Crash Recoverable ARMv8-Oriented B+-Tree for Byte-Addressable Persistent Memory

Chundong Wang
Singapore University of Technology and Design
Singapore
cd_wang@outlook.com

Sudipta Chattopadhyay
Singapore University of Technology and Design
Singapore
sudipta_chattopadhyay@sutd.edu.sg

Gunavaran Brihadiswarn*
University of Moratuwa
Sri Lanka
gunavaran.15@cse.mrt.ac.lk

Abstract

The byte-addressable non-volatile memory (NVM) promises persistent memory. Concretely, ARM processors have incorporated architectural supports to utilize NVM. In this paper, we consider tailoring the important B+-tree for NVM operated by a 64-bit ARMv8 processor. We first conduct an empirical study of performance overheads in writing and reading data for a B+-tree with an ARMv8 processor, including the time cost of cache line flushes and memory fences for crash consistency as well as the execution time of binary search compared to that of linear search. We hence identify the key weaknesses in the design of B+-tree with ARMv8 architecture. Accordingly, we develop a new B+-tree variant, namely, crash recoverable ARMv8-oriented B+-tree (Crab-tree). To insert and delete data at runtime, Crab-tree selectively chooses one of two strategies, i.e., copy on write and shifting in place, depending on which one causes less consistency cost to performance. Crab-tree regulates a strict execution order in both strategies and recovers the tree structure in case of crashes. We have evaluated Crab-tree in Raspberry Pi 3 Model B+ with emulated NVM. Experiments show that Crab-tree significantly outperforms state-of-the-art B+-trees designed for persistent memory by up to 2.6× and 3.2× in write and read performances, respectively, with both consistency and scalability achieved.

CCS Concepts • **Information systems → B-trees**; **Storage class memory**; • **Software and its engineering → Consistency**; • **Hardware → Non-volatile memory**.

*This work was done when Gunavaran Brihadiswarn was an intern in Singapore University of Technology and Design.

LCTES '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6724-0/19/06…$15.00
https://doi.org/10.1145/3316482.3326358

Keywords B+-tree, Non-volatile Memory, ARMv8, Persistent Memory

ACM Reference Format:
Chundong Wang, Sudipta Chattopadhyay, and Gunavaran Brihadiswarn. 2019. Crash Recoverable ARMv8-Oriented B+-Tree for Byte-Addressable Persistent Memory. In *Proceedings of the 20th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3316482.3326358

1 Introduction

The next-generation non-volatile memory (NVM) technologies, such as Magnetoresistive RAM (MRAM) [1], Resistive RAM (Re-RAM) [2, 3], 3D XPoint [4] and Phase-Change memory (PCM) [5, 6], emerge as a new tier in the memory and storage hierarchy. They have DRAM-like byte addressability and flash memory-like durability. Researchers have proposed to place them on the processor-memory bus to build *persistent memory* that blurs the boundary between memory and storage. A number of system and application softwares have been designed to leverage the persistent memory in x86-based computing platforms [7–21].

NVM technologies are generally denser and more energy-efficient than DRAM. They also have shorter access latency and higher write endurance than flash memory. Bringing NVM into embedded systems is likely to gain benefits for system performance and capacity, energy consumption and device lifetime. ARM architectures have recently started to provide architectural supports for persistent memory like x86, particularly for 64-bit ARMv8 [22, 23]. It might seem natural to adopt the scientific works that deploy NVM into x86-based system for ARMv8-based system. In this paper, we challenge such an approach and show that deploying NVM with ARMv8 requires rethinking and redesign of critical data structures. Concretely, we presume an embedded system equipped with an ARMv8-A 64-bit processor and byte-addressable NVM. We place our emphasis on tailoring a crucial structure, i.e., B+-tree [24–31], for such a system.

The differences between x86 and ARMv8 introduce new challenges in designing in-NVM B+-tree. First, ARMv8 processors that are widely used in smartphones and Raspberry

Pi have relatively fewer computational resources. For example, a commonplace x86 processor installed in desktop computers can have a shared L3 cache of 8192KB while the ARMv8 processor of the latest Raspberry Pi just has a shared L2 cache of 512KB. Different computational resources lead to different write and read performance for B+-tree. Secondly, although the time cost of using cache line flushes and memory fences has been known for x86 processors [25, 28, 32–36], such cost is not thoroughly evaluated for ARMv8. We note that cache line flushes and memory fences are required for the crash consistency of an in-NVM data structure, i.e., to correctly recover it after a system crash. Processors or memory controllers may alter the programmed order of writing multiple cache lines from CPU cache to memory for high memory bandwidth [7, 28, 32, 33, 36–38]. Cache line flush explicitly flushes a dirty cache line to memory while memory fence acts as a barrier enforcing that memory operations after the barrier to stall until operations before the barrier complete. A series of cache line flushes and memory fences thus secures a desired writing order to NVM and maintains crash consistency. Thirdly, x86 processors guarantee total store ordering (TSO) [39] that prevents store-after-store instructions from being reordered, but ARMv8 with non-TSO may reorder independent store instructions. TSO cannot avoid multiple cache lines being written back to memory in an altered order, but it helps x86-based system to waive the use of memory fences in consecutive stores that demand an order. State-of-the-art design of in-NVM B+-tree exploits the TSO to improve the performance of B+-tree, but this assumption completely breaks when moving to an ARM-based system, with inferior write performance reported [28].

To evaluate these concerns, we have done an empirical study with an ARMv8-A processor. The study reveals several inspiring observations and opens the door of a plethora of research opportunities in deploying NVM with ARMv8-based systems. Specifically, we have the following observations:

- Using cache line flushes and memory fences to preserve a writing order of dirty cache lines is significantly time consuming with ARMv8. For example, the insertion performance dropped by 5.1× when cache line flushes and memory fences were enabled for B+-tree.
- Interposing extra memory fences to counteract non-TSO incurs substantial performance overhead. In particular, the insertion performance further dropped by 3.1× after adding such memory fences.
- Flushing multiple cache lines of a contiguous memory space in a batch ended by one eventual memory fence improves performance by 2.7× compared to flushing each of them through one cache line flush and memory fence.

In line with these observations, we have designed a new **c**rash **r**ecoverable **A**RMv8-oriented **B**+-tree (Crab-tree). Crab-tree keeps all keys sorted in a tree node. It only considers crash consistency of leaf nodes (LNs), as upper-level index nodes (INs) can be reconstructed from LNs. The key ideas of Crab-tree to suit ARMv8 are summarized as follows.

- Crab-tree uses two consistency strategies: 1) copy on write (COW) copies key-value (KV) pairs from an LN to a new LN, flushing them in a batch with one ending memory fence, and then replaces the original LN with the new LN, and 2) shifting in place (SIP) shifts KV pairs in the original LN, flushing dirty cache lines and counteracting non-TSO with memory fences where necessary.
- Crab-tree selects a strategy when it receives an insertion or deletion request. Given a key to be inserted or deleted, Crab-tree estimates the consistency cost of COW and SIP. It chooses one strategy that incurs less cost.
- For both COW and SIP, Crab-tree enforces a strict modification and store order, and always ends an insertion or deletion transaction with an 8B atomic write supported by ARMv8. In such a fashion, Crab-tree is able to identify and recover the exact inconsistent LN after a crash.

COW and SIP complement each other. COW is applied if using SIP has to shift a substantial number of KV pairs. SIP is chosen when shifting only a few KV pairs in-situ can complete an insertion or deletion. We have prototyped Crab-tree in a Raspberry Pi 3 Model B+ with ARM Cortex-A53 and emulated NVM. Experiments show that Crab-tree significantly outperforms state-of-the-art B+-trees for persistent memory by up to 2.6× and 3.2× in write and read performances, respectively, without losing consistency or scalability.

