FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory

Ismail Oukid[†‡] Johan Lasperas[‡] Anisoara Nica[‡]
Thomas Willhalm[*] Wolfgang Lehner[†]

[†]TU Dresden first.last@tu-dresden.de
[‡]SAP SE first.last@sap.com
[*]Intel Deutschland GmbH first.last@intel.com

ABSTRACT

The advent of Storage Class Memory (SCM) is driving a rethink of storage systems towards a single-level architecture where memory and storage are merged. In this context, several works have investigated how to design persistent trees in SCM as a fundamental building block for these novel systems. However, these trees are significantly slower than DRAM-based counterparts since trees are latency-sensitive and SCM exhibits higher latencies than DRAM. In this paper we propose a novel hybrid SCM-DRAM persistent and concurrent B$^+$-Tree, named *Fingerprinting Persistent Tree* (FPTree) that achieves similar performance to DRAM-based counterparts. In this novel design, leaf nodes are persisted in SCM while inner nodes are placed in DRAM and rebuilt upon recovery. The FPTree uses *Fingerprinting*, a technique that limits the expected number of in-leaf probed keys to one. In addition, we propose a hybrid concurrency scheme for the FPTree that is partially based on Hardware Transactional Memory. We conduct a thorough performance evaluation and show that the FPTree outperforms state-of-the-art persistent trees with different SCM latencies by up to a factor of 8.2. Moreover, we show that the FPTree scales very well on a machine with 88 logical cores. Finally, we integrate the evaluated trees in *memcached* and a prototype database. We show that the FPTree incurs an almost negligible performance overhead over using fully transient data structures, while significantly outperforming other persistent trees.

1. INTRODUCTION

The last few years have seen the rapid emergence of many novel types of memory technologies that have the additional characteristic of being non-volatile. These technologies are grouped under the umbrella term of *Storage Class Memory* (SCM) and include Phase Change Memory [6], Memristors [29], and Spin-Transfer Torque Magnetic Random Access Memory [4]. SCM combines the economic characteristics, capacity, and non-volatility property of traditional storage media with the low latency and byte-addressability of DRAM. Hence, SCM has a potential that goes beyond replacing DRAM: it can be used as universal memory, that is, as main memory and storage at the same time.

Given the characteristics of SCM, it is theoretically possible to persist data structures in SCM and at the same time obtain near-DRAM performance. This potential has been acknowledged by recent works and new systems that adopt this novel paradigm are emerging [5, 21, 18]. In this paper we investigate the design of index structures as one of the core database structures, motivated by the observation that traditional main memory B-Tree implementations do not fulfill the consistency requirements needed for such a use case. Furthermore, while expected in the range of DRAM, SCM latencies are slower and asymmetric with writes noticeably slower than reads. We argue that these performance differences between SCM and DRAM imply that the assumptions made for previous well-established main memory B-Tree implementations might not hold anymore. We therefore see the need to design a novel, persistent B-Tree that leverages the capabilities of SCM while exhibiting performance similar to that of traditional transient B-Tree.

Designing persistent data structures presents unprecedented challenges for data consistency since SCM is accessed via the volatile CPU cache over which software has only little control. Several works proposed SCM-optimized B-Trees such as the CDDS B-Tree [24], the wBTree [8], and the NV-Tree [28], but they fail to match the speed of an optimized DRAM-based B-Tree. Additionally, they do not address all SCM programming challenges we identify in Section 2, especially those of persistent memory leaks and data recovery.
To lift this shortcoming, we propose in this paper the *Fingerprinting Persistent Tree* (**FPTree**) that is based on four design principles to achieve near-DRAM-based data structure performance:

1. ***Fingerprinting***. Fingerprints are one-byte hashes of in-leaf keys, contiguously placed in the first cache-line-sized piece of the leaf. The FPTree uses unsorted leaves with in-leaf bitmaps –originally proposed in [8]–, such that a search would iterate linearly over all valid keys in a leaf. By scanning the fingerprints first, we are able to limit the number of in-leaf probed keys to **one** in the average case, which leads to a significant performance improvement.
2. ***Selective Persistence***. The idea is based on the well-known distinction between primary data, whose loss would infer an irreversible loss of information, and non-primary data that can be rebuilt from the former. Selective Persistence consists in storing primary data in SCM and non-primary data in DRAM. Applied to the FPTree, this corresponds to storing the leaf nodes in SCM and the inner nodes in DRAM. Hence, only leaf accesses are more expensive during a tree traversal compared to a fully transient counterpart.
3. ***Selective Concurrency***. This concept consists in using different concurrency schemes for the transient and persistent parts. Basically, the FPTree uses Hardware Transactional Memory (HTM) to handle the concurrency of inner nodes, and fine-grained locks to handle that of leaf nodes. Selective Concurrency elegantly solves the apparent incompatibility of HTM and persistence primitives required by SCM such as cache line flushing instructions. Con-

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-3531-7/16/06...$15.00

DOI: http://dx.doi.org/10.1145/2882903.2915251

currency control techniques such as key-range locking [14] and orthogonal key-value locking [16] are orthogonal to our work as we focus on concurrency of physical operations.

4. ***Sound programming model***. We identify four SCM programming challenges: *data consistency*, *partial writes*, *data recovery*, and *memory leaks*, the last two of which are not fully addressed in related work. We solve all these challenges by relying on a sound programming model that encompasses the use of *Persistent Pointers*, an optimized, crash-safe, persistent allocator, and a memory leak prevention scheme. Contrary to existing work, this allows us to take into account the higher cost of persistent memory allocations in the performance evaluation.

We implemented the FPTree and two state-of-the art persistent trees, namely the NV-Tree and the wBTree. Using micro-benchmarks, we show that the FPTree outperforms the implemented competitors, with respectively fixed-size and variable-size keys, by up to 2.6x and 4.8x for an SCM latency of 90 ns, and by up to 5.5x and 8.2x for an SCM latency of 650 ns. The FPTree achieves these results while keeping less than 3% of its data in DRAM. In addition, we demonstrate how the FPTree scales on a machine with 88 logical cores, both with fixed-size and variable-size keys. Moreover, we show that the FPTree recovery time is 76.96x and 29.62x faster than a full rebuild for SCM latencies of 90 ns and 650 ns, respectively. Finally, we integrate the FPTree and the implemented competitors in *memcached* and a prototype database system. Compared with fully transient trees, results show that the FPTree incurs only 2% overhead in *memcached* using the *mc-benchmark* and that performance is network-bound. When using the TATP benchmark on the prototype database, the FPTree incurs a performance overhead of only 8.7% and 12.8% with an SCM latency of respectively 160 ns and 650 ns, while the overheads incurred by state-of-the-art trees are up to 39.6% and 51.7%, respectively.

The rest of the paper is organized as follows: Section 2 details our SCM programming model while Section 3 surveys related work. Section 4 presents our design goals and a description of our contributed design principles. Then, Section 5 explains the base operations of the FPTree. Thereafter, we discuss the results of our experimental evaluation in Section 6. Finally, Section 7 concludes the paper.

2. SCM PROGRAMMING MODEL

In the following we discuss SCM programming challenges and detail how we solve them with our programming model.

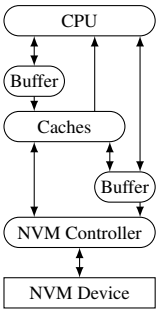


Figure 1: Volatility chain in x86-like processors.

Data consistency. SCM is accessed via a long volatility chain, illustrated in Figure 1, that includes store buffers, CPU caches, and the memory controller buffers, over all of which software has little control. The SNIA [3] recommends to manage SCM using an SCM-aware file system that grants the application layer direct access to SCM with *mmap*, enabling load/store semantics. Hence, ordering and durability of SCM writes cannot be guaranteed without software effort. To address this problem, we use, similarly to state-of-the art, persistence primitives, namely CLFLUSH, MFENCE, SFENCE, and non-temporal writes. CLFLUSH evicts a cache line from the cache and writes its content back to memory. When an MFENCE is issued, all pending load and store memory operations are completed before the program proceeds further. As for non-temporal writes, they bypass the cache and are buffered in a special buffer that is flushed when it is full or when an MFENCE is issued. Additionally, hardware vendors have announced new instructions to enhance SCM programming performance. For example, Intel has announced the CLFLUSHOPT (an optimized version of CLFLUSH), PCOMMIT, and cache line write back (CLWB) instructions [2]. When a PCOMMIT is issued, all pending writes that were evicted from the cache are made durable. Contrary to CLFLUSH, CLWB does not evict the cache line but simply writes it back, which can lead to significant performance gains when the cache line is re-used shortly after it was written back. In this work we assume the existence of a function *Persist* that implements the most efficient way of making data durable. In our evaluation systems, this function can correspond to either a CLFLUSH wrapped by two MFENCEs, since only MFENCE can order CLFLUSH [2], or a non-temporal write followed by an MFENCE.

Data recovery. When a program restarts, it does so with a new address space that renders all stored virtual pointers invalid. Therefore, there is a need for a mechanism that allows restoring data in SCM upon restart. We propose to use *Persistent Pointers* that encompass an 8-byte *File ID* and an 8-byte *Offset* inside that file. The file ID corresponds to a file that is created by the persistent allocator and used as an *Arena* to allocate memory. The persistent allocator provides bidirectional conversions between persistent and volatile pointers. Since persistent pointers stay valid across failures, they are used to refresh the volatile ones on restart.

Memory leaks. Since memory allocations are persistent in SCM, a memory leak would also be persistent. Hence, memory leaks have a deeper impact in SCM than in DRAM. For instance, let us consider a vector resize operation. The first step is to allocate a new, larger array. Then, the old array is copied into the new one. Finally, the old array is deallocated. If a crash occurs after the allocation of the new array but before the end of the copy phase, the persistent allocator will remember that it allocated the new array but the data structure will not, leading to a persistent memory leak. To solve this problem, we modify the interface of our persistent memory allocator so that it takes a reference to a persistent pointer that must belong to the persistent data structure that calls the allocator. In case of an allocation, the allocator persistently writes the address of the returned memory into that persistent pointer. In case of a deallocation, the allocator persistently resets (i.e., sets to null) that persistent pointer to convey that the deallocation has executed successfully. If a crash occurs during an allocation, upon recovery the allocator will either complete or roll-back the allocation, and the data structure will inspect the persistent pointer it used for the allocation. If the latter is not null, the data structure will know that it received memory. In brief, the task of discovering memory leaks is split between the allocator and the data structure.

