible by increasing counter value. Deletion is completed via a single atomic store, simply invalidating a key by setting the value entry to be NULL. If the node overflows (or underflows), the node is copied to a new larger (or smaller) node and then the parent pointer is atomically swapped.

SMOs. If two keys share the same prefix, ART compresses the native radix tree structure by simply storing the common prefix in a single node (instead of allocating a node per character in the key). As key distribution varies, the compressed prefix could be expanded or compressed, resulting in split or merge of existing nodes. Unlike non-SMOs, these structural changes are installed in multiple atomic steps. If the insertion of a key requires a path compression split, a new node pointing to the key is first installed, and then the header is updated to contain the correct prefix.

Conversion to PM. Since non-SMOs are always committed atomically, they abide by Condition #1. However, the path compression mechanism in ART exposes intermediate states which reads can tolerate. A read counts the depth of the native decompressed radix tree while traversing tree, and compares level field with the sum of the depth and the prefix length stored in a node; if there is a mismatch, the read simply ignores a part of the prefix at this node to access the correct key. To ensure correctness, reads verify if the retrieved key is same as the search key before returning. Writes similarly detect inconsistencies, but do not fix them.

To build P-ART, we modify the write path to include crash detection and recovery. When the node traversal in the write path detects an inconsistency, it first checks for a crash using a `try lock`. If it successfully acquires the lock, the write calculates and persists the correct prefix. Implementing these changes, along with insertion of cache line flushes and memory fences required adding 52 LOC to the 1.5K LOC of ART.

6.5 Hybrid Index: Masstree

Masstree is a cache-efficient, highly concurrent trie-like concatenation of B+ tree nodes [52]. Masstree provides synchronization using write exclusion and lock-free readers retry when inconsistencies are detected by using version numbers.

Non-SMOs. Similar to ART, the non-SMOs of Masstree start with non-blocking tree traversal to return correct leaf node. Inserts to the leaf nodes in Masstree are performed by appending a new key-value pair to the node with unsorted order and atomically switching to an updated copy of the 8-byte permutation table, specifying the sorted orders of keys and empty entries. For deletes, it is sufficient to atomically update the permutation table to invalidate the entry.

SMOs. The internal nodes in Masstree maintain keys in sorted order using a non-atomic key-shifting algorithm which exposes inconsistent data to readers [9]. Reads therefore retry until the ongoing operation completes. Node splits and merges lock the corresponding nodes and update version counters upon completion. Meanwhile, all concurrent reads and writes to these nodes would simply retry from the root.

Conversion to PM. The non-SMOs of Masstree abide by Condition #1, since insertions and deletions are atomically reflected by updating a permutation table. However, Masstree SMOs do not directly fit into our conditions as the readers do

Workload	Description	Application pattern
Load A	100% writes	Bulk database insert
A	Read/Write, 50/50	A session store
B	Read/Write, 95/5	Photo tagging
C	100% reads	User profile cache
E	Scan/Write, 95/5	Threaded conversations

Table 3. YCSB workload patterns. The table describes different workload patterns from the YCSB test suite.

not tolerate inconsistency without restarts. While the structure of leaf nodes allows a 2-step atomic split mechanism, the internal nodes do not. Therefore, we modify the internal nodes to resemble the leaf nodes, modifying the data structure to resemble the B-link Tree. This modification allows a 2-step atomic split mechanism across all levels. For example, if the insertion requires node split, half of the entries in split node are copied into the new sibling node, and then the sibling pointer of split node is atomically installed to the new sibling. Finally, the entries copied into new sibling node are atomically invalidated from split node by updating 8-byte permutation table. Furthermore, this eliminates restarts at the read path. All the intermediate states exposed by SMOs are tolerated by moving towards next sibling node, utilizing the B-link Tree's sibling link and high key [6]. Reads therefore always return consistent data and writes can reach the correct leaf node without retry.

With this change, SMOs of Masstree fits into Condition #3, where reads return consistent values, but writers have no mechanism to fix inconsistent states. We implement write path recovery by simply replaying the node split algorithm whenever a crash is detected using a `try lock`. If the intermediate state observed was due to a node split, then this action would complete the split operation. If the observed crash state was due to a node merge, replaying the split operation will undo the merge, bringing the index back to a consistent state.