The remainder of this paper is organized as follows. In Section 2 we show the background of NVM and state-of-the-art in-NVM B+-tree variants. In Section 3 we present our empirical study. In Section 4 we detail the design of Crab-tree. In Section 5 we evaluate Crab-tree and conclude the paper in Section 6.

2 Background and Related Work

2.1 NVM and Architectural Supports

Front runners of NVM technologies include MRAM, ReRAM, 3D XPoint and PCM. In order to exploit the superb characteristics of NVM, such as the byte-addressability, non-volatility and relatively short access latency, a number of proposals that optimize, design or even revolutionize computer systems with NVM have been considered from micro-architectural level to application level [2, 5, 7, 9, 11, 12, 14–19, 32, 40–53]. One such approach that works with existing architectural support is to build persistent memory by placing NVM on the memory bus, either with or without DRAM. In this fashion, processors directly load and store data with NVM.

Most of the proposals to deploy NVM are x86-oriented due to the maturity of architectural supports in x86 for NVM. ARM architecture has started to provide instructions for cache line flushes[1] (e.g., `dc civac`, `dc cvac` and `dc cvap`)

[1] These instructions are called cache line *clean* in ARMv8 manual [22]. We refer to them as cache line flush as regards the convention of x86 processors.

and memory fences (e.g., `dmb`) since ARMv8. These instructions help programmers to guarantee the consistency and persistency of data stored in NVM when developing system- and application-level softwares for embedded systems.

Special architectural supports are necessitated because persistent memory blurs the boundary between memory and storage. Traditional main memory is volatile by the very nature of DRAM while data storage relies on block-based devices like hard disk or SSD [54]. Operating data structures in the persistent memory is non-trivial and needs to take into account characteristics of both memory and storage. For example, when processors store data into memory, they only guarantee at most 8B of data to be atomically (i.e., all-or-nothing) written. By contrast, a hard disk supports a sector-level (e.g., 512B) atomic write [9, 25, 28, 55].

Moreover, multiple writes to the persistent memory may not occur in the same order as they are written by an arbitrary program running on the CPU. While processor preserve the functional correctness of the program, there might be many in-flight writes to the persistent memory that break the order in which they occurred during program execution [7, 10, 28, 33, 37]. If all such in-flight writes complete, the program runs as expected. However, if a system crash happens before the completion of all write transactions to the persistent memory, the program might reach an inconsistent state. For example, in a correct program execution, the allocation of a variable must be performed before the pointer to it is recorded. Assume that the write respective to the allocation happens at cache line L_A and the same respective to recording the pointer happens at cache line L_R. The processor ensures that the write to L_A precedes the write to L_R to maintain a consistent program state. However, when the respective cache lines are written back to the persistent memory, the order of writes might be reversed depending on the processor. If the system crashes after writing L_R to the persistent memory, but before writing L_A, then the program's data structure will remain at an inconsistent state, basically creating a *dangling pointer*. We note that volatile memories, such as DRAM, do not have such consistency issues. This is because writing L_R to volatile memory would have been discarded naturally when the system restarts.

In summary, regarding the crash consistency of data stored in persistent memory, we must ensure that memory writes happen in the order they have been *intended* by the programmer. To this end, in-NVM data structures need to leverage cache line flushes and memory fences. Cache line flush forcefully flushes a cache line to memory and memory fences guarantee that memory operations, i.e., load and/or store instructions, after a fence would not be performed until memory operations before the fence complete. Thus, a combination of cache line flushes and memory fences following each memory write can ensure a desired order of writing data to the persistent memory. Nevertheless, it is well known for x86 processors that the overhead of using `clflush` (cache line flush) and `mfence` (memory fence) is significant, leading to severe performance degradation [10, 25, 28].

2.2 B+-tree Variants for Persistent Memory

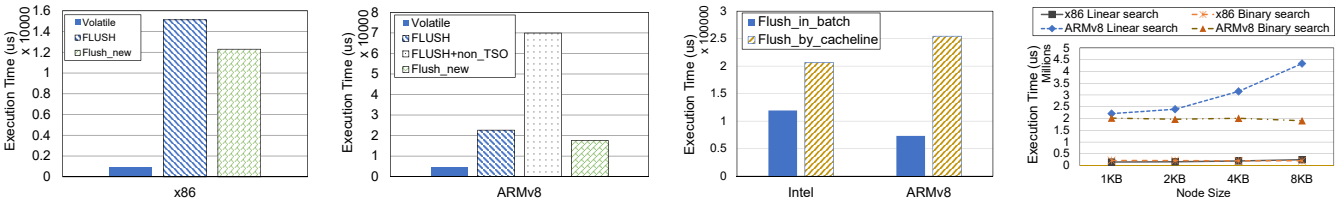
Several crash-consistent B+-tree variants have been proposed for persistent memory [24–28]. In a B+-tree, a key and a value (an 8B pointer to a record) form a KV pair. The KV pair is stored in a leaf node (LN) while upper-level index nodes (INs) assist in searching a specific key. LNs are critical to a B+-tree, as keys and pointers maintained in INs can be reconstructed by traversing LNs [25, 27].

Access performance and crash consistency are of paramount importance for an in-NVM B+-tree. Owing to the significant overhead of using cache line flushes and memory fences, existing B+-tree variants all try to reduce the consistency cost. It was discovered initially that maintaining *sorted* keys in a B+-tree node entails a large amount of cache line flushes and memory fences. As a result, NV-Tree [25], wB+-tree [26] and FPTree [27] append unsorted keys with values into tree nodes to reduce the use of cache line flushes and memory fences. Although this improves write performance, searching a key in an unsorted node is suboptimal, as a linear scan must be performed over unsorted keys.

Recently Hwang et al. [28] found that, on insertion or deletion with a sorted B+-tree node, shifting neighboring KV pairs in one cache line does not need cache line flush or memory fence because of the store dependencies between them ($key_i \rightarrow key_{i+1}$, $key_{i-1} \rightarrow key_i$, and $value_i \rightarrow value_{i+1}$, $value_{i-1} \rightarrow value_i$). But shifting KV pairs from one cache line to another does not guarantee these two cache lines will be written back to memory in their modification order. Also, store dependencies only exist between keys or values, and key and value are separately stored ($key_i \nleftrightarrow value_i$); for x86 with TSO, consecutive stores of key and value of one KV pair persist their writing order. These observations indicate that `clflush` and `mfence` only need to be used for orderly flushing dirty cache line, instead of each KV pair, in inserting and deleting data with B+-tree in an x86-based system. Unfortunately, for non-TSO architectures, such as ARM, an additional memory fence must be called to preserve an order in storing the key and value of a KV pair so as to avoid mismatch in each KV pair after a system crash.

3 Motivational Study

Designing a crash-consistent B+-tree in NVM demands cache line flushes and memory fences, the cost of which has not been quantitatively analyzed for ARMv8. Moreover, for ARMv8 with non-TSO, a memory fence is needed to orderly store the key and value of each KV pair to avoid any mismatch after a crash. Whereas, the cost of such extra memory fences is also unknown.