Partial writes. Since writes to SCM have word granularity, partial writes may occur. Indeed, if a failure happens while writing data that is larger than the supported *p-atomic* write size, there is no guarantee on how much data was written. In contrast to *atomic* in the sense of concurrency, we mean by *p-atomic* a write that executes in one CPU cycle, i.e., a write that is immune to the partial write problem. To address this problem, we use flags that can be written p-atomically to indicate whether a larger write operation has completed. In this work we assume that only 8-byte writes are p-atomic.

3. RELATED WORK

Persistent trees. Data structures are traditionally persisted using undo-redo logging and shadowing techniques. The rise of flash memory lead to the emergence of new optimized data structures such as the Bw-Tree [19]. However, these remain intrinsically tied to the logging and paging mechanisms, which SCM can completely do without. Indeed, the advent of SCM enables novel and more

fine-grained techniques in designing persistent data structures, but it presents unprecedented challenges as discussed in Section 2. Several works proposed global solutions to these challenges, such as NVHeap [10], Mnemosyne [26], and REWIND [7], which are based on garbage collection and software transactional memory. However, these solutions incur additional overhead due to systematic logging of changed data, which adds up to the overhead caused by the higher latency of SCM. Another line of work relies on the persistence primitives provided by the CPU, namely memory barriers, cache line flushing instructions, and non-temporal stores to design persistent data structures that are stored in and accessed directly from SCM.

In this context, Venkataraman et al. [24] proposed the CDDS B-Tree, a persistent and concurrent B-Tree that relies on versioning to achieve consistency. It recovers from failures by retrieving the version number of the latest consistent version and removing changes that were made past that version. Nevertheless, its scalability suffers from using a global version number, and it requires garbage collection to remove old versions. Chen et al. [8] proposed to use unsorted nodes with bitmaps to decrease the number of expensive writes to SCM. They extended their work by proposing the write-atomic B-Tree (wBTree) [9], a persistent tree that relies on the atomic update of the bitmap to achieve consistency, and on undo-redo logs for more complex operations such as splits. It employs sorted indirection slot arrays in nodes to avoid linear search and enable binary search. Following another approach, Yang et al. [28] proposed the NV-Tree, a persistent and concurrent B-Tree based on the CSB$^+$-Tree [23]. They proposed to enforce the consistency of leaf nodes while relaxing that of inner nodes, and rebuilding them in case of failure. This approach assumes an SCM-only configuration, while our proposed *Selective Persistence* assumes a hybrid SCM-DRAM configuration, stores only primary data in SCM, and keeps the rest in DRAM to enable better performance. The NV-Tree keeps inner nodes contiguous in memory and uses unsorted leaves with an append-only strategy. This design implies the need for costly rebuilds when a leaf parent node overflows, and leads to a high memory footprint.

While the wBTree and the NV-Tree perform better than existing persistent trees, including the CDDS B-Tree, their performance is still significantly slower than fully transient counterparts. Additionally, they are oblivious to the problem of persistent memory leaks. Indeed, both the wBTree and the NV-Tree do not log the memory reference of a newly allocated or deallocated leaf, which makes these allocations prone to *persistent* memory leaks. Moreover, while the NV-Tree can retrieve its data thanks to using offsets, the wBTree does not elaborate on how data is recovered: It uses volatile pointers which become invalid after a restart, making data recovery practically infeasible. Our proposed FPTree successfully solves these issues.

Other related work. Arulraj et al. [5] studied different SCM-based database architectures and recovery strategies. They concluded that using in-place updates is the optimal approach as there is no need to reload primary data or to apply a redo log upon recovery since transaction changes are made persistent at commit time. Oukid et al. [21] proposed a novel database architecture where data is stored in and directly accessed from SCM, eliminating the need for traditional logging mechanisms. Kimura [18] presented FOEDUS, a novel database architecture that aims at scaling with many cores and very large SCM capacity. The FPTree can be a building block for such systems since it is based on the same architectural assumptions.

To manage SCM, Condit et al. [11] presented BPFS, a high performance transactional file system that runs on top of SCM. Nayaranan et al. [20] proposed Whole System Persistence where data is flushed only on power failures using the residual energy of the system, but do not consider software failures. Zhao et al. [30] proposed Kiln, a persistent memory design that leverages SCM to offer persistent in-place updates without logging or copy-on-write. However, they assume a non-volatile last level cache. Several works used SCM to optimize OLTP durability management, mainly by optimizing the logging infrastructure [13, 22, 27]. Finally, other works focused on optimizing database algorithms on SCM such as sorts and joins, but without leveraging the non-volatility property of SCM [8, 25].

4. FPTree DESIGN PRINCIPLES

We put forward the following design goals for the FPTree:

1. *Persistence.* The FPTree must be able to self-recover to a consistent state from any software crash or power failure scenario. We do not cover hardware failures in this work.
2. *Near-DRAM performance.* The FPTree must exhibit similar performance to transient counterparts.
3. *Robust performance.* The performance of the FPTree must be resilient to higher SCM latencies.
4. *Fast recovery.* The recovery time of the FPTree must be significantly faster than a complete rebuild of a transient B$^+$-Tree.
5. *High scalability.* The FPTree should implement a robust concurrency scheme that scales well in highly concurrent situations.
6. *Variable-size keys support.* The FPTree should support variable-size keys (e.g. strings) which is a requirement for many systems.

Figure 2 depicts the inner node and leaf node layouts of the FPTree. Since inner nodes are kept in DRAM, they have a classical main memory structure with sorted keys. Leaf nodes however keep keys unsorted and use a bitmap to track valid entries in order to reduce the number of expensive SCM writes, as first proposed by Chen et al [8]. Additionally, leaf nodes contain fingerprints which are explained in detail in Section 4.2. The next pointers in the leaves are used to form a linked list whose main goals are: (1) enable range queries, and (2) allow the traversal of all the leaves during recovery to rebuild inner nodes. The next pointers need to be *Persistent Pointers* in order to remain valid across failures. Finally, a one-byte field is used as a lock in each leaf.

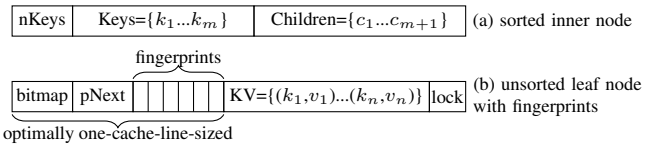


Figure 2: FPTree inner node and leaf node layouts.

In the following we present our proposed design principles that enable us to achieve the above design goals.

4.1 Selective persistence

Selective persistence can be described as keeping in SCM the minimal set of primary data on which all the implementation effort for consistency will be focused, and rebuilding all non-primary data that is placed in DRAM upon recovery. Applied to a B$^+$-Tree, as illustrated in Figure 3, the leaf nodes are placed in SCM using a persistent linked-list, while inner nodes are placed in DRAM and can be rebuilt as long as the leaves are in a consistent state. As a result only accessing the leaves is more expensive compared to a transient B$^+$-Tree. In addition, inner nodes represent only a small fraction of the total size of the B$^+$-Tree. Hence, selective persistence should enable our persistent tree to have similar performance to that of a transient B$^+$-Tree, while using only a minimal portion of DRAM. Our approach differs from that of the NV-Tree [28] in its underlying hardware assumptions: We assume a hybrid SCM-DRAM configuration while the NV-Tree assumes an SCM-only configuration.

In a nutshell, inner nodes will keep a classical structure and fully reside in DRAM without needing any special implementation effort, while leaf nodes will fully reside in SCM and require special care to

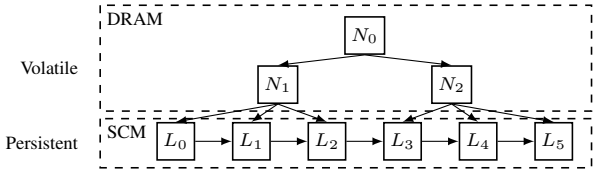


Figure 3: Selective persistence applied to a B$^+$-Tree: inner nodes are kept in DRAM while leaf nodes are kept in SCM.

ensure their consistency. This interplay between SCM and DRAM is crucial for our concurrency mechanism we present in Section 4.4.

4.2 Fingerprints

Unsorted leaves require an expensive linear scan in SCM. To enable better performance, we propose a technique called *Fingerprinting*. Fingerprints are one-byte hashes of leaf keys, stored contiguously at the beginning of the leaf as illustrated in Figure 2. By scanning them first during a search, fingerprints act as a filter to avoid probing keys that have a fingerprint that does not match that of the search key. In the following we show that, theoretically, Fingerprinting enables a much better performance than the wBTree and the NV-Tree. We consider only the case of unique keys, which is often an acceptable assumption in practice [15]. We demonstrate that, using Fingerprinting, the expected number of in-leaf key probes during a successful search is equal to *one*. In the following we compute this expected number. We assume a hash function that generates uniformly distributed fingerprints. Let m be the number of entries in the leaf and n the number of possible hash values ($n = 256$ for one-byte fingerprints).

First, we compute the expected number of occurrences of a fingerprint in a leaf, denoted $E[K]$, which is equivalent to the number of hash collisions in the fingerprint array plus one (since we assume that the search key exists):

$$E[K] = \sum_{i=1}^{m} i \cdot P[K = i]$$

where $P[K = i]$ is the probability that the search fingerprint has exactly i occurrences knowing that it occurs at least once. Let A and B be the following two events:

- A: the search fingerprint occurs exactly i times;
- B: the search fingerprint occurs at least once.

Then, $P[K = i]$ can be expressed with the conditional probability:

$$P[K = i] = P[A|B] = \frac{P[A \cap B]}{P[B]} = \frac{\left(\frac{1}{n}\right)^i \left(1 - \frac{1}{n}\right)^{m-i} \binom{m}{i}}{1 - \left(1 - \frac{1}{n}\right)^m}$$

The nominator reflects the binomial distribution, while the denominator reflects the condition, i.e., the probability that at least one matching fingerprint exists, expressed as the complementary probability of that of no matching fingerprint exists.