7 Evaluation

We evaluate the performance of indexes converted using the Recipe approach against state-of-the-art hand-crafted PM indexes on Intel Optane DC Persistent Memory Module (PMM). The experiments are performed on a 2-socket, 96-core machine with 768 GB PMM, 375 GB DRAM, and 32 MB Last Level Cache (LLC). We use the ext4-DAX file system running kernel 4.17 on the Fedora distribution. All our experiments are performed in the *App Direct* mode of Optane DC which exposes a separate persistent memory device [32]. All experiments are performed on a single socket by pinning threads to a local NUMA node. Since the machine supports `clwb` instruction which is more efficient than `clflush`, we use `clwb` for cache line flushes in our experiments.

We split our evaluation based on the data structure into ordered indexes and unordered indexes. An ordered index aims to support both point and range queries, but an unordered index only provides point queries. FAST & FAIR, P-Bw tree, P-Masstree, P-ART, and P-HOT are the ordered indexes, while CCEH, Level Hashing, and P-CLHT are the unordered indexes. We use the `libvmmalloc` library from PMDK that transparently converts traditional dynamic allocation interfaces to work on a volatile memory pool built on a memory-mapped file on PMEM [59]. We further collect low-level performance counters such as the number of `clwb` and `mfence` instructions along with the number of LLC misses per operation using the `perf` tool.

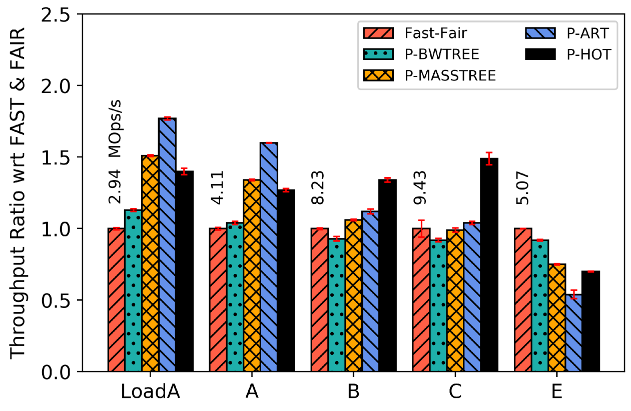
Workloads. We use the Yahoo! Cloud Serving Benchmark (YCSB) [14], the industry standard for evaluating key-value indexes. We use the index micro-benchmark to generate workload files for YCSB and statically split them across multiple threads [76]. Each generated workload mimics a real application pattern as shown in Table 3. We exclude workloads D and F as they involve updates and some indexes (FAST & FAIR, CCEH, CLHT) do not support key updates. For each workload, we test two key types - **randint** (8 byte random integer keys) and **string** (24 byte YCSB string keys), all uniformly distributed.

To evaluate the ordered indexes, we use both random integer and string type keys. As the open-source implementation of FAST & FAIR does not support string type keys, we implement string type support for FAST & FAIR by replacing integer key entries with pointers to the address of the actual string key, which is simplest way to support variable-sized string-type keys in B+tree in a crash-safe manner [9]. In both cases, we first populate the index with 64M keys using Load A, and then run the respective workloads that insert or read a total of 64M keys. For unordered indexes, we only use integer key types. We present the results from multi-threaded workloads using 16 threads and omit single threaded results as the performance trends are comparable to the multi-threaded workload. We use the default node size for each of the tree-based indexes, and a starting hash table size of 48KB. The reported numbers are averaged over several runs (with an average variance of 0.1%).

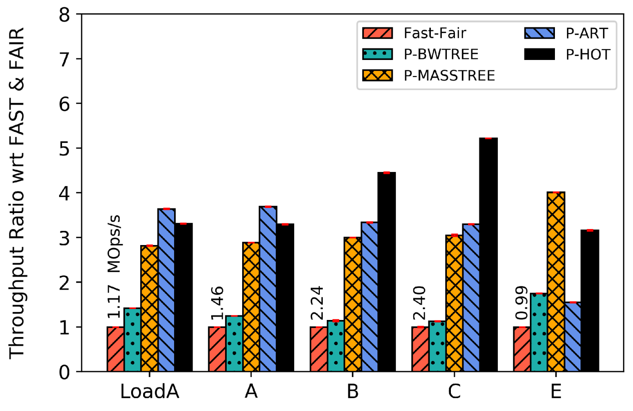
7.1 Ordered indexes

We evaluate converted indexes P-ART, P-HOT, P-Masstree, and P-BwTree against the only concurrent and open-source state-of-the-art PM B+ tree, FAST & FAIR.