(a) Consistency cost with x86 (b) Consistency cost with ARMv8 (c) Batch flush vs. flush by cache lines (d) Linear search vs. binary search

Figure 1. A Study of Consistency Costs and Search Performance for B+-tree with x86 and ARMv8 Processors

Setup. We have performed an empirical study with a use case to address these concerns with ARMv8. We choose a 2KB B+-tree node in which every key/value takes 8B. We simulate a scenario where we need to shift 63 KV pairs to insert a new smallest key with a value. We have repeated the insertion for 10,000 times both with an x86 processor (Intel Core i7-7700) and with an ARMv8 processor (ARM Cortex-A53). We perform these operations for several different configurations of B+-trees. We first consider a volatile B+-tree without cache line flush or memory fence (`Volatile`). Another configuration is to orderly flush dirty cache lines during shifting KV pairs (`FLUSH`). For ARMv8, the third configuration is adding a memory fence to `FLUSH` to orderly shift the key and value for each KV pair (`FLUSH+Non_TSO`). This is to counteract non-TSO of ARMv8. `FLUSH` and `FLUSH+Non_TSO` follow [28] to guarantee crash consistency.

Key observations. We have measured the execution time for different configurations with two processors and the left two diagrams of Figure 1 capture all the results. Let us first analyze the performance overhead of using cache line flushes and memory fences through a comparison between `Volatile` and `FLUSH` for both processors. Compared to `Volatile`, `FLUSH` spent 17.1× and 5.1× time in inserting KV pairs for x86 and ARMv8, respectively. Therefore, *the consistency cost due to using cache line flushes and memory fences, despite being seemingly less than that of x86, is still significant for ARMv8.* Therefore, the design of an in-NVM B+-tree with ARMv8 should minimize the use of cache line flushes and memory fences.

`FLUSH` is sufficient in preserving crash consistency for x86 processors. But for ARMv8, `FLUSH+non_TSO` is the one that guarantees crash consistency with additional memory fences to counteract non-TSO. To the best of our knowledge, there does not exist quantitative analysis of such extra consistency cost incurred for non-TSO architectures. In Figure 1(b), the execution time of `FLUSH+non_TSO` is 3.1× that of `FLUSH`. In other words, *the time cost for counteracting non-TSO using additional memory fences is substantial and might badly degrade the performance of a consistent in-NVM B+-tree.* Added by this further considerable overhead, the overall consistency cost stretches the execution time by 15.8× from `Volatile` to `FLUSH+non_TSO` for ARMv8, which is close to the 17.1× performance gap between `Volatile` and `FLUSH` for x86.

Reducing consistency cost with ARMv8. As revealed with x86 processors [24, 25], a batched flush of a contiguous memory space, i.e., flushing cache lines of it with a series of cache line flushes but only one ending memory fence, consumes much less time than flushing the space cache line by cache line through many cache line flushes and memory fences. We first did a test to verify if ARMv8 complies with this observation. We allocated a cache line-aligned memory space of 512MB, and flushed it in the batched way and by each cache line, respectively, with both x86 and ARMv8 processors. Figure 1(c) shows the execution time. With x86 processor, the time cost of batched flush improves performance by a factor of 1.7. For ARMv8, such improvement is more significant, i.e., by a factor of 3.5. As a result, *with ARMv8 processors, flushing in an aggregated batch should save more time than individually flushing cache lines.*

Inspired by this observation, we designed a new configuration called `Flush_new` and applied it to the insertion use case. In `Flush_new`, we allocated a new tree node and copied all KV pairs, including the new KV pair, into the newly-allocated node. We then flushed all KV pairs in the new node to NVM using the batched flush. Finally, we replaced the original node in the B+-tree with the new node through an 8B atomic write supported by ARMv8. Prior to replacing the original node, there is no need to counteract non-TSO via extra memory fences, because the new node is still *outside* the tree and flushing all its KV pairs with one ending memory fence to NVM resolves all ordering issues. As shown in Figure 1(a), `Flush_new` marginally reduces the execution time with x86, but, with ARMv8, it significantly drops the execution time by 4.0× compared against `FLUSH+non_TSO`. This performance boost motivates the design of our in-NVM B+-tree on ARMv8 with minimized consistency cost.

Cost of search in ARMv8. Searching a key is required for both write and read operations. State-of-the-art B+-tree variants have different strategies to accelerate the process of search. NV-Tree performs binary search in INs and linear scan in LNs since its LNs are unsorted [25]. FAST-FAIR applies linear search in both INs and LNs although all of its nodes keep the keys sorted, because linear search yields better performance with node size no greater than 4KB with x86 processor [28]. Since embedded systems generally have

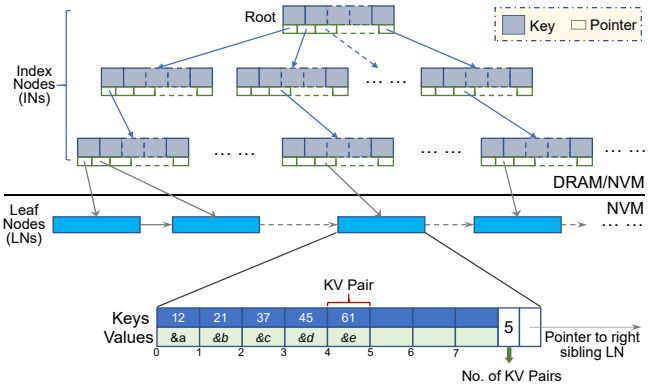


Figure 2. The Structures of Crab-tree and its Leaf Node

fewer computational resources and employ different micro-architectural features (e.g., non-inclusive cache and in-order pipeline), we have empirically compared binary search to linear search between our x86 and ARMv8 processors with varied node sizes. Figure 1(d) shows the execution time of searching 1 million keys in a volatile B+-tree with all sorted nodes. It is evident that binary search costs much less or leads to comparable performance to linear search with ARMv8. Therefore, we consider binary search in designing our B+-tree for ARMv8.

We note that we have utilized an economical ARMv8 Cortex-A53 for case study. Nevertheless, the overhead of using cache line flushes and memory fences to guarantee ordered writes and counteract non-TSO persists with high-end ARMv8 processors. In fact, such overhead shall cause more severe performance degradation for processors with higher frequencies and larger caches.

4 Design of Crab-tree

4.1 Overview of Crab-tree

Crab-tree is short for **c**rash **r**ecoverable **A**RMv8-oriented **B**+-tree, developed for persistent memory operated by an ARMv8 processor. Figure 2 shows the architecture of Crab-tree. As INs can be rebuilt by scanning LNs, Crab-tree only enforces crash consistency to LNs. Figure 2 also captures a logical view of an LN. An LN has KV pairs, a counter that records the number of valid KV pairs in the LN, and a pointer pointing to the right sibling LN. Crab-tree organizes LNs in an LN linked list using this pointer. Both counter and pointer can be modified with an 8B atomic write. INs can be maintained in DRAM with a hybrid DRAM/NVM system.

Insertion, update and deletion are write operations to a target LN for B+-tree. To update a KV pair, Crab-tree replaces the value (pointer to a new record) via an 8B atomic write. As to insertion and deletion, existing KV pairs in the target LN should be shifted unless the key to be inserted or deleted is the largest. Crab-tree employs two strategies for insertion and deletion: copy-on-write (COW) and shifting in place (SIP). COW copies all KV pairs to a new LN and replaces the original LN through a batched flush. SIP shifts KV pairs in the original LN by orderly flushing dirty cache lines. Crab-tree chooses one of them to handle an insertion or deletion depending on their consistency costs. Given a specific key, the target LN it belongs to is deterministic, and assuredly, the number of KV pairs that should be shifted is calculable, which entails the number of cache line flushes and memory fences required to orderly flush dirty cache lines and counteract non-TSO. The consistency cost of SIP is hence determined. Also, the cost of COW in copying and flushing all KV pairs to a new LN as well as replacing the original LN can be estimated. Crab-tree picks the one with lower cost.