Knowing the expected number of fingerprint hits, we can determine the expected number of in-leaf key probes, denoted $E_{FPTree}[T]$, which is the expected number of key probes in a *linear search* of length $E[K]$ on the keys indicated by the fingerprint hits:

$$E[T] = \frac{1}{2}(1 + E[K]) = \frac{1}{2}\left(1 + \sum_{i=1}^{m} i \frac{\left(\frac{1}{n}\right)^i \left(1 - \frac{1}{n}\right)^{m-i} \binom{m}{i}}{1 - \left(1 - \frac{1}{n}\right)^m}\right)$$

$$= \frac{1}{2}\left(1 + \frac{\left(\frac{n-1}{n}\right)^m}{1 - \left(\frac{n-1}{n}\right)^m} \sum_{i=1}^{m} \frac{i\binom{m}{i}}{(n-1)^i}\right)$$

$$= \frac{1}{2}\left(1 + \frac{\left(\frac{n-1}{n}\right)^m}{1 - \left(\frac{n-1}{n}\right)^m} \left(\frac{m}{n-1}\right) \sum_{i=0}^{m-1} \frac{\binom{m-1}{i}}{(n-1)^i}\right)$$

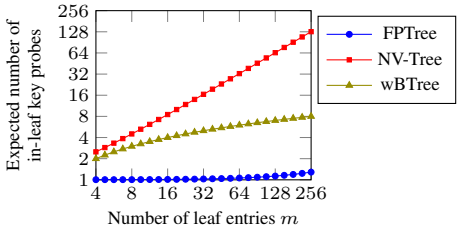


Figure 4: Expected number of in-leaf key probes during a successful search operation for the FPTree, NV-Tree, and wBTree.

By applying the binomial theorem on the sum we get:

$$E[T] = \frac{1}{2}\left(1 + \frac{\left(\frac{n-1}{n}\right)^m}{1 - \left(\frac{n-1}{n}\right)^m} \left(\frac{m}{n-1}\right) \left(\frac{n}{n-1}\right)^{m-1}\right)$$

$$= \frac{1}{2}\left(1 + \frac{m}{n\left(1 - \left(\frac{n-1}{n}\right)^m\right)}\right)$$

The wBTree is able to use binary search thanks to its sorted indirection slot arrays, hence, its expected number of in-leaf key probes is: $E_{\text{wBTree}}[T] = log_2(m)$. The NV-Tree scans a leaf by performing a reverse linear search, starting from the last entry of the leaf, so that if a matching key is found, it is guaranteed to be the most recent version. Then, the expected number of in-leaf key probes during a search operation for the NV-Tree is that of a linear search: $E_{\text{NV-Tree}}[T] = \frac{1}{2}(m + 1)$. Figure 4 shows the expected number of in-leaf key probes for the FPTree, the wBTree, and the NV-Tree. We observe that the FPTree theoretically enables a much better performance than the wBTree and NV-Tree. For instance, for $m = 32$, the FPTree needs a single in-leaf key probe, while the wBTree and NV-Tree need 5 and 16, respectively. Basically, fingerprinting requires less than two key probes in average up to $m \approx 400$. The wBTree outperforms the FPTree only starting from $m \approx 4096$. It is important to note that, in the case of variable-size keys, since only key references are kept in leaf nodes, every key probe results in a cache miss. Hence, every saved in-leaf key probe is a saved cache miss to SCM. This theoretical result is verified by our experimental evaluation as shown in Section 6.2.

4.3 Amortized persistent memory allocations

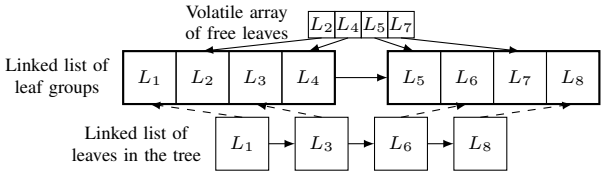


Figure 5: FPTree leaf groups management.

Since persistent allocations are expensive, allocating new leaves during splits is costly to the overall performance of the persistent tree. To remedy this issue, we propose to amortize the overhead of those allocations by allocating blocks of multiple leaves at once. As illustrated in Figure 5, leaves are managed through two structures:

- A linked-list of groups of leaves currently allocated;
- A volatile array of leaves currently free and not used in the tree.

Two methods are used to get and free leaves:

- *GetLeaf:* if the vector of free leaves is not empty, we pop the last element and return it. If it is empty, we allocate a new leaf group, append it to the linked list of groups, and add its leaves to the vector of free leaves, except the one that we return.
- *FreeLeaf:* when a leaf is freed, it is pushed into the vector of free leaves. If its corresponding leaf group is completely free, it is deallocated.

Using leaf groups allows to decrease the number of expensive persistent memory allocations which leads to better insertion performance, as shown in Section 6.2.

4.4 Selective concurrency

Transactional memory is a tool that simplifies concurrency by making a batch of writes atomically visible using transactions. HTM is hardware supported transactional memory. Several hardware vendors provide HTM, such as IBM on Blue Gene/Q and POWER8 processors, and Intel with Transactional Synchronization Extensions (TSX). Although we use Intel TSX in this work, our algorithms are valid for any currently available HTM implementation.

From a programmer's point of view, HTM is used as a coarse-grained lock around a critical section, but it behaves from the hardware point of view as a fine-grained lock: conflicts are detected between transactions at the granularity of a cache line. In the case of TSX, the critical code is put inside a transaction using the XBEGIN and XEND instructions. When a thread reaches the XBEGIN instruction, the corresponding lock is first read, and if it is available, a transaction is started without acquiring the lock. All changes made inside a transaction are made atomically visible to other threads by an atomic commit if the transaction is successful, that is, if it reaches the XEND instruction without detecting any conflicts. If the transaction aborts, all the buffered changes are discarded, and the operation is re-executed following a programmer-defined fall-back mechanism. In our implementation we use the Intel Threading Building Block[1] *speculative spin mutex* that uses a global lock as a fall-back mechanism.

To detect conflicts, each transaction keeps read and write sets in L1 cache. The read set comprises all the memory cache lines that the transaction reads, and the write set consists of all the memory cache lines that the transaction writes to. A conflict is detected by the hardware if a transaction reads from the write set of another transaction or writes to the read or write set of another transaction. When this happens, one of the two transactions is aborted. In this case, the aborted transaction falls back to a programmer-defined concurrency mechanism. To increase performance, a transaction is allowed to retry a few times before resorting to the fall-back mechanism. This is an optimistic concurrency scheme as it works under the assumption that only few conflicts will occur and the transactions will execute lock-free with high probability.

HTM is implemented in current architectures by monitoring changes using the L1 cache. Consequently, CPU instructions that affect the cache, such as CLFLUSH, are detected as conflicts which triggers the abortion of a transaction if they are executed within its read or write sets. A trivial implementation of HTM-based lock-elision for a standard B$^+$-Tree would be to execute its base operations within HTM transactions [17]. However, insert and delete operations would need to flush the modified records in the leaf, thus aborting the transaction and taking a global lock. This means that all insertion and deletion operations will in practice get serialized. Therefore, there is an apparent incompatibility between the use of HTM on the one hand and our need to flush the data to persistent memory to ensure consistency on the other hand. To solve this issue, we propose to use different concurrency schemes for the transient and the persistent parts of the data, in the same way we applied different consistency schemes for the data stored in DRAM and in SCM. We name our approach *Selective Concurrency*.

As illustrated in Figure 6, Selective Concurrency consists in performing the work that does not involve modifying persistent data inside an HTM transaction, and the work that requires persistence primitives outside HTM transactions. In the case of our FPTree, the traversal of the tree and changes to inner nodes only involve transient data. Therefore, these operations are executed within an HTM transaction and are thus protected against other operations done in concurrent HTM transactions. For other operations that cannot be executed inside the transaction, fine-grained locks are used. Basically, the leaf nodes to be modified are locked during the traversal of the tree inside the HTM transaction. After all modifications on the transient part are done, the transaction is committed. The remaining modifications on the persistent part are then processed outside of the transaction. Finally, the locks on the persistent part are released. The implementation of the base operations using TSX are detailed in Section 5.

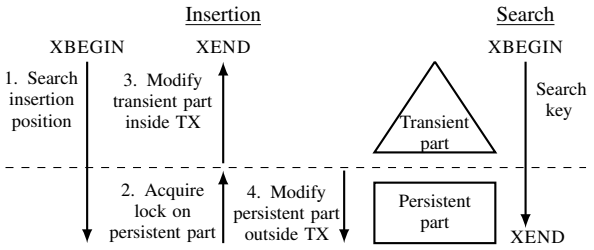


Figure 6: Selective Concurrency applied to a hybrid data structure.

5. BASE OPERATIONS

In this work, we implement three different persistent trees:

1. *FPTree.* It is the single-threaded version that implements selective persistence, fingerprinting, amortized allocations, and unsorted leaves.
2. *Concurrent FPTree.* This version implements selective persistence, selective concurrency, fingerprinting, and unsorted leaves. As for amortized allocations, since they constitute a central synchronization point, we found that they hinder scalability. Hence, they are not used in this version.
3. *PTree.* It reflects a light version of the FPTree that implements only selective persistence and unsorted leaves. Contrary to the FPTree and the wBTree, it keeps keys and values in separate arrays for better data locality when linearly scanning the keys.

In the following we discuss the base operations of the FPTree with fixed-size keys and explain how it can recover in a consistent state from any software crash or power failure scenario. A similar discussion is available for variable-size keys in Appendix C. Moreover, the algorithms for managing the leaf groups in the single-threaded version of the FPTree are explained in Appendix B. In the following, *speculative lock* denotes a TSX-enabled lock. Since leaf locks are only taken within TSX transactions, there is no need to modify them using atomics. Indeed, if many threads try to write the same lock –thus writing to the same cache line–, only one will succeed and the others will be aborted.

Find

Since search operations do not modify persistent data, they can be fully wrapped in a TSX transaction that protects them from conflicting with write operations of other threads. If another thread writes to a location read by the thread performing the lookup, the transaction will abort and retry, eventually taking a global lock if the retry threshold is exceeded. Search operations still need to check for any non-TSX lock (i.e., leaf locks) that might have been taken by another thread to perform changes outside a TSX transaction. Algorithm 1 shows the pseudo-code for the *Find* operation.

The *concurrent Find* is executed until it succeeds, making it retry as long as it aborts, which occurs either because the target leaf is locked or because a conflict has been detected. The speculative lock

[1] https://www.threadingbuildingblocks.org/

can execute in two ways: either a regular TSX transaction is started with the XBEGIN instruction as long as the TSX retry threshold is not reached, or a global lock is taken.