Integer type keys. P-ART outperforms FAST & FAIR by up to 1.6× on write-heavy workloads as in Fig 4 a. The FAST algorithm sorts inserted keys in-place, which results in higher number of cache line flushes as compared to P-ART. This explains the lower performance of FAST & FAIR



(a) Integer keys : Multi threaded YCSB



(b) String keys : Multi threaded YCSB

PM Index	Instructions		Last Level Cache Miss				
	`clwb`	`mfence`	LoadA	A	B	C	E
FAST & FAIR	7	8	11	10	8	7	8
P-Bw Tree	7	4	17	15	10	9	26
P-Masstree	3	5	7	7	6	5	8
P-ART	3	3	4	4	4	4	12
P-HOT	7	5	4	4	2	2	10

(c) Integer keys : Performance counters

PM Index	Instructions		Last Level Cache Miss				
	`clwb`	`mfence`	LoadA	A	B	C	E
FAST & FAIR	8	10	36	47	40	39	76
P-Bw Tree	8	6	40	48	39	37	62
P-Masstree	4	7	9	10	8	7	11
P-ART	3	4	4	5	5	5	22
P-HOT	7	5	5	5	3	3	12

(d) String keys : Performance counters

Figure 4. YCSB workload for tree indexes. The plot compares the performance of various tree based PM indexes using YCSB workloads (higher is better). All the indexes converted using Recipe outperform FAST & FAIR, the state-of-the-art B+ tree by up to 5× for string keys. For integer keys, FAST & FAIR has better range scan performance. The fine grained performance counters per operation help explain the observed trends (lower is better).

in write-intensive workloads. Trie-based indexes like P-HOT eliminate key comparisons in their search path as they do not store full keys in internal nodes. Therefore, point reads are more cache-efficient (P-HOT incurs 3× lower LLC misses compared to FAST & FAIR), thereby outperforming FAST & FAIR by 1.5× on read intensive workloads.

The performance of FAST & FAIR and P-BwTree is similar. P-BwTree performance is low because its operations require pointer chasing; for example, an insert can be only be performed after applying prior deltas. This leads to many LLC misses. As a result, P-BwTree performance is not significantly better than B+ trees with in-place updates.

FAST & FAIR outperforms all other indexes in range scans. There are two primary reasons for this. First, the keys are more compactly packed into nodes in B+ trees unlike tries, which makes it cache efficient in range scans. Second, the leaf nodes do not have sibling pointers in prefix tries, thereby requiring extensive traversals for range queries.

String type keys. The absolute value of throughput decreases for string key types as compared to randint keys for all indexes. However, the magnitude of performance drop is the highest for FAST & FAIR and native B+ trees, due to the high cost of string key comparison and pointer dereference to access the string key. This results in 8× more LLC misses in average as compared to prefix tries. Comparing absolute throughput, we see that FAST & FAIR performs 2.5 – 5× worse for all YCSB workloads using string type keys as compared to integer keys. Whereas, prefix tries are only about 20% slower when switched to the string keys.

As shown in fig 4b, B+ tree's cache inefficiency results in 3.2 – 5.2× worse performance compared to P-HOT. We observe that although Masstree uses a data structure that is a combination of B+ trees and prefix tries, its trie-based structure enables native key comparison by storing 8-byte partial keys to each B+tree's layer. Furthermore, it uses a collection of cache-friendly techniques such as prefetching, reduced tree depth, and careful layout of data across cachelines. These design choices makes Masstree better than its B+ tree counterparts across all workloads.

7.2 Unordered indexes

We evaluate P-CLHT against two state-of-the-art persistent hash tables, CCEH and Level hashing. Fig 5 shows that

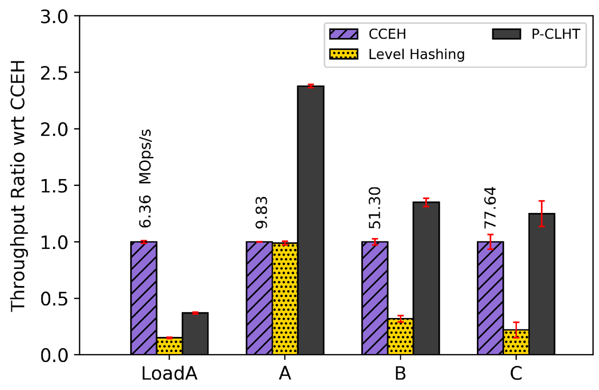


Figure 5. YCSB workload with integer keys for hash tables. The figure compares the performance of various hash based PM indexes using YCSB workloads (higher is better). P-CLHT, the index converted using RECIPE outperforms CCEH, the state-of-the-art hash table, by up to 2.4×.