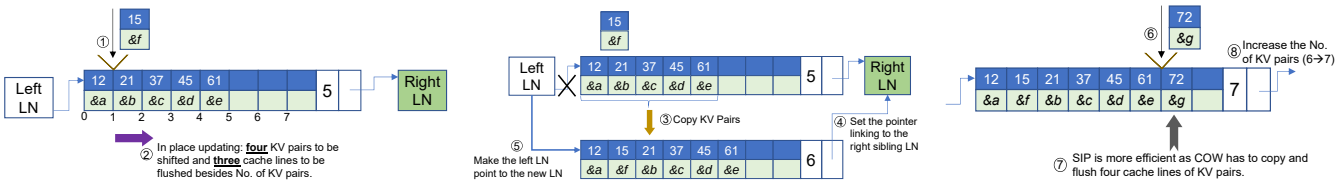
Both COW and SIP abide by strict orders of modification and store. To do COW, Crab-tree first creates a new LN and copies KV pairs. In case of insertion, Crab-tree copies and puts the newly-arrived KV pair in the appropriate position of the new LN, while for deletion it omits the KV pair to be deleted in copying. After copying, Crab-tree sets the new LN's pointer linking to the right sibling of the original LN. Crab-tree replaces the original LN by modifying its left sibling's pointer to point to the new LN. Modifying this pointer is to incorporate the new LN into the LN linked list. Doing so is done via an 8B atomic write, which does not compromise Crab-tree's crash consistency, as the atomic write either catches the original LN or the new LN in the LN linked list.

SIP leverages a property of B+-tree: B+-tree nodes do not allow duplicate values (pointers to lower-level nodes or records). On shifting KV pairs, SIP shifts a value to make duplicate values prior to shifting its corresponding key. In the final step of storing a new KV pair, SIP stores the key before storing the value so as to retain duplicate values for consistency check. However, using duplicate values alone cannot exactly recover an inconsistent LN, because both insertion and deletion shift KV pairs but in opposite directions (right for insertion and left for deletion). It is also difficult to tell which key factually matches the duplicate value. To resolve this issue, after shifting KV pairs, our SIP subtly enforces an atomic write to modify the number of KV pairs, by which Crab-tree can correctly recover from a crash.

In summary, Crab-tree tries to minimize consistency costs by online selecting COW or SIP. In either way it always ends a write transaction by an 8B atomic write, which secures the crash recoverability of Crab-tree. As a result, Crab-tree achieves both high performance and crash consistency.

4.2 Insertion

Algorithm 1 illustrates how Crab-tree inserts a KV pair. Inserting a new KV pair starts by traversing from the root node until a target LN (Line 1). Then Crab-tree checks if the current LN is full (Line 2, D means the maximum number of KV pairs allowed to be stored in one LN). If so, the LN will be split with the newly-arrived KV pair (Line 3). Otherwise, Crab-tree uses binary search to locate the position where the new KV pair will be put (Lines 5 to 6). Given a system with a



(a) COW (Part 1) (b) COW (Part 2) (c) SIP

Figure 3. An Illustration of Inserting KV Pairs by Crab-tree

Algorithm 1 Insertion of Crab-Tree (`Insert`($< k, v >$))

Input: A KV pair $< k, v >$ to be inserted

1: Search from the root until the target LN ln is obtained //ln has the number of KV pairs n
2: **if** ($ln.n \geq$ D) **then** //D is the degree (max No. of keys) of an LN
3: `split`(ln, $< k, v >$); //We must split ln with $< k, v >$
4: **else**
5: //Get the insertion location and decide to do COW or SIP
6: p := `get_insert_location`(ln, $ln.n$, k);
7: **if** (`COW_is_more_efficient`($ln.n$, p) == **True**) **then**
8: new_ln := `pmalloc`(LN_SIZE);
9: **for** ($i := 0$; $i < ln.n$; $i := i + 1$) **do** //Copy KV pairs
10: $new\_ln.KVs[i < p\ ?\ i : i + 1].k := ln.KVs[i].k$;
11: $new\_ln.KVs[i < p\ ?\ i : i + 1].v := ln.KVs[i].v$;
12: $new\_ln.KVs[p].k := k$, $new\_ln.KVs[p].v := v$;
13: $new\_ln.n := ln.n + 1$;
14: $new\_ln.rightSibling := ln.rightSibling$;
15: //Let us flush new_ln to NVM
16: `FLUSH_FENCE`(new_ln, (KV_SIZE $\times$ $new\_ln.n$) + `sizeof`($new\_ln.rightSibling$) + `sizeof`($new\_ln.n$));
17: $left\_ln$:= `get_left_sibling`(ln);
18: $left\_ln.rightSibling := new\_ln$;
19: `FLUSH_FENCE`(($left\_ln.rightSibling$), `sizeof`($left\_ln.rightSibling$));
20: Update parental INs of ln with new_ln if necessary;
21: `pfree`(ln);
22: **else** //We need to do SIP
23: **for** ($i := ln.n + 1$; $i > p$; $i := i - 1$) **do** //Shift KV pairs
24: $ln.KV[i].v := ln.KV[i-1].v$; FENCE();
25: $ln.KV[i].k := ln.KV[i-1].k$; FENCE();
26: **if** ($ln.KV[i]$ is at boundary of cache lines) **then**
27: //We need to flush a dirty cache line
28: `FLUSH_FENCE`(($ln.KV[i]$), CL_SIZE);
29: $ln.KV[p].k := k$; FENCE();
30: $ln.KV[p].v := v$;
31: `FLUSH_FENCE`(($ln.KV[p]$), KV_SIZE);
32: $ln.n := ln.n + 1$;
33: `FLUSH_FENCE`(($ln.n$, `sizeof`($ln.n$));
34: **Return** the completion of insertion;

specific ARMv8 processor, the time cost of cache line flushes and memory fences as well as copying data can be estimated through a quick test. With the insertion position and the respective numbers of KV pairs to be copied or shifted, Crab-tree estimates both costs of COW and SIP (Line 7). If using COW is likely to spend less time, Crab-tree will follow COW to insert the newly-arrive KV pair (Lines 8 to 21). Otherwise, Crab-tree will perform SIP (Lines 23 to 33).

Crab-tree first allocates a new LN for COW (Line 8). There are libraries that manage NVM space [15, 17]. `pmalloc` at Line 8 particularly fills the allocated space with nulls (zeros). Crab-tree then copies existing KV pairs from the original LN to the newly-allocated LN except leaving an appropriate position (i.e., p at Lines 6) vacant for the new KV pair (Lines 9 to 11). After the new KV pair is stored into the newly-allocated LN (Lines 12), Crab-tree sets the number of KV pairs (Line 13) and makes the newly-allocated LN's pointer point to the right sibling of original LN. By doing so, Crab-tree 'half' links the new LN into the LN linked list. Crab-tree flushes the new LN into NVM to persist all copied KV pairs, the right sibling pointer and the number of KV pairs. Crab-tree then finds the left sibling of original LN (Line 17), and alters the left sibling's pointer to point to the new LN (Line 18). After flushing the left sibling's pointer (Line 19), the new LN successfully replaces the original LN in the LN linked list. Next Crab-tree updates parental INs with the pointer of the new LN (Line 20) and frees the space occupied by the original LN (Line 21).