Algorithm 1 ConcurrentFind(Key K)

```
 1: while TRUE do
 2:     speculative_lock.acquire();
 3:     Leaf = FindLeaf(K);
 4:     if Leaf.lock == 1 then
 5:         speculative_lock.abort();
 6:         continue;
 7:     for each slot in Leaf do
 8:         set currentKey to key pointed to by Leaf.KV[slot].PKey
 9:         if Leaf.Bitmap[slot] == 1 and Leaf.Fingerprints[slot] == hash(K)
            and currentKey == K then
10:             Val = Leaf.KV[slot].Val;
11:             Break;
12:     speculative_lock.release();
13:     return Val;
```

The way a transaction is aborted depends on the current situation: if a regular TSX transaction was started, the XABORT instruction will undo all changes (not relevant here since no changes were made), and rewind to the XBEGIN instruction. If the global lock was taken, the XABORT instruction has no effect. In this case, the ***while*** loop and the continue directive play the role of aborting and retrying. Upon reaching the leaf, and if its lock is not already taken, a lookup for the key is performed without taking the leaf lock. If another thread takes the leaf lock during the search for the key, a conflict is detected and one of the two transactions is aborted.

Insert

Algorithm 2 presents the pseudo-code of the *concurrent Insert* operation. It follows three steps: (1) Inside a TSX transaction, the tree is traversed to reach a leaf and lock it, and the information of whether a split is needed is propagated; (2) Outside a TSX transaction, changes to the insertion leaf are applied: a leaf split if needed, and the insertion of the key-value pair in the leaf. These operations use persistence primitives which would trigger an abort if used inside a TSX transaction; (3) If a leaf split occurred in step 2, the inner nodes are updated inside a second TSX transaction. This can be done in two ways: either by directly updating the leaf's parent node if it did not split while modifying the leaf, or by re-traversing the tree. The leaf lock is then released by setting it to 0. One can notice in Algorithm 2 that no **while** loop is used for the second transaction since inner nodes have no locks. Hence, there is no need to manually abort the transaction.

When no leaf split is needed, the key-value pair and the fingerprint are written to their respective slots and persisted. The order of these writes does not matter since they are not visible as long as the bitmap is not updated. Therefore, in case of a crash while writing the key and the value, no action is needed and the operation is considered as not completed. The bitmap is then p-atomically updated and persisted. In brief, if a failure occurs before the bitmap is persisted, the key-value pair was not inserted, otherwise, the operation successfully executed. In both cases, no action is required.

Algorithm 3 represents the pseudo-code for a leaf split. To ensure the consistency of the tree in case of failure during a split, we need to use a micro-log that consists of two persistent pointers: one pointing to the leaf to split, denoted *PCurrentLeaf*, and another pointing to the newly allocated leaf, denoted *PNewLeaf*. Basically, the concurrent FPTree contains split and delete micro-log arrays that are indexed by transient lock-free queues. A micro-log is provided by the lock-free queue and is returned at the end of the operation. The split recovery function is shown in Algorithm 4. The split operation starts by writing to *PCurrentLeaf* the persistent address of the leaf to split. If a failure occurs at this point, we only need to reset the micro-log, since *PNewLeaf* is still null. Then, we allocate a new leaf and provide a reference to *PNewLeaf* to the allocator, which persists the address of the allocated memory in *PNewLeaf* before returning. If a crash occurs at this point, the split recovery function checks whether *PNewLeaf* is null. If it is, it resets the micro-log and returns. Otherwise, it detects that the allocation did complete and thus continues the split operation. Thereafter, the content of the split leaf is persistently copied into the new leaf. The new key discriminator (split key) is then determined and the bitmap of the new leaf is updated accordingly. If a crash occurs during the latter two steps, the recovery function simply re-executes them when it detects that *PNewLeaf* is not null. Afterwards, the bitmap of the split leaf is updated and the next pointer of the split leaf is set to point to the new leaf. The latter write does not need to be p-atomic since in case of a failure at this point in time, the recovery function will redo the split starting from the copy phase. Finally, the micro-log is reset.

Algorithm 2 ConcurrentInsert(Key K, Value V)

```
 1: Decision = Result::Abort;
 2: while Decision == Result::Abort do
 3:     speculative_lock.acquire();
 4:     (Leaf, Parent) = FindLeaf(K);
 5:     if Leaf.lock == 1 then
 6:         Decision = Result::Abort; Continue;
 7:     Leaf.lock = 1;              /* Writes to leaf locks are never persisted */
 8:     Decision = Leaf.isFull() ? Result::Split : Result::Insert;
 9:     speculative_lock.release();
10: if Decision == Result::Split then
11:     splitKey = SplitLeaf(Leaf);
12: slot = Leaf.Bitmap.FindFirstZero();
13: Leaf.KV[slot] = (K, V); Leaf.Fingerprints[slot] = hash(K);
14: Persist(Leaf.KV[slot]); Persist(Leaf.Fingerprints[slot]);
15: Leaf.Bitmap[slot] = 1; Persist(Leaf.Bitmap);
16: if Decision == Result::Split then
17:     speculative_lock.acquire();
18:     UpdateParents(splitKey, Parent, Leaf);
19:     speculative_lock.release();
20: Leaf.lock = 0;
```

Algorithm 3 SplitLeaf(LeafNode Leaf)

```
 1: get μLog from SplitLogQueue;
 2: set μLog.PCurrentLeaf to persistent address of Leaf;
 3: Persist(μLog.PCurrentLeaf);
 4: Allocate(μLog.PNewLeaf, sizeof(LeafNode))
 5: set NewLeaf to leaf pointed to by μLog.PNewLeaf;
 6: Copy the content of Leaf into NewLeaf;
 7: Persist(NewLeaf);
 8: (splitKey, bmp) = FindSplitKey(Leaf);
 9: NewLeaf.Bitmap = bmp;
10: Persist(NewLeaf.Bitmap);
11: Leaf.Bitmap = inverse(NewLeaf.Bitmap);
12: Persist(Leaf.Bitmap);
13: set Leaf.Next to persistent address of NewLeaf;
14: Persist(Leaf.Next);
15: reset μLog;
```

Delete

The concurrency scheme of deletions, whose pseudo-code is shown in Algorithm 5, is very similar to that of insertions. The traversal of the tree is always done inside the TSX transaction which is aborted if the leaf is already locked by another thread. Upon reaching the leaf, three cases can arise: (1) The key to delete is not found in the leaf; (2) The leaf contains the key to delete and other keys; (3) The leaf contains only the key to delete.

Algorithm 4 RecoverSplit(SplitLog μLog)

```
1: if μLog.PCurrentLeaf == NULL then
2:     return;
3: if μLog.PNewLeaf == NULL then
4:     /* Crashed before SplitLeaf:4 */
5:     reset μLog;
6: else
7:     if μLog.PCurrentLeaf.Bitmap.IsFull() then
8:         /* Crashed before SplitLeaf:11 */
9:         Continue leaf split from SplitLeaf:6;
10:    else
11:        /* Crashed after SplitLeaf:11 */
12:        Continue leaf split from SplitLeaf:11;
```

In the first case, we simply return that the key was not found (not indicated in Algorithm 5). In the second case that is similar to an insertion with no split, the leaf is locked and the TSX transaction is committed. Outside of the transaction, the bitmap position corresponding to the value to delete is set to 0 and persisted. Then, the leaf is unlocked. As discussed earlier this operation is crash-safe. In the third case, the leaf will be deleted. The inner nodes are modified inside the TSX transaction as no persistence primitives are needed. Basically, the key and pointer corresponding to the leaf are removed from its parent, possibly triggering further inner node modifications. We note that the leaf to be deleted does not need to be locked because it will become unreachable by other threads after the update of the inner nodes completes. The only node that needs to be updated outside of the TSX transaction is the left neighbor of the deleted leaf; its next pointer is updated to point to the next leaf of the deleted leaf. Before committing the transaction, this left neighbor is retrieved and locked. Finally, outside of the transaction, the next pointer of the left neighbor is updated, its lock released, and the deleted leaf is deallocated.

Algorithm 5 ConcurrentDelete(Key K)

```
1: Decision = Result::Abort;
2: while Decision == Result::Abort do
3:     speculative_lock.acquire();
4:     /* PrevLeaf is locked only if Decision == LeafEmpty */
5:     (Leaf, PPrevLeaf) = FindLeafAndPrevLeaf(K);
6:     if Leaf.lock == 1 then
7:         Decision = Result::Abort; Continue;
8:     if Leaf.Bitmap.count() == 1 then
9:         if PPrevLeaf->lock == 1 then
10:            Decision = Result::Abort; Continue;
11:        Leaf.lock = 1; PPrevLeaf->lock = 1;
12:        Decision = Result::LeafEmpty;
13:    else
14:        Leaf.lock = 1; Decision = Result::Delete;
15:    speculative_lock.release();
16: if Decision == Result::LeafEmpty then
17:    DeleteLeaf(Leaf, PPrevLeaf);
18:    PrevLeaf.lock = 0;
19: else
20:    slot = Leaf.FindInLeaf(K);
21:    Leaf.Bitmap[slot] = 0; Persist(Leaf.Bitmap[slot]);
22:    Leaf.lock = 0;
```

Algorithm 6 shows the pseudo-code for a leaf delete operation. Similarly to leaf splits, a leaf deletion requires a micro-log to ensure consistency. It consists of two persistent pointers, denoted *PCurrentLeaf* and *PPrevLeaf* that point respectively to the leaf to be deleted and to its previous leaf. The leaf delete function first updates *PCurrentLeaf* and persists it. If *PCurrentLeaf* is equal to the head of the linked list of leaves, its head pointer must be updated. If a crash occurs at this point, the delete recovery procedure (shown in Algorithm 7) detects that *PCurrentLeaf* is set and either itself or its next pointer is equal to the head of the linked list of leaves, and continues the operation accordingly. If the leaf to be deleted is not the head of the linked list of leaves, *PPrevLeaf* is updated and persisted. Then, the next pointer of the previous leaf of the leaf to be deleted is updated to point to the next pointer of the the latter. If a crash occurs at this point, the delete recovery procedure detects that the micro-log is fully set (i.e., both pointers are not null), and thus, it repeats the latter operation. Thereafter, in both cases, the deleted leaf is deallocated by passing *PCurrentLeaf* to the deallocate function of the allocator, which resets it to null. If a crash occurs at this point, the delete recovery function detects that *PCurrentLeaf* is null and resets the micro-log. Finally, the micro-log is reset.