P-CLHT outperforms CCEH by up to 2.5× on the multi-threaded YCSB workload. Starting from a hash table size of 48KB, we insert 64M keys into the hash table in Load A, which triggers multiple re-hashing operations in both indexes. P-CLHT is 2× worse than CCEH for concurrent insert only workload, due to the globally-locked rehashing scheme that throttles concurrency. We confirm this by evaluating the two indexes using a single thread, where P-CLHT is only 12% slower than CCEH even in the presence of rehashing.

Table 4 shows that CCEH has lower throughput than P-CLHT though both similar number of cache misses and `clwb` instructions. This is due to the segment split mechanism of CCEH. When the hash table is sufficiently large, P-CLHT performs no rehashing (in workload A and B), thereby requiring only one `clwb` per insert. On the other hand, even when similarly sized, CCEH performs frequent segment splits that require multiple cache line flushes and expensive copy-on-write of new segments (117K segment splits occurred on inserting 10M keys into a sufficiently large hash table). CCEH requires additional pointer reads due to indirections introduced using directory and segment, which results in lower read performance over P-CLHT. Level hashing incurs a higher number of cache misses due to its two level architecture that results in non-contiguous cache line accesses [55] and lower throughput.

7.3 Comparison to WOART

WOART [44] is a single-threaded, hand-crafted, write-optimal PM variant of ART. WOART introduces a new recovery mechanism and modifies the node structure to be failure-atomic. The authors suggest modifying WOART to be multi-threaded using a global lock; since this leads to low concurrency, P-ART outperforms WOART on multi-threaded YCSB workloads by by $2 - 20\times$.

PM Index	Instructions		Last Level Cache Miss			
	`clwb`	`mfence`	LoadA	A	B	C
CCEH	2.3	3.0	1.5	1.5	1.1	1.0
Level hashing	3.7	5.8	4.0	3.3	4.0	4.0
P-CLHT	1.5	2.5	2.4	1.3	1.1	1.1

Table 4. Performance counters. The table shows the average number of `clwb`, `mfence` instructions per insert operation, and the average number of LLC misses per operation during each workload for randint keys (lower is better).

7.4 Summary

Recipe-converted indexes outperform state-of-the-art hand-crafted PM indexes by up-to 5.2× on multi-threaded YCSB workloads. Recipe-converted indexes are optimized for cache-efficiency and concurrency as they are built from mature DRAM indexes. Recipe-converted indexes encounter fewer cache misses as compare to hand-crafted PM indexes. The append-only nature of indexes like P-ART results in up-to 2× lower cache line flushes, compared to hand-crafted PM indexes like FAST & FAIR. All these factors contribute to the performance gain of Recipe-based PM indexes.

7.5 Testing Crash Recovery

We test each index for 10K crash states. We load 10K entries into the index, allowing it to crash probabilistically. We then perform a mixed workload consisting of a total of 10K inserts and reads into the index using 4 concurrent threads. Finally, we read back all successfully inserted keys from the index. On average, the end-to-end time for generating a crash state and testing it is 20ms.

We tested the current state-of-the-art PM indexes, and our converted PM indexes using the approach outlined in Section 5. Our testing revealed crash-consistency bugs in FAST & FAIR and CCEH. In FAST & FAIR, when two consecutive crashes occur during a node split and a node merge, the node to be deleted by the merge algorithm is not cleaned up correctly, which makes its right sibling inaccessible by a reader. This results in data loss. CCEH results in stalled operations if a crash occurs during directory doubling, as it does not update directory metadata atomically. All PM indexes converted using Recipe passed the testing with no bugs. Additionally, our durability test reveals that the initial node allocation containing the root pointer is not persisted in FAST & FAIR and CCEH.

8 Discussion

Optimization. We can increase the performance of converted PM indexes by reducing the number of cache line flushes or memory fences using techniques like persist buffering and coalescing [58]. Persistent buffering reduces the excessive flush and fence overhead by allowing flushes between independent cache lines to be reordered. Persistent coalescing facilitates batching multiple cache line flushes to the same cache line [12, 13]. Recipe-based conversion inserts a flush and fence operation after each store. We optimized this by buffering and coalescing the flushes wherever possible in our Recipe-converted indexes presented in Section 6; however, such optimizations turned out to be heavily dependent on the implementation of the index structure. As we could not generalize these optimizations into conditions, Recipe leaves it to the developer to identify and apply them.