A pointer in NVM has a size of 8B with ARMv8 64-bit processors, so storing an in-NVM pointer can be atomically done. Modifying the pointer of left sibling LN either completes or fails. If a crash occurs after it completes, Crab-tree observes the new LN. Otherwise it finds the original LN. In either case, the crash consistency of Crab-tree is not impaired. Moreover, before the batched flush at Line 16, Crab-tree does not execute any cache line flush or memory fence in copying KV pairs and linking the right sibling LN. The reason is that the new LN is still outside the tree before modifying the pointer of left sibling, so any changes and any store order in the new LN have no impact to Crab-tree's crash consistency. In addition, on locating the left sibling of the original LN (the function `get_left_sibling` at Line 17), Crab-tree does not need to traverse again from the root until the parental IN. At the beginning of each insertion, Crab-tree has traversed INs to find the target LN (Line 1). At that time Crab-tree can maintain an array, say `traversed_INs`[$h - 1$] (h is the height of tree), in which an element records a traversed IN and the position of the node traversed at the next level. To get an LN's left sibling, Crab-tree just checks the last element of `traversed_INs`. Unless the recorded position is

zero, which means that upper-level traversed IN(s) should be considered, Crab-tree promptly locates the LN's left sibling. `traversed_INs` is an auxiliary array to temporarily record traversed INs, and does not need consistency guarantee.

If the cost of SIP is less, Crab-tree first shifts KV pairs to the right (Lines 23 to 28) to make space. It shifts values before shifting keys to maintain duplicate values with extra memory fences (Lines 24 to 25). During shifting KV pairs, when Crab-tree moves from one cache line to the left one, it needs to flush that dirty cache line (Lines 26 to 28). After shifting, Crab-tree stores the new key with a memory fence and then the new value (Lines 29 to 30), followed by a cache line flush and memory fence to persist the new KV pair (Line 31). Crab-tree increases and persists the number of KV pairs to end the insertion (Lines 32 to 33). This is also done via an 8B atomic write for crash recovery (cf. Section 4.5).

Figure 3 shows two insertions with Crab-tree. We assume that a cache line holds two KV pairs. In Figure 3(a), Crab-tree first decides where the new KV pair should be inserted (①). If Crab-tree chooses SIP, four KV pairs have to be shifted with three cache line flushes and memory fences (②) as well as memory fences to counteract non-TSO. So Crab-tree chooses COW. In Figure 3(b), it allocates a new LN and copies all six KV pairs to the new LN (③). Then it replaces the original LN with the new LN by setting two pointers (④ and ⑤). Figure 3(c) illustrates another scenario in which the key to be inserted is the new largest key. COW copying seven KV pairs takes more time than SIP. So Crab-tree performs SIP (⑥ and ⑦) and increases the number of KV pairs (⑧).

Split. Crab-tree splits an LN when the LN is full on inserting a new KV pair. Crab-tree still aims to minimize the use of cache line flushes and/or memory fences in splitting. Previous B+-tree variants for NVM split an LN into two LNs and insert the new KV pair into one of them through a normal insertion [25, 28]. Crab-tree works differently. It first determines the split point for the LN, from which half KV pairs with larger keys will be copied to a new LN. If the new KV pair falls into the larger half, Crab-tree just copies it with other KV pairs and do a batched flush. It then modifies the original LN's number of KV pairs. Next Crab-tree adds the new LN into the LN linked list and adjusts parental IN(s) if necessary. If the new KV pair should join the smaller half, after copying the larger half, Crab-tree will assess the costs of COW and SIP as in a normal insertion. If it chooses COW, two new LNs will be linked into the LN linked list.

Splitting also demands strict execution orders. If SIP adds one new LN, Crab-tree first lets this LN point to the right sibling of the original LN. Then Crab-tree sets the original LN's pointer with the new LN's address as the original LN's new right sibling. Both pointers are modified by 8B atomic writes. COW adding two new LNs, Crab-tree first links them according to the ascending order of keys. It then makes the right new LN point to the original LN's right sibling. Next, Crab-tree finds out the original LN's left sibling and changes its pointer to link the left new LN. With this order, two LNs are consistently incorporated into the LN linked list.

4.3 Search

Looking up a value given a key for Crab-tree follows a binary search as justified in Section 3. In the beginning, Crab-tree gets the tree root. Then it traverses from the root until the target LN by comparing keys in binary search. Next Crab-tree does binary search in the LN. It will return the value if the key is found or a code for miss (−1). The same procedure is also used in inserting, updating and deleting a KV pair.

4.4 Deletion

Deleting a KV pair is a reverse operation of insertion and requires shifting KV pairs in a tree node. Crab-tree considers two cases when removing a KV pair from an LN.

- If shifting KV pairs in the LN via SIP incurs less consistency cost than COW, Crab-tree will call SIP to shift KV pairs in-situ.
- Otherwise, Crab-tree allocates a new LN and copies to it KV pairs excluding the one to be deleted. Crab-tree then replaces the original LN using the new LN.

We note that there is a particular act for Crab-tree using SIP to delete a KV pair. With SIP for deletion, KV pairs are shifted to the left and the largest KV pair would hence be duplicate in the end of valid KV pairs. Prior to decreasing the number of KV pairs of the LN, Crab-tree clears the value (pointer) of the largest key in its original position to be null (zero). As mentioned in Section 4.2, when allocating a new LN, Crab-tree calls `pmalloc` to fill the LN with nulls. On deletion Crab-tree again sets a vacated value to be null. Null pointers exist as a landmark to identify valid KV pairs and are especially useful in recovery (cf. Section 4.5).

Merge & Redistribution. Continuous deletions decrease the space of LNs. When an LN becomes underutilized (the number of valid KV pairs less than $\frac{D}{2}$), Crab-tree considers merging or redistributing its KV pairs with its siblings. Merging is called if the LN's either sibling has sufficient vacant space to absorb all the underutilized LN's KV pairs. After merging, the LN linked list is adjusted by detaching the underutilized LN, and parental INs will be updated where necessary. In addition, Crab-tree prefers merging KV pairs from an LN to its left sibling if possible so that it can perform a batched flush with less time cost.

If neither sibling of the underutilized LN has enough space, Crab-tree will redistribute KV pairs from two siblings to the underutilized LN until it has more than $\frac{D}{2}$ KV pairs. Crab-tree applies SIP in shifts KV pairs for redistribution. It also needs to update parental IN(s) after the redistribution.

4.5 Rebuilding and Recovery at Startup

Crab-tree keeps INs volatile like a standard B+-tree. After a normal exit or crash, Crab-tree rebuilds INs. Note that

Crab-tree rebuilds only once at startup, unlike NV-tree that has to frequently rebuild INs by demanding ever-increasing memory space and stall write transactions at runtime [25].

The rebuilding of Crab-tree is also its recovery procedure. On rebuilding, Crab-tree scans every LN in the LN linked list with two tasks. One task is to identify the smallest and largest keys of the LN for filling upper-level INs. The other task is to check crash consistency. COW does not cause inconsistency because the ending atomic write of COW leaves either the original LN or the new LN. However, to avoid memory leakage caused by a crash, the addresses of newly-allocated and original LNs involved in COW can be recorded. In recovery, if either address is not found in the LN linked list, Crab-tree will ask memory library to reclaim that space.

SIP might cause inconsistency. When scanning an LN, Crab-tree tries to find duplicate values. It also obtains the number of KV pairs stored in the LN and counts the number of non-null values. There are three inconsistent scenarios.

- No duplicate value is found, but the recorded number of KV pairs differs from the number of non-null values.
- Crab-tree finds duplicate value and the recorded number of KV pairs differs from the number of non-null values.
- Crab-tree finds duplicate value but the recorded number of KV pairs is the same as the number of non-null values.