Algorithm 6 DeleteLeaf(LeafNode Leaf, LeafNode PPrevLeaf)

```
1: get the head of the linked list of leaves PHead
2: get μLog from DeleteLogQueue;
3: set μLog.PCurrentLeaf to persistent address of Leaf;
4: Persist(μLog.PCurrentLeaf);
5: if μLog.PCurrentLeaf == PHead then
6:     /* Leaf is the head of the linked list of leaves */
7:     PHead = Leaf.Next;
8:     Persist(PHead);
9: else
10:    μLog.PPrevLeaf = PPrevLeaf;
11:    Persist(μLog.PPrevLeaf);
12:    PrevLeaf.Next = Leaf.Next;
13:    Persist(PrevLeaf.Next);
14: Deallocate(μLog.PCurrentLeaf);
15: reset μLog;
```

Algorithm 7 RecoverDelete(DeleteLog μLog)

```
1: get head of linked list of leaves PHead;
2: if μLog.PCurrentLeaf != NULL and μLog.PPrevLeaf != NULL then
3:     /* Crashed between lines DeleteLeaf:12-14 */
4:     Continue from DeleteLeaf:12;
5: else
6:     if μLog.PCurrentLeaf != NULL and μLog.PCurrentLeaf == PHead
       then
7:         /* Crashed at line DeleteLeaf:7 */
8:         Continue from DeleteLeaf:7;
9:     else
10:        if μLog.PCurrentLeaf != NULL and μLog.PCurrentLeaf→Next
           == PHead then
11:            /* Crashed at line DeleteLeaf:14 */
12:            Continue from DeleteLeaf:14;
13:        else
14:            reset μLog;
```

Update

Algorithm 8 presents pseudo-code for the update operation. Although the update operation looks like an insert-after-delete operation, it is in fact much more optimized. Indeed, the update operation relies on the fact that the bitmap can be updated p-atomically to reflect both the insertion and the deletion at the same time. This has the advantage of maintaining the leaf size unchanged during an update, which makes leaf splits required only if the leaf that contains the record to update is already full. As for the recovery logic, it is exactly the same as for insertions, where a micro-log is needed to ensure consistency only in case of a leaf split.

Recovery

Algorithm 9 shows the pseudo-code for the concurrent FPTree recovery function. It starts by checking whether the tree crashed during initialization by testing a state bit. Then, for each micro-log in the micro-logs arrays, it executes either the leaf split or the leaf delete recovery function. Afterwards, it rebuilds inner nodes by traversing

Algorithm 8 ConcurrentUpdate(Key K, Value V)

```
1: Decision = Result::Abort;
2: while Decision == Result::Abort do
3:     speculative_lock.acquire();
4:     (Decision, prevPos, Leaf, Parent) = FindKeyAndLockLeaf(K);
5:     (Leaf, Parent) = FindLeaf(K);
6:     if Leaf.lock == 1 then
7:         Decision = Result::Abort; Continue;
8:     Leaf.lock = 1;
9:     prevPos = Leaf.FindKey(K);
10:    Decision = Leaf.isFull() ? Result::Split : Result::Update;
11:    speculative_lock.release();
12: if Decision == Result::Split then
13:    splitKey = SplitLeaf(Leaf);
14: slot = Leaf.Bitmap.FindFirstZero();
15: Leaf.KV[slot] = (K, V); Leaf.Fingerprints[slot] = hash(K);
16: Persist(Leaf.KV[slot]); Persist(Leaf.Fingerprints[slot]);
17: copy Leaf.Bitmap in tmpBitmap;
18: tmpBitmap[prevSlot] = 0; tmpBitmap[slot] = 1;
19: Leaf.Bitmap = tmpBitmap; Persist(Leaf.Bitmap);
20: if Decision == Result::Split then
21:    speculative_lock.acquire();
22:    UpdateParents(splitKey, Parent, Leaf);
23:    speculative_lock.release();
24: Leaf.lock = 0;
```

the leaves, resetting their locks, and retrieving the greatest key in each leaf to use it as a discriminator key. This step is similar to how inner nodes are built in a bulk-load operation. Finally, the queues of micro-logs are rebuilt.

Please note that micro-logs are cache-line-aligned. Thus, back-to-back writes to a micro-log that are not separated by other writes can be ordered with a memory barrier and then persisted together. Indeed, when two writes target the same cache line, the first write is guaranteed to become persistent no later than the second one.

Algorithm 9 Recover()

```
1: if Tree.Status == NotInitialized then
2:     Tree.init();
3: else
4:     for each SplitLog in Tree.SplitLogArray do
5:         RecoverSplit(SplitLog);
6:     for each DeleteLog in Tree.DeleteLogArray do
7:         RecoverDelete(DeleteLog);
8: RebuildInnerNodes();
9: RebuildLogQueues();
```

Variable-size keys

To support variable-sized keys (e.g., string keys), we replace the keys in inner nodes by virtual pointers to keys and those in leaf nodes by persistent pointers to keys. Ensuring consistency does not require any additional micro-logging compared to the case of fixed-size keys, although every insert or delete operation respectively involves a persistent allocation or deallocation of a key. Appendix C elaborates in detail on the FPTree base operations for variable-size keys and their corresponding recovery procedures.

6. EVALUATION

In this section we compare the performance of the FPTree with that of state-of-the-art persistent and transient trees.

6.1 Experimental setup

In addition to the PTree and the FPTree, we re-implemented the wBTree with indirection slot arrays and the NV-Tree as faithfully as possible with regard to their description in their respective original paper. To ensure the NV-Tree has the same level of optimization as the FPTree, we placed its inner nodes in DRAM. For the same purpose, we replace the wBTree undo-redo logs with the more lightweight FPTree micro-logs. As a reference transient implementation, we use the STXTree, an open-source optimized main memory B$^+$-Tree[2]. The test programs were compiled using *GCC-4.7.2*. We use *jemalloc-4.0* as DRAM allocator, and our own persistent allocator for SCM.

Tree	Inner size	Leaf size	Key size	Memory
PTree	4096	32	8 bytes	SCM + DRAM
FPTree	4096	56	8 bytes	SCM + DRAM
NV-Tree	128	32	8 bytes	SCM + DRAM
wBTree	32	64	8 bytes	SCM
STXTree	16	16	8 bytes	DRAM
FPTreeC	128	64	8 bytes	SCM + DRAM
NV-TreeC	128	32	8 bytes	SCM + DRAM
PTreeVar	256	32	16 bytes	SCM + DRAM
FPTreeVar	2048	56	16 bytes	SCM + DRAM
NV-TreeVar	128	32	16 bytes	SCM + DRAM
wBTreeVar	32	64	16 bytes	SCM
STXTreeVar	8	8	16 bytes	DRAM
FPTreeCVar	64	64	16 bytes	SCM + DRAM
NV-TreeCVar	128	32	16 bytes	SCM + DRAM

Table 1: Chosen node sizes for the evaluated trees. Values are 8-byte integers for all trees. The suffixes *C* and *Var* indicate respectively the concurrent and variable-size keys versions of the trees.

To emulate different SCM latencies, we use a special system setup based on DRAM that provides two additional capabilities: (1) Part of the memory is separated out as a special memory region. We treat this memory region as *persistent memory*. It is managed by Persistent Memory File System (PMFS) [12], an open-source SCM-aware file system that like *tempfs*, gives direct access to the memory region with *mmap*; (2) Thanks to a special BIOS, the latency of this memory region can be configured to a specified value. A full description of this system can be found in [1].

This SCM evaluation platform is equipped with two Intel Xeon E5 processors. Each one has 8 cores, running at 2.6GHz and featuring each 32 KB L1 data and 32 KB L1 instruction cache as well as 256 KB L2 cache. The 8 cores of one processor share a 20 MB last level cache. The system has 64 GB of DRAM and 192 GB of emulated SCM. The same type of emulation system was used in [5, 12, 21]. In the experiments, we vary the latency of SCM between 160 ns (the lowest latency that can be emulated) and 650 ns. In addition, we use *ext4* with Direct Access[3] (DAX) support to emulate a DRAM-like latency of SCM (90 ns). DAX is a new feature in *ext4* that aims at supporting SCM-like devices. It is part of recent official Linux kernels releases. Similarly to PMFS, *ext4* DAX allows to do zero-copy memory mapping of files.

Unfortunately, the emulation system does not support TSX which prevents us from testing concurrency on this system. Hence, we use for concurrency tests a system equipped with two Intel Xeon E5-2699 v4 processors that support TSX. Each one has 22 cores (44 with HyperThreading) running at 2.1GHz. The system has 128 GB of DRAM. The local-socket and remote-socket DRAM latencies are respectively 85 ns and 145 ns. We mount *ext4 DAX* on a reserved DRAM region to emulate SCM.

We conducted a preliminary experiment to determine the best node sizes for every tree considered in the evaluation. Table 1 summarizes the results. We note that the best node sizes for the single-threaded and the concurrent versions of the FPTree differ. This is because larger inner nodes increase the probability of conflicts inside TSX transactions.

[2] https://panthema.net/2007/stx-btree/

[3] https://www.kernel.org/doc/Documentation/filesystems/dax.txt

6.2 Single-threaded micro-benchmarks

In this sub-section, we focus on the single-threaded performance of the base operations of the trees. In all our micro-benchmarks, we use uniformly distributed generated data. For fixed-size keys, keys and values are 8-byte integers, while for variable sized-keys, keys are 16-byte strings and values are 8-byte integers.