Automation. Converting indexes using Condition #1 and #2 only requires cache line flushes and memory fences after every `store` instruction. Although this sounds simple and easy to automate, the challenge in automating these conversions lies in the many different ways in which the same logical steps are implemented in different indexes. For example, an atomic `store` operation could be implemented using the C++ atomic library, or through a simple pointer assignment, followed by `mfence`.

9 Related Work

Isolation and Crash Recovery. Memory Persistency [58] makes the connection between crash recovery and the semantics of memory consistency by introducing the concept of Recovery Observer. Durable Linearizability [37] and Recoverable Linearizability [3] theoretically define the relationship between crash recovery and non-blocking synchronization. However, these works only propose model semantics, without connecting the findings to practical index structures.

TSP [56] proposes the broad insight that non-blocking indexes can be converted into crash-consistent counterparts by coupling Recovery Observer and Flush-on-Failure. However, Flush-on-Failure technique requires additional hardware support like the backup power supply and kernel modifications. Recipe, on the other hand, exploits and extends these broad observations to build concurrent, crash-consistent PM indexes without any hardware support and kernel changes. While TSP assumes non-blocking writes, Recipe relaxes the assumption, allowing write exclusion (which most concurrent DRAM indexes use).

Concurrent Persistent Indexes. In the past five years, 15 PM indexes have been proposed, out of which only three have open source, concurrent implementations: FAST & FAIR, CCEH, and Level Hashing. Recipe is complementary to these efforts in building a concurrent PM index. Recipe takes a more principled approach by reusing decades of research in building concurrent in-memory indexes with no modifications to the underlying design of the DRAM index.

Transactional PM Systems. Previous work like Atlas [7], JUSTDO [36], NVThreads [30], and iDO [49], persist data at boundaries of critical sections called Failure Atomic SEctions (FASE). They automatically inject logging for every persistent update [7, 30] or program states [36, 49] within FASE by using compile-time analysis. However, their approaches amplify the overhead of cache line flushes, as they require an additional persistent log. These systems also pay a startup cost to replay the log during recovery, which could be significant for large indexes. However, Recipe-converted indexes do not employ additional logging mechanisms and pay no startup recovery cost when the index restarts after a crash.

Crash-Consistency Testing for PM Applications. PM application testing frameworks such as Yat [42], Intel PM-Inspector [34], and pmreorder [60] aim at enabling correctness testing and debugging for applications built for PM. However, these tools use either random or exhaustive techniques to construct crash states, which does not scale as the number of writes to the PM increase [34, 42, 60]. Our crash testing strategy, on the other hand, exploits the fact that operations in PM indexes are comprised of a small set of atomic steps, thereby simulating crashes only after these atomic steps. This technique makes our approach efficient and powerful enough to reveal bugs within a few crash states. PMTest [50] requires that developers manually annotate their source code with assert-like statements to find errors [50]. However, our approach requires lower effort from developers, since changes are localized to the write path.

10 Conclusion

This paper presents Recipe, a principled approach to convert concurrent in-memory indexes to be persistent. Recipe exploits the relationship between isolation provided by concurrent DRAM indexes and crash recovery. Recipe provides three conditions to identify DRAM indexes that can be converted to PM in a principled manner, and corresponding conversion actions. Using Recipe, we convert five DRAM indexes, all based on different data structures to their PM counterparts. When evaluated on Intel DC Persistent Memory, our converted indexes outperform state-of-the-art hand-crafted PM indexes by as much as 5.2×. Recipe-converted indexes are publicly available at https://github.com/utsaslab/RECIPE.

Acknowledgments

We would like to thank our shepherd, Peter Chen, the anonymous reviewers, and members of Systems and Storage Lab and LASR group for their feedback and guidance. This work was supported by NSF CAREER #1751277 and donations from VMware, Google and Facebook. We would like to thank Intel and ETRI IITP/KEIT[2014-3-00035] for providing access to Optane DC Persistent Memory to perform our experiments.