For the first scenario, if the recorded number is one smaller than the number of non-null vaules, the crash must have occurred in insertion before increasing the number of KV pairs. If it is one larger than the number of non-null values, the crash must have happened in deletion before decreasing the number of KV pairs. In either case, Crab-tree just needs to modify the number of KV pairs via an 8B atomic write.

The second and third scenarios are caused when Crab-tree is shifting KV pairs. Insertion and deletion shift KV pairs in opposite directions. As shifting in insertion always moves KV pairs to the right to make space, a crash that has happened at that moment leaves one more non-null and duplicate value, which corresponds to the second scenario. Crab-tree shifts KV pairs to the left for *roll-back* recovery. On the other hand, a crash that has happened during shifting KV pairs in deletion would have one duplicate value but the number of KV pairs has not been changed yet, i.e., being equal to the number of non-null values, which matches the third scenario. Crab-tree shifts KV pairs to the left for *roll-forward* recovery.

We note that Crab-tree shares similarities with FAST-FAIR in using duplicate values to identify inconsistency. However, they differ a lot in checking consistency and recovering from crashes. First, FAST-FAIR does lazy recovery and fixes inconsistencies in future write transactions. Crab-tree actively recovers inconsistent LNs at startup. More important, Crab-tree leverages the number of valid KV pairs with duplicate values to swiftly and precisely recover an inconsistent LN, which is not covered by FAST-FAIR.

5 Evaluation

5.1 Experimental Setup

We have chosen the latest Raspberry Pi 3 Model B+ [56] as the evaluation platform. It has a 64-bit quad-core ARM Cortex-A53 processor. We installed Opensuse Leap 15.0 ARMv8 version [57]. The compiler is GCC/G++ 7.3.1. Because there is no NVM shipped for Raspberry Pi as of today, we assume that all the 1GB memory of Raspberry Pi is made of NVM with the same access latency as DRAM. This is in line with the MRAM products produced by Everspin which have good write endurance as well as symmetrical write and read latencies that are comparable to those of DRAM [1].

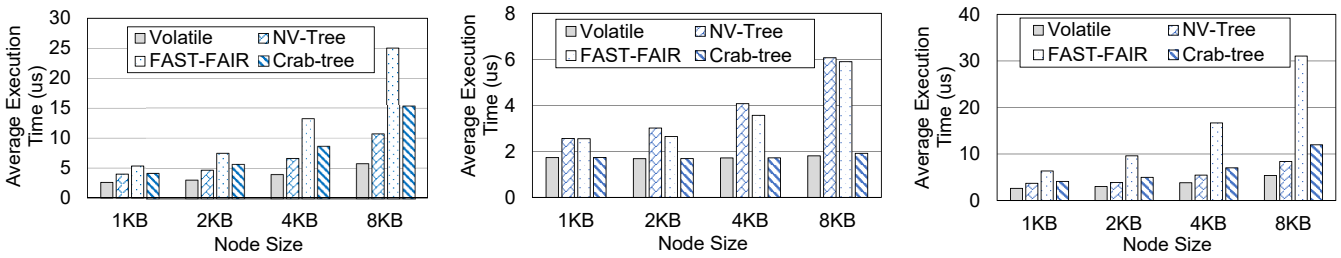
We have implemented our Crab-tree with COW and SIP in C (`Crab-tree`). The instructions of cache line flush and memory fence are `dc cvac` and `dmb`, respectively, because `dc cvac` corresponds to the newest cache line flush instruction of x86 (`clwb`) as indicated by Intel PMDK [17]. As to competitors in evaluation, besides a volatile B+-tree without any consistency guarantee (`Volatile`), we have considered NV-Tree and FAST-FAIR. We implemented an NV-Tree (`NV-Tree`) regarding its original design [25]. We downloaded the source codes of FAST-FAIR for x86. We replaced x86's cache line flush and memory fence instructions using ARMv8's and added memory fences to counteract non-TSO in line with the algorithms of FAST-FAIR [28] (`FAST-FAIR`). `Volatile` and `Crab-tree` use binary search while `FAST-FAIR` uses linear search. `NV-Tree` performs binary search in INs and linear search in LNs as its LNs are unsorted.

For each tree, IN and LN have the same size, i.e., 1KB, 2KB, 4KB and 8KB in our experiments. We do not choose smaller node sizes (<1KB) concerning both time- and space-efficiencies. First, a smaller node is likely to trigger more splits that incur more performance overheads. Second, a smaller node means smaller capacity; given the same quantity of stored data, the tree will have a larger height, which impairs searching from the root to a target LN for every operation. Third, still with the same quantity of stored data, a smaller LN requires many more INs, which underutilizes the system's memory space.

As to the workloads, we have considered keys following a uniform distribution without write skewness. It was used by FAST-FAIR and also favored by NV-Tree [25, 28]. We have inserted, searched and deleted 1 million and 10 million keys, respectively, with aforementioned four trees. The performance of each tree is measured using the average execution time per operation in micro-seconds (μs).

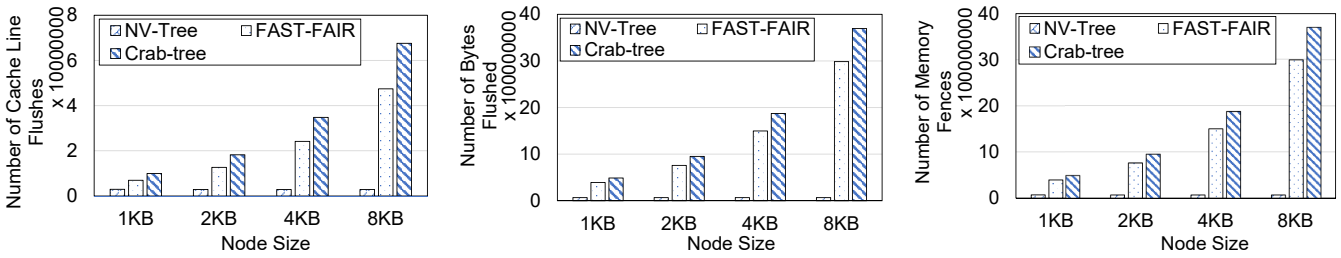
5.2 Write and Read Performance

The three diagrams in Figure 4 capture the average execution time of inserting, searching and deleting one million keys with four trees, respectively. Since `Volatile` has no consistency cost and employs binary search, its write and



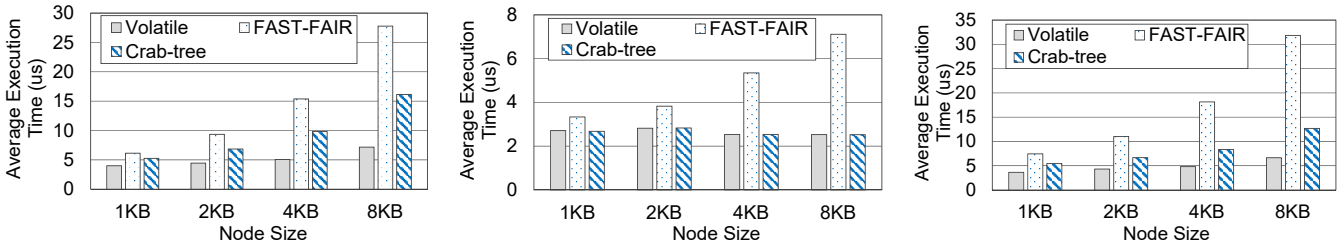
(a) Insertion Performance (b) Search Performance (c) Deletion Performance

Figure 4. A Comparison of Insertion, Search and Deletion Performance with 1 Million Keys



(a) The Number of Cache Line Flushes (b) The Amount of Data Flushed in Bytes (c) The Number of Memory Fences

Figure 5. The Numbers of Cache Line Flushes, Bytes Flushed and Memory Fences for Inserting 1 Million Keys



(a) Insertion Performance (b) Search Performance (c) Deletion Performance

Figure 6. A Comparison of Insertion, Search and Deletion Performances with 10 Million Keys

read performances can be considered as the optimal performances. `NV-Tree` maintains unsorted keys in each LN, and it does not shift KV pairs but just append to the end of an LN a KV pair with positive flag for insertion or negative flag for deletion. As a result, it shows higher write performance by 5.5%–30.0% than `Crab-tree` with four node sizes. However, owing to the unsorted keys, `NV-Tree` has to linearly scan all KV pairs in an LN for searching. That explains why its search performance is much worse than `Crab-tree`'s. In particular, the search time of `NV-Tree` is 1.6×, 1.8×, 2.5× and 3.2× that of `Crab-tree`, respectively, with four node sizes. The gap of search performance between them increases because a larger node requires more time for linear scan.