Base operations

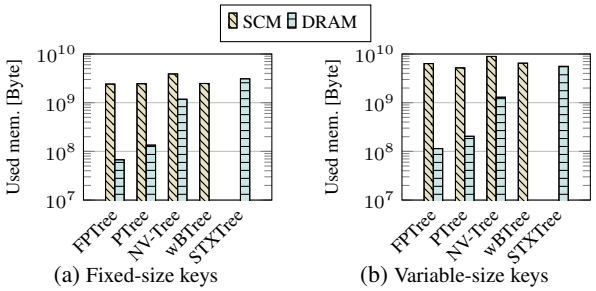
To study the performance of the *Find*, *Insert*, *Update*, and *Delete* operations, we first warm up the trees with 50M key-values, then we execute back-to-back 50M *Finds*, *Inserts*, *Updates*, and *Deletes*. Figure 7 shows the performance results for different SCM latencies. For fixed-size keys, we observe that with an SCM latency of 90 ns (DRAM's latency), the FPTree outperforms the PTree, NV-Tree, and wBTree by 1.18x/1.48x/1.08x/1.08x, 1.47x/1.83x/2.63x/1.79x, and 1.97x/2.05x/1.81x/2.05x for *Find*/*Insert*/*Update*/*Delete*, respectively. The speedups increase up to 1.19x/1.63x/1.08x/1.02x, 1.92x/2.97x/3.65x/2.82x, and 3.93x/4.89x/4.09x/5.48x, respectively, with an SCM latency of 650 ns. We notice that compared with the STXTree, the FPTree and the PTree have better *Delete* performance with the lower SCM latencies because deletions consist in simply flipping a bit in the bitmap, while it consists in a sorted delete for the STXTree. Besides, the FPTree exhibits a slowdown of only 1.51x/1.67x/1.93x/1.21x and 2.56x/2.17x/2.76x/1.70x, for SCM latencies of 250 ns and 650 ns, respectively. The slowdowns for the PTree, NV-Tree, and wBTree are respectively 1.76x/2.46x/2.04x/1.21x, 2.47x/3.51x/5.78x/2.60x, and 4.32x/5.16x/5.25x/4.26x for an SCM latency of 250 ns, and 3.05x/3.55x/2.99x/1.73x, 4.92x/6.46x/10.09x/4.79x, and 10.07x/10.63x/11.30x/9.31x for an SCM latency of 650 ns. Moreover, we notice that the difference in performance between the FPTree and the PTree is greater for *Inserts* than for *Finds* and *Deletes*, showing the benefit of using leaf groups in addition to fingerprints.

At an SCM latency of 650 ns, the average time of an FPTree *Find* is 1.3 µs, which corresponds to the cost of two SCM cache misses: the first one to access the bitmap and fingerprints of the leaf node, and the second one to access the searched key-value. This is in line with our theoretical result of Section 4.2.

As for variable-size keys, the benefit of using fingerprints is more salient since any additional string key comparison involves a pointer dereferencing, and thus a cache miss. Indeed, for *Find*/*Insert*/*Update*/*Delete*, the FPTree outperforms the PTree, NV-Tree, and wBTree by 1.82x/1.65x/1.71x/1.24x, 1.72x/1.93x/4.80x/1.78x, and 2.20x/1.62x/2.15x/1.64x respectively for an SCM latency of 90 ns, and by increased speedups of 2.15x/2.16x/2.36x/1.35x, 2.50x/2.42x/8.19x/2.14x, and 2.93x/2.93x/5.43x/2.86x for an SCM latency of 650 ns. Another important observation is that the curves of the FPTree tend to be more flattened than those of the other trees, which denotes a decreased dependency with respect to the latency of SCM. Additionally, the FPTree outperforms the STXTree for *Find*/*Insert*/*Update*/*Delete* by 2.10x/ 1.13x/1.71x/1.22x at a latency of 90 ns, and thanks to fingerprinting, it is still 1.23x/1.02x faster for *Find*/*Update* at a latency of 650 ns, while the STXTree is 2.04x/1.99x faster for *Insert*/*Delete*.

Recovery

We evaluate the recovery performance of the trees, both for fixed-size and variable-size keys for different tree sizes and SCM latencies. Figure 7 depicts the experimental results. Since the wBTree resides fully in SCM, it exhibits constant recovery time, in the order of one millisecond, and thus it is not depicted in the figure. We observe that for a tree size of 100M entries, the FPTree recovers 1.52x/2.93x, and 5.97x/6.38x faster than the PTree and the NV-Tree for an SCM latency of 90 ns/650 ns, respectively. This difference between the FPTree and the PTree is explained by the leaf groups that provide more data locality when traversing the leaves. The slower recovery time of the NV-Tree is due to the inner nodes being rebuilt in a sparse way, requiring a large amount of DRAM to be allocated. The recovery of the FPTree is 76.96x and 29.62x faster than a full rebuild of the STXTree in DRAM, for an SCM latency of 90 ns and 650 ns, respectively.



(a) Fixed-size keys (b) Variable-size keys

Figure 8: DRAM and SCM consumption of trees with 100M key-value: 8-byte and 16-byte keys for fixed-size and variable-size keys versions, respectively.

Regarding variable-size keys, most of recovery time of the PTree, the FPTree and the NV-Tree is spent dereferencing the persistent pointers to the keys in the leaves to retrieve the greatest key in each leaf. The rebuild of the inner nodes itself represents only a minor part in the total rebuild time. This explains why all three implementations tend to perform the same for recovery. Nevertheless, for a tree size of 100M entries, the recovery of the FPTree is 24.68x and 5.68x faster than a full rebuild of the STXTree in DRAM, for an SCM latency of 90 ns and 650 ns, respectively.

Memory consumption

Figure 8 shows DRAM and SCM consumption of the trees with 100M key-values and a node fill ratio of ~70%. Since the wBTree resides fully in SCM, it does not consume DRAM. We observe that for fixed-size keys, the FPTree needs 2.24 GB of SCM and only 64.12 MB of DRAM, i.e., only 2.71% of the total size of the tree. The PTree needs slightly more DRAM (5.19% of the size of the tree) because of its smaller leaves that lead to more inner nodes. The NV-Tree requires much more SCM and DRAM than the FPTree and the PTree: 3.62 GB of SCM and 1.09 GB of DRAM–which corresponds to 23.19% of the size of the tree. On the one side, the increase in SCM consumption is due to the additional flag that is added to every key-value and to the alignment of the leaf entries to be cache-line-aligned. On the other side, the increase in DRAM consumption is due to creating one leaf parent per leaf node when rebuilding the contiguous inner nodes. We observe similar results with variable-size keys, where consumed DRAM corresponds to 1.76% (108.73 MB), 3.78% (194.56 MB), and 12.67% (1.20 GB) of the total size of the tree for the FPTree, PTree, and NV-Tree, respectively. We note that the FPTree consumes slightly more SCM than the PTree due to the additional fingerprints in the leaves. The most salient observation is that the FPTree consumes one order of magnitude less DRAM than the NV-Tree.

6.3 Concurrent micro-benchmarks

For concurrency experiments, we use the HTM system that supports Intel TSX. We compare the fixed-size and variable-size concurrent versions of the FPTree and the NV-Tree. The experiments consist in warming up the trees with 50M key-values, then executing in order 50M concurrent *Finds*, *Inserts*, *Updates*, and *Deletes* with a fixed number of threads. We also consider a mixed workload made of 50% *Inserts* and 50% *Finds*. To eliminate result variations, we use the *numactl* utility to bind every thread to exactly one core. The resource allocation strategy is to first allocate all physical cores

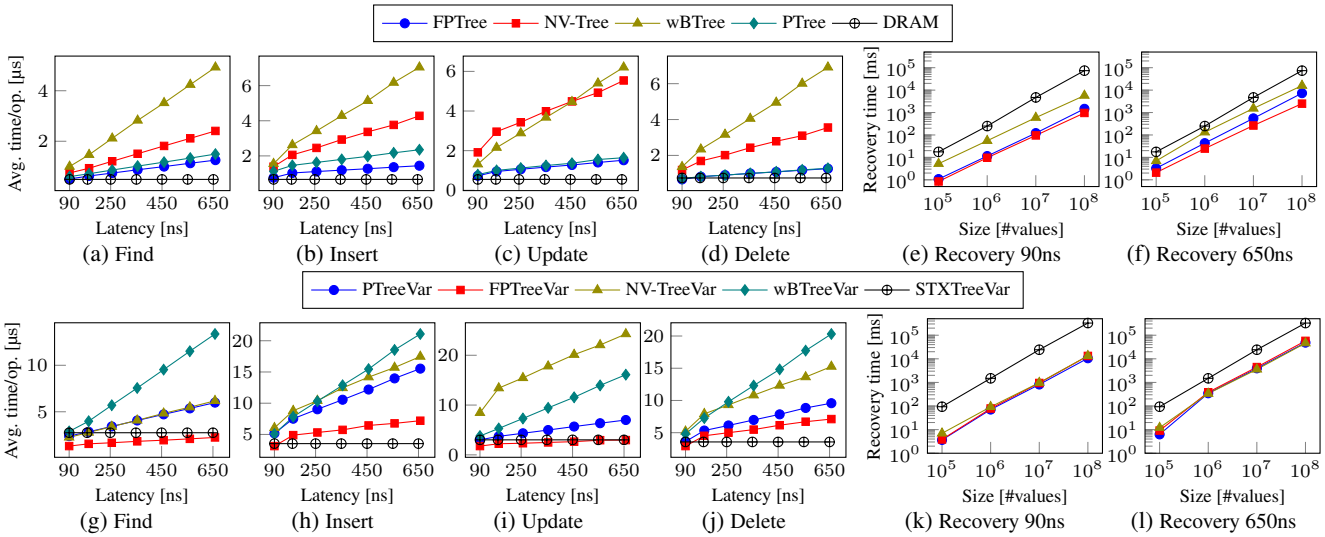


Figure 7: Effect of the latency of SCM on treess *Find*, *Insert*, *Update*, *Delete*, and *Recovery* operations performance.

before allocating the HyperThreads. When the two sockets are used, the cores are split equally between them.

We evaluate concurrency in three scenarios: on a single socket (up to 44 logical cores), on two sockets (up to 88 logical cores), and on one socket with a higher SCM latency. For each scenario we depict throughput figures alongside speedup figures over single-threaded execution. Ideal speedup is depicted with the mark-less line.

Single-socket experiments

Figure 9 depicts the results for scalability experiments on one socket. We observe that the FPTree scales well for both fixed-size and variable-size keys throughout the range of threads considered. In the case of fixed-size keys, performance is increased with 22 threads over single-threaded execution by a factor of 18.3/18.4/18.3/18.5/18.4 for the *Find*/*Insert*/*Update*/*Delete*/*Mixed* benchmarks, respectively. With 44 threads, and thus HyperThreading limiting the scaling, performance is increased over the execution with 22 threads by a factor of 1.57/1.55/1.56/1.63/1.56 for the *Find*/*Insert*/*Update*/*Delete*/*Mixed* benchmarks, respectively. Compared with the FPTree, the NV-Tree has lower base performances, and scales less. For fixed-size keys, its performance is increased with 22 threads over single-threaded execution by a factor of 16.4/11.5/14.1/11.2/15.1 for the *Find*/*Insert*/*Update*/*Delete*/*Mixed* benchmarks, respectively. With 44 threads, performance is increased over the execution with 22 threads by a factor of 1.49/1.73/1.07/1.74/1.61 for the *Find*/*Insert*/*Update*/*Delete*/*Mixed* benchmarks, respectively. We witness the same scalability patterns with variable-size keys. Between 45 and 88 threads, the performance of both the FPTree and the NV-Tree is stable and resists over-subscription.