References

[1] S. V. Adve and K. Gharachorloo. 1996. Shared memory consistency models: a tutorial. *Computer* 29, 12 (Dec 1996), 66–76. https://doi.org/10.1109/2.546611

[2] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-performance Latch-free Range Index for Non-volatile Memory. *Proceedings of the VLDB Endowment* 11, 5 (Jan. 2018), 553–565. https://doi.org/10.1145/3187009.3164147

[3] Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. 2016. Robust Shared Objects for Non-Volatile Main Memory. In *19th International Conference on Principles of Distributed Systems (OPODIS 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 46. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1–17. https://doi.org/10.4230/LIPIcs.OPODIS.2015.20

[4] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 521–534. https://doi.org/10.1145/3183713.3196896

[5] Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. 2015. Help!. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing (PODC '15)*. ACM, New York, NY, USA, 241–250. https://doi.org/10.1145/2767386.2767415

[6] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 181–190. http://dl.acm.org/citation.cfm?id=645927.672375

[7] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224

[8] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. 2001. Improving Index Performance Through Prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD '01)*. ACM, New York, NY, USA, 235–246. https://doi.org/10.1145/375663.375688

[9] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-volatile Main Memory. *Proceedings of the VLDB Endowment* 8, 7 (Feb. 2015), 786–797. https://doi.org/10.14778/2752939.2752947

[10] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture. *Proceedings of the VLDB Endowment* 1, 2 (Aug. 2008), 1313–1324. https://doi.org/10.14778/1454159.1454171

[11] Ping Chi, Wang-Chien Lee, and Yuan Xie. 2014. Making B+-tree Efficient in PCM-based Main Memory. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design (ISLPED '14)*. ACM, New York, NY, USA, 69–74. https://doi.org/10.1145/2627369.2627630

[12] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. 2019. Fine-Grain Checkpointing with In-Cache-Line Logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 441–454. https://doi.org/10.1145/3297858.3304046

[13] Nachshon Cohen, Michal Friedman, and James R. Larus. 2017. Efficient Logging in Non-volatile Memory by Exploiting Coherency Protocols. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 67 (Oct. 2017), 24 pages. https://doi.org/10.1145/3133891

[14] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[15] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. 631–644. https://doi.org/10.1145/2694344.2694359

[16] Tudor Alexandru David. 2017. *Universally Scalable Concurrent Data Structures*. Ph.D. Dissertation. EPFL.

[17] Tudor Alexandru David, Rachid Guerraoui, Tong Che, and Vasileios Trigonakis. 2014. *Designing ASCY-compliant Concurrent Search Data Structures*. Technical Report.

[18] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[19] Jose M. Faleiro and Daniel J. Abadi. 2017. Latch-free synchronization in database systems: Silver bullet or fool's gold?. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR '17)*. 9.

[20] Panagiota Fatourou, Nikolaos D Kallimanis, and Thomas Ropars. 2018. An Efficient Wait-free Resizable Hash Table. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. ACM, 111–120.

[21] Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.599193

[22] Keir Fraser and Tim Harris. 2007. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)* 25, 2 (2007), 5.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*. 29–43. https://doi.org/10.1145/945445.945450

[24] R. Hagmann. 1987. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP '87)*. ACM, New York, NY, USA, 155–162. https://doi.org/10.1145/41457.37518

[25] Maurice Herlihy. 1991. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 124–149.

[26] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2003. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*. IEEE Computer Society, 522–529. https://doi.org/10.1109/ICDCS.2003.1203503

[27] Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann.

[28] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[29] Dave Hitz, James Lau, and Michael A. Malcolm. 1994. File System Design for an NFS File Server Appliance. In *USENIX Winter 1994 Technical Conference, San Francisco, California, USA, January 17-21, 1994, Conference Proceedings*. USENIX Association, 235–246. https://www.usenix.org/conference/usenix-winter-1994-technical-conference/file-system-design-nfs-file-server-appliance

[30] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 468–482.

[31] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 187–200. https://www.usenix.org/conference/fast18/presentation/hwang

[32] Alper Ilkbahar. 2019. *Intel Optane DC Persistent Memory Modes*. https://itpeernetwork.intel.com/intel-optane-dc-persistent-memory-operating-modes/#gs.6518fh

[33] Intel. 2019. Intel 64 and IA-32 Architectures Software Developers Manual Combined Volumes. https://software.intel.com/en-us/articles/intel-sdm.