`Crab-tree` also outperforms `FAST-FAIR` in both write and read performances. With the node size of 8KB, the execution time of `FAST-FAIR` for insertion and deletion are 1.6× and 2.6× that of `Crab-tree` while its search time is 3.1× that of `Crab-tree`. The reason why `Crab-tree` yields higher performances than `FAST-FAIR` is threefold. First, on insertions and deletions, `Crab-tree` selectively uses COW and SIP to minimize consistency costs but `FAST-FAIR` always does shifting in situ. Second, `Crab-tree` enforces consistency to LNs while `FAST-FAIR` covers all nodes. Third, the linear search used by `FAST-FAIR` is slower than binary search of `Crab-tree`, which affects the efficiency for all write and read operations of B+-tree. The design of FAST-FAIR, especially its crash recovery, is yet closely bound to linear search [28].

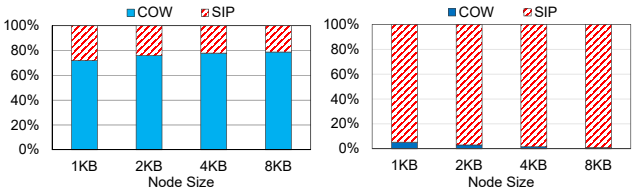
For a quantitative analysis, we have recorded the number of cache line flushes, the amount of bytes flushed to NVM and the number of memory fences during inserting 1 million keys. Figure 5 shows the results. As `NV-Tree` always appends a new KV pair for insertion, it flushes the fewest data with the fewest memory fences. A comparison between `FAST-FAIR` and `Crab-tree` in Figure 5(a) and 5(b), however, may seem contradictory to the results shown in Figure 4, because `Crab-tree` executed more cache line flushes and flushed more data than `FAST-FAIR`, i.e., at most 23.7% with 8KB node size. We note that one idea of Crab-tree is to leverage COW with batched flushes to flush KV pair in

an aggregated manner instead of flushing by cache line. The large amount of data flushed by `Crab-tree` is just entailed by COW. We have made a breakdown of data flushed by COW and SIP when `Crab-tree` was inserting 1 million keys. Figure 7(a) captures the percentages contributed by COW and SIP. It is evident that at least 71.9% flushed data was done by COW through batched flushes, which enables `Crab-tree` to achieve higher write performance than `FAST-FAIR`.

We further did a pressure test to evaluate Crab-tree. We increased the number of keys to be operated from 1 million to 10 million. Whereas, the design of `NV-Tree` has to allocate a huge contiguous memory space when rebuilding INs at runtime [25]. Given a large volume of keys to be processed, the ever-increasing memory space required by NV-Tree will be eventually beyond the capability of an embedded system. Thus, `NV-Tree`, being with good write performance but suboptimal read performance under small workloads, is not scalable or robust for large workloads. We thus compare among `Volatile`, `FAST-FAIR` and `Crab-tree`. Figure 6 presents their execution time. The write and read performances of three trees degrade with increased keys. However, `Crab-tree` still significantly outperforms `FAST-FAIR`. For example, with 8KB node, `FAST-FAIR` had to spend 1.7×, 2.8× and 2.5× time than `Crab-tree` to completes an insertion, search and deletion request, respectively. More important, when the workload scales up, the performance of `Crab-tree` does not badly fluctuate. Take 8KB node size for example again. The average time of deleting a KV pair is 12.0μs and 12.7μs with 1 millon and 10 million keys, respectively. As a result, Crab-tree has good robustness and scalability.

5.3 The Impact of COW and SIP

The high performance of `Crab-tree`, especially the write performance, is accredited to the selection between COW and SIP. `Crab-tree` makes a decision when a key is to be inserted to an LN. The breakdowns of flushed data and executed memory fences in Figure 7(a) and 7(b) show that COW and SIP act in their respective ways for crash consistency. More, we solely used either strategy to insert 1 million and 10 million keys. In Figure 8, `COW-only` and `SIP-only` mean using COW and SIP alone for inserting keys, respectively.



(a) The Breakdown of Data Flushed by COW and SIP (b) The Breakdown of Memory Fences Executed by COW and SIP

Figure 7. The Breakdowns of COW and SIP in Using Cache Line Flushes and Memory Fences

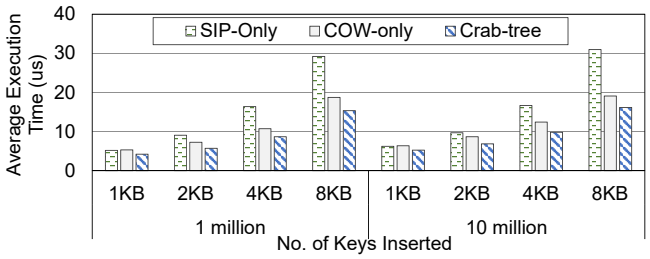


Figure 8. A Comparison among `COW-only`, `SIP-only` and `Crab-tree`

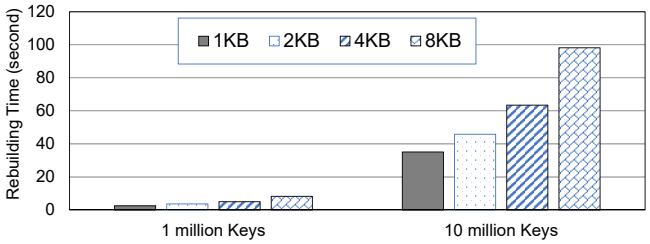


Figure 9. The Rebuilding Time of `Crab-tree`

The execution time of `Crab-tree` with a runtime combination of COW and SIP is much less than that of `COW-only` and `SIP-only`. For example, on inserting 1 million keys with 8KB node size, the time costs of `COW-only` and `SIP-only` are 90.5% and 22.2% more than that of `Crab-tree`. These results clearly demonstrate that employing one strategy would surely yield suboptimal performance compared to an integration and runtime selection between them.

5.4 Rebuilding Time

Crab-tree rebuilds INs at startup. The rebuilding of Crab-tree is a one-time operation, unlike NV-Tree that uses special IN format and must frequently rebuild INs especially under a workload with write skewness [25]. We have measured the time of rebuilding after a normal shutdown with 1 million and 10 million keys stored, respectively. Because the NVM we have used is emulated using DRAM, we saved LNs of Crab-tree to a file stored in the Micro-SD card of Raspberry Pi. `Crab-tree` loaded LNs from the file and initiated rebuilding. It scanned each LN to check whether any inconsistency existed or not. Figure 9 illustrates the rebuilding time with four node sizes. The average time over all keys is about 8.3μs and 9.7μs for 1 million and 10 million stored keys, respectively. Crab-tree is hence robust and scalable in rebuilding by a marginal increase of time cost (16.9%) with ten times more stored data.