Two-socket experiments

Results of the two-socket experiments are presented in Figure 10. We notice that the FPTree scales well for both fixed-size and variable-size keys. In the former case, the performance of the FPTree is increased using 44 threads compared with single-threaded execution by a factor of 36.8/36.3/37.5/37.6/36.7 for the *Find*/*Insert*/*Update*/*Delete*/*Mixed* benchmarks, respectively. Using HyperThreading, performance is increased with 88 threads over the execution with 44 threads by a factor of 1.50/1.30/1.34/1.53/1.37 for the *Find*/*Insert*/*Update*/*Delete*/*Mixed* benchmarks, respectively. As for the NV-Tree, it scales less than in the single socket scenario: performance is increased with 44 threads compared with single-threaded execution by a factor of 31.2/10.5/10.9/9.1/15.6 for the *Find*/*Insert*/*Update*/*Delete*/*Mixed* benchmarks, respectively. With 88 threads, performance is increased over the execution with 44 threads by a factor of 1.18/1.81/2.01/1.97/1.61 for the *Find*/*Insert*/*Update*/*Delete*/*Mixed* benchmarks, respectively.

Regarding variable-size keys, the FPTree's performance is increased with 44 threads compared with single-threaded execution by a factor of 37.2/35.0/38.4/38.5/34.9 for the *Find*/*Insert*/*Update*/*Delete*/*Mixed* benchmarks, respectively. Using HyperThreading, performance is increased with 88 threads over the execution with 44 threads by a factor of 1.54/1.63/1.63/1.62/1.67 for the *Find*/*Insert*/*Update*/*Delete*/*Mixed* benchmarks, respectively.

Single-socket experiments with a higher latency

To emulate a higher SCM latency, we bind the CPU and DRAM resources to one socket and use the memory of the second socket as emulated SCM. Hence, latency is increased from 85 ns (local socket latency) to 145 ns (remote socket latency). Figure 11 depicts the results for scalability experiments on one socket with an SCM latency of 145 ns. We observe that both the FPTree and the NV-Tree scale the same as with an SCM latency of 85 ns. The only difference is the decrease in throughput which is expected due to the higher emulated latency of SCM. Therefore, the scalability of our selective concurrency scheme is not affected by the latency of SCM.

Overall, the FPTree has better scaling and throughput performance than the NV-Tree, both for fixed-size and variable-size keys, for the three scenarios, namely using one socket, two sockets, and one socket with a higher SCM latency.

6.4 End-to-end evaluation

We integrated the evaluated trees in two systems: a single-level database prototype, and *memcached*, a popular key-value cache.

Database experiments

We replace the dictionary index of the dictionary-encoded, columnar storage engine of the database by the evaluated trees. Only the fixed-size keys versions of the trees are needed for the dictionary index. To measure the impact of the persistent trees on transaction performance, we run the read-only queries of the Telecom Application Transaction Processing Benchmark[4] (TATP) with 50M subscribers and 8 clients on the emulation system. The database size is ~45 GB.

During the warm-up phase where the database is created, *Subsriber Ids* are sequentially generated, thus creating a highly skewed

[4] `http://tatpbenchmark.sourceforge.net/`

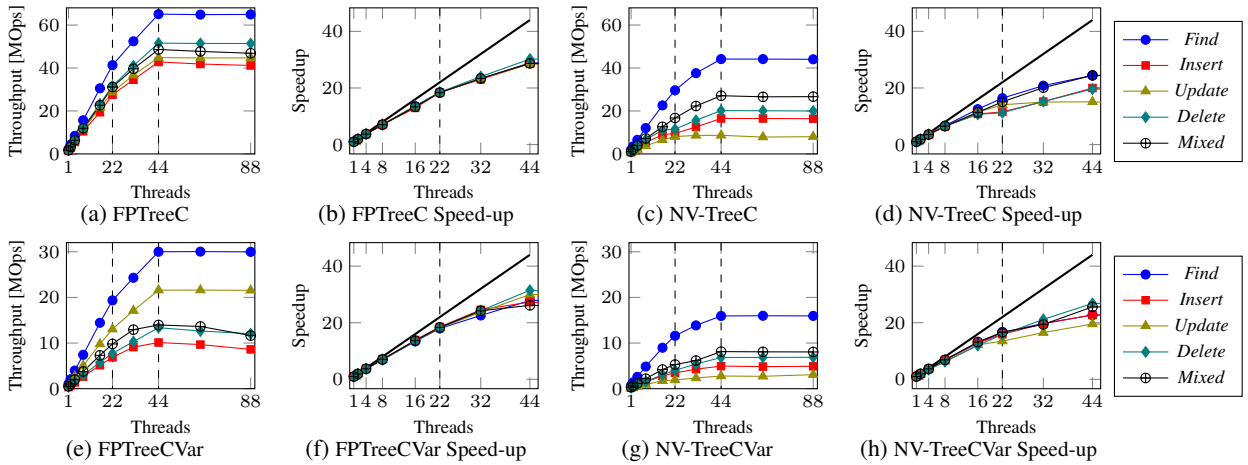


Figure 9: Concurrency performance on one socket – 50M key-values warmup followed by 50M operations.

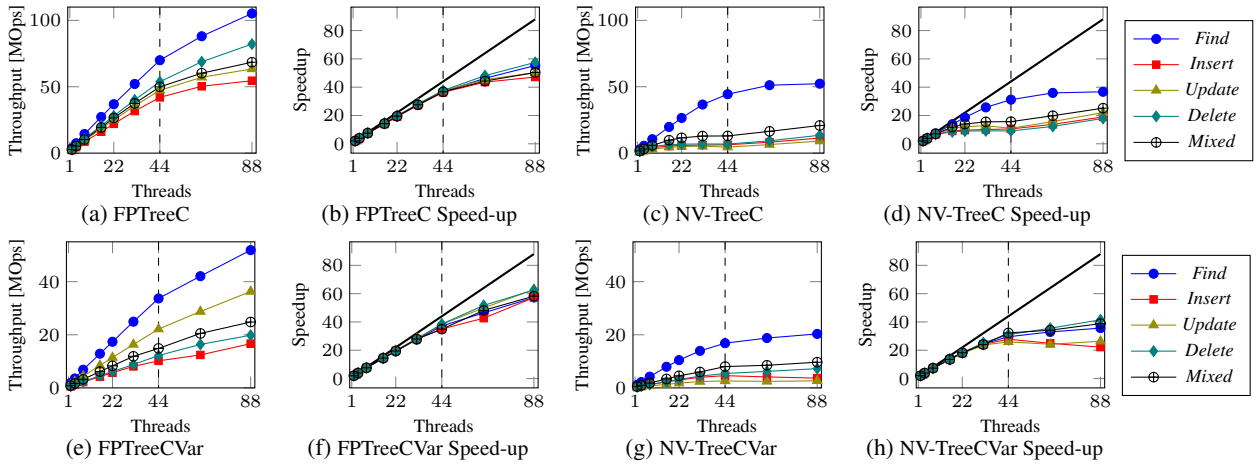


Figure 10: Concurrency performance on two sockets – 50M key-values followed by 50M operations.

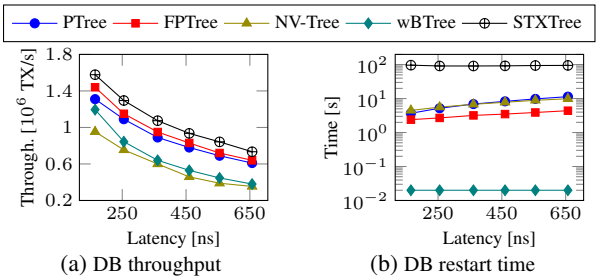
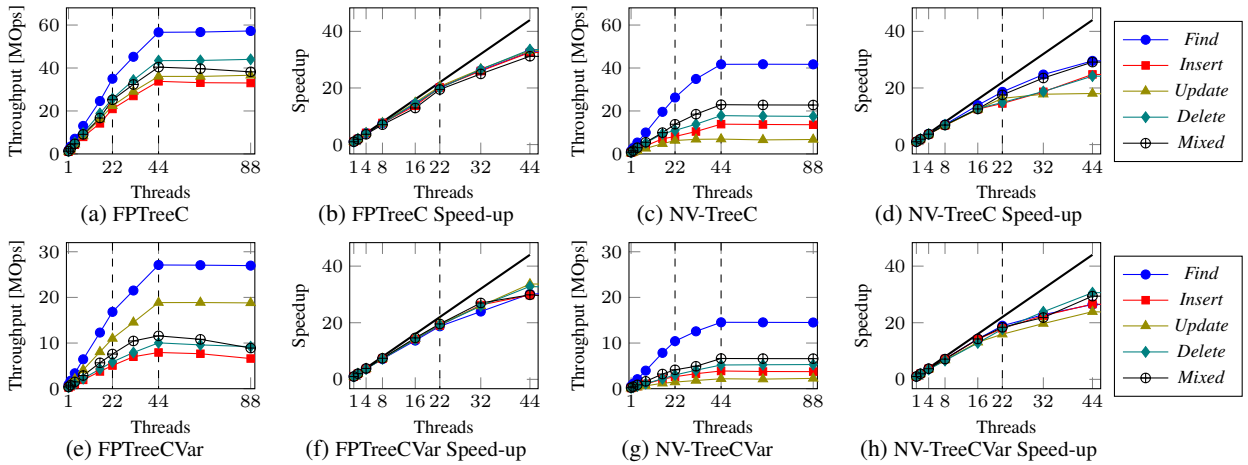


Figure 12: Trees impact on database throughput and restart performance – TATP with 50M subscribers, DB size ~45GB.

insertion workload, a situation that the NV-Tree was unable to handle. Indeed, highly skewed insertions happen most often in the same leaf node. A last-level inner node will then quickly become full and trigger a costly rebuild of the inner nodes. The workload-adaptivity scheme of the NV-Tree tries to space the rebuild phases by dynamically increasing the size of write-intensive nodes at split time, which defers but does not solve the issue of frequent rebuilds in presence of sorted insertions. Eventually, the NV-Tree will have one parent node per leaf node which provokes a memory overflow in our system. To avoid this issue, we set the size of leaf nodes to 1024 and that of inner nodes to 8. The large leaf nodes aim at decreasing the number of inner node rebuilds, while the small size of inner nodes aims at keeping the memory footprint of inner nodes small, even in the case of having one parent node per leaf node.

