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

1 Introduction

The low latency and durability of Persistent Memory (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. However, 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 system crash. This complexity leads to subtle bugs [1, 4, 6].

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. Modern DRAM indexes are carefully designed keeping in mind cache efficiency, pre-fetching, concurrency, and parallelism. Concurrent DRAM indexes are widely used in industry and academia; for example, latch-free BwTree in the Hekaton OLTP engine, Adaptive Radix Tree (ART) in the HyPer database, the Timeline Index in SAP HANA, and Masstree in the Silo database. In this work, we seek to leverage the research on concurrent DRAM indexes to build crash-consistent PM indexes.

We present Recipe [6], a principled approach for converting concurrent DRAM indexes into crash-consistent indexes for PM. The main insight behind Recipe is that *isolation* provided by a certain class of concurrent DRAM 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 a hash table (CLHT [7]), a trie (HOT [5]), a B+ tree (BwTree [2]), a radix tree (ART [8]), and a hybrid index (Masstree [9]) to their crash consistent PM counterparts. The effort involved in this conversion is minimal, requiring 30–200 lines of code (1–9% of the codebase). We evaluated the converted PM indexes on Intel DC Persistent Memory, and found that they outperform state-of-the-art, hand-crafted PM indexes [1, 4] in multi-threaded workloads by up-to 5.2×. Recipe-converted indexes are available at https://github.com/utsaslab/RECIPE.

2 The Recipe Approach

Recipe provides 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 indexes are recoverable from crashes correctly. Thus, if a 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 a conversion action. We first present the intuition behind the Recipe approach, and then describe each category.

2.1 Overall Intuition

We observe that some DRAM indexes use non-blocking 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. Similarly, write operations may also see an inconsistent state and *fix* the inconsistency. Prior theoretical work has termed this a *helping mechanism*, where an operation started by one thread which fails is later completed by another thread [3].

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.

2.2 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.

2.3 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`. For non-blocking writes, if a mismatch between the store orders to CPU cache and PM can occur, each `load` should also be followed by cache line flush and memory fence instructions.

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

2.4 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` and specific `load` instructions to prevent concurrent threads from acting on stale or unpersisted information.

Example (P-BwTree). 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 helping 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 by adding cache line flushes and memory fences after each store and load in helping mechanisms.

2.5 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 helping mechanisms needed to fix the inconsistency.

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

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

3 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). For workloads, we use the Yahoo! Cloud Serving Benchmark (YCSB), the industry standard for evaluating key-value indexes.

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 compared 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 [1]. All these factors contribute to the performance gain of Recipe-based PM indexes.

References

[1] Deukyeon Hwang, et al. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *FAST 2018*.

[2] Justin J. Levandoski, et al. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE 2013*.

[3] Keren Censor-Hillel, et al. Help!. In *PODC 2015*.

[4] Moohyeon Nam, et al. Write-Optimized Dynamic Hashing for Persistent Memory. In *FAST 2019*.

[5] Robert Binna, et al. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD 2018*.

[6] Se Kwon Lee, et al. Recipe: Converting Concurrent DRAM Indexes to Persistent-memory Indexes. In *SOSP 2019*.

[7] Tudor David, et al. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *ASPLOS 2015*.

[8] Viktor Leis, et al. The ART of practical synchronization. In *DaMoN 2016*.

[9] Yandong Mao, et al. Cache craftiness for fast multicore key-value storage. In *EuroSys 2012*.