6 Conclusion

In this paper, we consider tailoring an in-NVM B+-tree with ARMv8. We first conduct an empirical study to figure out the weaknesses of state-of-the-art in-NVM B+-tree by porting it to ARMv8. We accordingly develop Crab-tree that

guarantees crash consistency with minimized performance overhead. Crab-tree selects one of two strategies, i.e., copy-on-write and shifting in place, for inserting and deleting data, depending on their respective consistency costs. Crab-tree also defines strict execution orders to rule out inconsistency after a system crash. Extensive experiments confirm that Crab-tree has both high performance and good scalability.

Acknowledgments

This work is partially supported by the Ministry of Education of Singapore under the grant MOE2018-T2-1-098.

References

[1] Everspin. MRAM technology attributes. https://www.everspin.com/mram-technology-attributes, December 2018.

[2] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *5th Biennial Conference on Innovative Data Systems Research (CIDR '11)*, pages 1–11, January 2011.

[3] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. GraphR: Accelerating graph processing using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 531–543, Feb 2018.

[4] Micron and Intel. 3D XPoint technology. http://www.micron.com/about/innovations/3d-xpoint-technology.

[5] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 57:1–57:12, New York, NY, USA, 2009. ACM.

[6] Mingzhe Zhang, Lunkai Zhang, Lei Jiang, Zhiyong Liu, and Frederic T. Chong. Balancing performance and lifetime of MLC PCM by using a region retention monitor. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 385–396, Feb 2017.

[7] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[8] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 73–80, San Jose, CA, 2013. USENIX.

[9] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[10] Youyou Lu, Jiwu Shu, and Long Sun. Blurred persistence in transactional persistent memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13, May 2015.

[11] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[12] Qingsong Wei, Chundong Wang, Cheng Chen, Yechao Yang, Jun Yang, and Mingdi Xue. Transactional NVM cache with high performance and crash consistency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 56:1–56:12, New York, NY, USA, 2017. ACM.

[13] Tseng-Yi Chen, Yuan-Hao Chang, Shuo-Han Chen, Chih-Ching Kuo, Ming-Chang Yang, Hsin-Wen Wei, and Wei-Kuan Shih. Enabling write-reduction strategy for journaling file systems over byte-addressable NVRAM. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 44:1–44:6, New York, NY, USA, 2017. ACM.

[14] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 12:1–12:16, New York, NY, USA, 2016. ACM.

[15] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[16] Deshan Zhang, Lei Ju, Mengying Zhao, Xiang Gao, and Zhiping Jia. Write-back aware shared last-level cache management for hybrid main memory. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 172:1–172:6, New York, NY, USA, 2016. ACM.

[17] Intel. Persistent memory development kit. http://pmem.io/pmdk/.

[18] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[19] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[20] Chundong Wang and Sudipta Chattopadhyay. LAWN: Boosting the performance of NVMM file system through reducing write amplification. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 6:1–6:6, New York, NY, USA, 2018. ACM.

[21] Fang Wang, Zhaoyan Shen, Lei Han, and Zili Shao. ReRAM-based processing-in-memory architecture for blockchain platforms. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, ASPDAC '19, pages 615–620, New York, NY, USA, 2019. ACM.

[22] ARM. ARM architecture reference manual ARMv8, for ARMv8-A architecture profile. https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf, December 2017.

[23] ARM. ARMv8-A architecture evolution. https://community.arm.com/processors/b/blog/posts/armv8-a-architecture-evolution, January 2016.

[24] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 1–15, Berkeley, CA, USA, 2011. USENIX Association.

[25] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, Santa Clara, CA, 2015. USENIX Association.

[26] Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[27] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-Tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. ACM.

[28] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, 2018. USENIX Association.

[29] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. Harmonia: A high throughput B+tree for GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 133–144, New York, NY, USA, 2019. ACM.

[30] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. Engineering a high-performance GPU B-Tree. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 145–157, New York, NY, USA, 2019. ACM.

[31] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. A specialized B-tree for concurrent datalog evaluation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 327–339, New York, NY, USA, 2019. ACM.

[32] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. NVM Duet: Unified working memory and persistent store architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 455–470, New York, NY, USA, 2014. ACM.

[33] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. Loose-ordering consistency for persistent memory. In *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pages 216–223, Oct 2014.

[34] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. ATOM: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, Feb 2017.

[35] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: A flexible and fast software supported hardware logging approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 178–190, New York, NY, USA, 2017. ACM.

[36] M. A. Ogleari, E. L. Miller, and J. Zhao. Steal but no force: Efficient hardware undo+redo logging for persistent memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 336–349, Feb 2018.

[37] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[38] Chundong Wang, Qingsong Wei, Jun Yang, Cheng Chen, and Mingdi Xue. How to be consistent with persistent memory? an evaluation approach. In *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, pages 186–194, Aug 2015.

[39] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

[40] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[41] Zhenyu Sun, Xiuyuan Bi, Hai (Helen) Li, Weng-Fai Wong, Zhong-Liang Ong, Xiaochun Zhu, and Wenqing Wu. Multi retention level STT-RAM cache designs with a dynamic refresh scheme. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 329–338, New York, NY, USA, 2011. ACM.

[42] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. *SIGPLAN Not.*, 47(4):401–410, March 2012.

[43] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.

[44] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 33–45, Santa Clara, CA, 2014. USENIX.

[45] Rujia Wang, Lei Jiang, Youtao Zhang, Linzhang Wang, and Jun Yang. Selective restore: An energy efficient read disturbance mitigation scheme for future STT-MRAM. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 21:1–21:6, New York, NY, USA, 2015. ACM.

[46] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 719–731, Santa Clara, CA, 2017. USENIX Association.

[47] Clinton W. Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R. Stan. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 50–61, Washington, DC, USA, 2011. IEEE Computer Society.

[48] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 672–685, Dec 2015.

[49] Chen Liu and Chengmo Yang. Secure and durable (SEDURA): An integrated encryption and wear-leveling framework for pcm-based main memory. In *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015*, LCTES'15, pages 12:1–12:10, New York, NY, USA, 2015. ACM.

[50] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 3–18, New York, NY, USA, 2015. ACM.

[51] Duo Liu, Kan Zhong, Tianzheng Wang, Yi Wang, Zili Shao, Edwin Hsing-Mean Sha, and Jingling Xue. Durable address translation in PCM-Based flash storage systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(2):475–490, Feb 2017.

[52] Daeyoung Lee and Hyunok Oh. A lifetime aware buffer assignment method for streaming applications on DRAM/PRAM hybrid memory. *ACM Trans. Embed. Comput. Syst.*, 12(1s):36:1–36:17, March 2013.

[53] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.

[54] Lei Han, Zhaoyan Shen, Zili Shao, and Tao Li. Optimizing RAID/SSD controllers with lifetime extension for flash-based SSD array. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2018, pages 44–54, New York, NY, USA, 2018. ACM.

[55] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Mannarswamy, Terence Kelly, and Charles B. Morrey. Failure-atomic updates of application data in a Linux file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 203–211, Berkeley, CA, USA, 2015. USENIX Association.

[56] The Raspberry Pi Foundation. Raspberry Pi 3 Model B+, 2018. https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/.

[57] openSUSE. openSUSE:AArch64, 2018. https://en.opensuse.org/openSUSE:AArch64.