[34] Intel. 2019. *Intel Inspector*. https://software.intel.com/en-us/get-started-with-inspector

[35] Intel. 2019. *Intel Optane DC Persistent Memory*. https://newsroom.intel.com/news-releases/intel-data-centric-launch/#gs.7kv3ru

[36] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. 2016. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 427–442. https://doi.org/10.1145/2872362.2872410

[37] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Proceedings of 30th International Symposium on Distributed Computing, DISC 2016, Paris, France, September 27-29, 2016*. 313–327. https://doi.org/10.1007/978-3-662-53426-7_23

[38] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[39] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. 2013. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 1173–1184. https://doi.org/10.1145/2463676.2465293

[40] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[41] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. 2018. clfB-tree: Cacheline friendly persistent B-tree for NVRAM. *ACM Transactions on Storage (TOS)* 14, 1 (2018), 5.

[42] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A Validation Framework for Persistent Memory Software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 433–438. https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz

[43] Se Kwon Lee. 2019. *BUG fix : FAIR algorithm*. https://github.com/DICL/FAST_FAIR/pull/4

[44] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 257–270. https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon

[45] Philip L. Lehman and S. Bing Yao. 1981. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 650–670. https://doi.org/10.1145/319628.319663

[46] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

[47] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*. ACM, 3.

[48] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[49] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L. Scott, Sam H. Noh, and Changhee Jung. 2018. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 258–270.

[50] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, NY, USA, 411–425. https://doi.org/10.1145/3297858.3304015

[51] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 190–200. https://doi.org/10.1145/1065010.1065034

[52] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 183–196.

[53] Paul E. McKenney and John D. Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*. 509–518.

[54] Memcached. 2011. *A distributed memory object caching system*. http://memcached.org

[55] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 31–44. https://www.usenix.org/conference/fast19/presentation/nam

[56] Faisal Nawab, Dhruva R. Chakrabarti, Terence Kelly, and Charles B. Morrey III. 2015. Procrastination Beats Prevention: Timely Sufficient Persistence for Efficient Crash Resilience. In *Proceedings of the 18th International Conference on Extending Database Technology (EBDT '15)*. 689–694.

[57] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. 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)*. ACM, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

[58] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*. IEEE Press, 265–276.

[59] PMDK. 2019. *The libvmmalloc library*. http://pmem.io/pmdk/libvmmalloc/

[60] PMDK. 2019. *pmreorder*. http://pmem.io/pmdk/manpages/linux/master/pmreorder/pmreorder.1.html

[61] William Pugh. 1989. Skip Lists: A Probabilistic Alternative to Balanced Trees. In *Algorithms and Data Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings*. 437–449. https://doi.org/10.1007/3-540-51542-9_36

[62] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *VLDB'99, Proceedings of 25th*

International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK. 78–89. http://www.vldb.org/conf/1999/P7.pdf

[63] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. 1–16.

[64] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-hashmap: A persistent and concurrent hashmap for non-volatile memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*. ACM, 4.

[65] Håkan Sundell and Philippas Tsigas. 2003. Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*. IEEE Computer Society, 84. https://doi.org/10.1109/IPDPS.2003.1213189

[66] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. 2011. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *2011 USENIX Annual Technical Conference, Portland, OR, USA, June 15-17, 2011*. https://www.usenix.org/conference/usenixatc11/resizable-scalable-concurrent-hash-tables-relativistic-programming

[67] Philippas Tsigas and Yi Zhang. 2001. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 29. ACM, 320–321.

[68] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 18–32.

[69] John D. Valois. 1995. Lock-Free Linked Lists Using Compare-and-Swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*. 214–222. https://doi.org/10.1145/224964.224988

[70] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 461–472.

[71] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. 2018. Building a Bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 473–488.

[72] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. 2016. Nvmcached: An nvm-based key-value cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. ACM, 18.

[73] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 349–362. https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia

[74] Zhongle Xie, Qingchao Cai, Gang Chen, Rui Mao, and Meihui Zhang. 2018. A Comprehensive Performance Evaluation of Modern In-Memory Indices. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 641–652. https://doi.org/10.1109/ICDE.2018.00064

[75] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 167–181. https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang

[76] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1567–1581. https://doi.org/10.1145/2882903.2915222

[77] Pengfei Zuo and Yu Hua. 2017. A write-friendly hashing scheme for non-volatile memory systems. In *Proceedings of the 33st Symposium on Mass Storage Systems and Technologies (MSST '17)*.

[78] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 461–476. https://www.usenix.org/conference/osdi18/presentation/zuo