The results are depicted in Figure 12a. We observe that compared with using the fully transient STXTree, the FPTree incurs an overhead of only 8.74%/12.80% for an SCM latency of 160 ns/650 ns, while the overheads incurred by the PTree, NV-Tree, and wBTree are 16.98%/16.89%, 39.61%/51.77%, and 24.27%/48.23%, respectively. On the one side, the limited slowdown caused by the FPTree and the PTree shows the significant benefit of selective persistence from which stems a decreased dependency with respect to the latency of SCM. On the other side, the FPTree outperforming the PTree shows the benefits of the fingerprints. The decrease in throughput with higher SCM latencies is due to other database data structures being placed in SCM. One can note that because of its larger leaf nodes, the NV-Tree performs worse than the wBTree.

To measure the impact on database recovery, we simulate a crash and monitor the restart time. We use the same benchmark settings as in the previous experiment. Recovery consists in checking the sanity of SCM-based data and rebuilding DRAM-based data. The recovery process is parallelized and uses 8 cores. Figure 12b shows restart time results. Since the wBTree resides fully in SCM, recovery is near-instantaneous –in the order of tens of milliseconds. However, using the wBTree incurs a severe overhead on query performance, as shown in Figure 12a. We observe that the FPTree, PTree, and NV-Tree allow respectively 40.17x/21.56x, 26.05x/8.32x, and 21.42x%/9.58% faster



(a) FPTreeC (b) FPTreeC Speed-up (c) NV-TreeC (d) NV-TreeC Speed-up

(e) FPTreeCVar (f) FPTreeCVar Speed-up (g) NV-TreeCVar (h) NV-TreeCVar Speed-up

Figure 11: Concurrency performance on one socket with an SCM latency of 145 ns – 50M key-values warmup followed by 50M operations.

restart times for an SCM latency of 160 ns/650 ns than using the STXTree. One can note that the NV-Tree performs on par with the PTree thanks to its larger leaves which allow for more data locality while scanning the leaf nodes during recovery. Still, the FPTree outperforms the NV-Tree thanks to rebuilding compact inner nodes, and outperforms the PTree thanks to using leaf groups that provide better data locality when traversing the leaves during recovery.

Memcached experiments

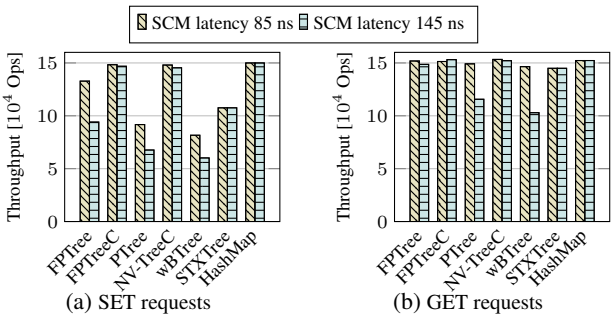
Memcached[5] is a key-value cache that uses a main-memory hash table internally. It employs a locking mechanism on two levels: the first is global locks on the LRU lists of items, and the second is locks on the buckets of the hash table. We replace the hash table by the variable-size keys versions of the evaluated trees. Besides, we remove the bucket locking mechanism and replace it by global locks for non-concurrent trees and rely on the concurrency scheme of the concurrent ones. Another necessary change was to insert the full string key in the trees instead of its hash value to avoid collisions.

To measure the impact on performance, we run the *memcached* server on the HTM system, and the *mc-benchmark*[6] with 50 clients on the emulation system. The measured network bandwidth between the two machines is 940 Mbits/s. We found that using 2 workers in *memcached* yielded the best results for the single-threaded trees while 4 workers was the optimal for the concurrent ones. The *mc-benchmark* executes 50M SET requests followed by 50M GET requests. In addition, we use the same method as in the concurrency experiments to emulate a higher latency, that is, we bind *memcached* to one socket and use the memory of the other socket for SCM.

Figure 13 summarizes the experimental results. We observe that the NV-Tree and the concurrent version of the FPTree (denoted *FPTreeC* in the figure) perform nearly equally to vanilla *memcached* with the hash map table, for both latencies (85 ns and 145 ns) because their concurrent nature allows them to service requests in parallel and saturate the network. Indeed, the incurred overheads are less than 2% for the concurrent FPTree, and less than 3% for the NV-Tree. The single-threaded trees however incur significant overheads. Indeed, for SET/GET requests, the overheads incurred by the single-threaded FPTree, PTree, and wBTree are respectively 11.45%/0.29%, 40.37%/3.40%, and 45.57%/3.81% for an SCM latency of 85 ns. These overheads surge to respectively 37.28%/2.39%, 54.90%/24.06%, and 59.88%/32.40% with a higher SCM latency of

[5] http://memcached.org/

[6] https://github.com/antirez/mc-benchmark



(a) SET requests (b) GET requests

Figure 13: Trees impact on *memcached* performance for different latencies – *mc-benchkark* with 50M operations.

145 ns. As for the STXTree that resides fully in DRAM, it incurs an overhead of 28.29%/4.83% for SET/GET requests, respectively.

In conclusion, in both micro-benchmarks and end-to-end scenarios, the FPTree performs nearly equally to transient counterparts, significantly outperforms state-of-the-art persistent trees, and scales well in highly-concurrent situations.

7. CONCLUSION

In this paper, we proposed the *FPTree*, a novel hybrid SCM-DRAM persistent, concurrent B$^+$-Tree that supports both fixed-size and variable-size keys. Pivotal to the design of the FPTree are the following key principles: (1) fingerprinting; (2) selective persistence; (3) selective concurrency; and (4) state-of-the-art unsorted leaves.

These design principles make it possible to achieve the goals we put forward for our work: (1) the FPTree is persistent and guarantees any-point crash recovery to a consistent state without loss of information; (2) the FPTree exhibits fast recovery compared to a full rebuild; (3) the performance of the FPTree is similar to transient data structures, is resilient to high SCM latencies, and scales well in highly concurrent scenarios.

We perform an extensive experimental evaluation which shows that our FPTree exhibits significantly better base operation performance compared with state-of-the-art persistent trees, while using DRAM for less than 3% of its total size. Moreover, the FPTree recovery performance is almost one order of magnitude faster than a full rebuild. We conduct an end-to-end evaluation using *memcached* and a prototype database. We show that compared with a fully transient tree, the overhead of using the FPTree is limited to less than 2% and 13% for *memcached* and the prototype database, respectively, while using state-of-the-art persistent trees incur overheads of up to 60% and 52%, respectively.

Acknowledgements

We thank the anonymous reviewers for their constructive comments that helped us improve the paper. We also thank Qingsong Wei for his help in figuring out the implementation details of the NV-Tree. Special thanks go to Ingo Müller for the fruitful discussions that helped refine the ideas of the paper, and to Roman Dementiev for his valuable help with TSX. This work is partially supported by the German Research Foundation (DFG) within the Collaborative Research Center "SFB 912/HAEC" as well as the Cluster of Excellence "Center for Advancing Electronics Dresden (cfaed)".

8. REFERENCES

[1] SR. Dulloor. *Systems and Applications for Persistent Memory.* PhD Thesis, 2016. https://smartech.gatech.edu/bitstream/handle/1853/54396/DULLOOR-DISSERTATION-2015.pdf.

[2] Intel®Architecture Instruction Set Extensions Programming Reference. Technical report, 2015. http://software.intel.com/en-us/intel-isa-extensions.

[3] SNIA NVM Programming Model V1.1. Technical report, 2015. http://www.snia.org/sites/default/files/NVMProgrammingModel_v1.1.pdf.

[4] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi. Spin-transfer torque magnetic random access memory (stt-mram). *ACM J. Emerg. Technol. Comput. Syst.*, 9(2), 2013.

[5] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *ACM SIGMOD*, 2015.

[6] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla, et al. Phase change memory technology. *Journal of Vacuum Science & Technology B*, 28(2), 2010.

[7] A. Chatzistergiou, M. Cintra, and S. D. Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(5), 2015.

[8] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, 2011.

[9] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *PVLDB*, 8(7), 2015.

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGPLAN Not.*, 47(4), 2011.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *ACM SOSP*, 2009.

[12] S. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, R. Sankaran, J. Jackson, and D. Subbareddy. System software for persistent memory. In *EuroSys*, 2014.

[13] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *IEEE ICDE*, 2011.

[14] G. Graefe. A survey of b-tree locking techniques. *ACM Transactions on Database Systems (TODS)*, 35(3):16, 2010.

[15] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[16] G. Graefe and H. Kimura. Orthogonal key-value locking. In *BTW*, 2015.

[17] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving in-memory database index performance with intel® transactional synchronization extensions. In *IEEE HPCA*, 2014.

[18] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *ACM SIGMOD*, 2015.

[19] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *IEEE ICDE*, 2013.

[20] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS XVII*, 2012.

[21] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main-memory databases. In *CIDR*, 2015.

[22] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage Management in the NVRAM Era. *PVLDB*, 7(2), 2013.

[23] J. Rao and K. A. Ross. Making B+- Trees cache conscious in main memory. In *ACM SIGMOD*.

[24] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *USENIX FAST*, 2011.

[25] S. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5), 2014.

[26] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *SIGPLAN Not., 47(4)*, 2011.

[27] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10), 2014.

[28] J. Yang, Q. Wei, C. Wang, C. Chen, K. Yong, and B. He. Nv-tree: A consistent and workload-adaptive tree structure for non-volatile memory. *IEEE Transactions on Computers*, PP(99), 2015.

[29] J. J. Yang and R. S. Williams. Memristive devices in computing system: Promises and challenges. *ACM J. Emerg. Technol. Comput. Syst.*, 9(2), 2013.

[30] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *IEEE/ACM MICRO-46*, 2013.

APPENDIX

A. PAYLOAD SIZE IMPACT

In this appendix we study the effect of the payload (value) size on the performance of the evaluated trees. We follow the same experimental setup as in Section 6 while varying the payload size from 8 bytes to 112 bytes. Figures 14b to 14d depict single-threaded performance results with an SCM latency set to 360 ns, while Figures 14e to 14f depict multi-threaded performance on a single socket with 44 threads. We observe two patterns: (1) The NV-Tree stands out as the tree that is most affected by larger payloads, which is explained by the fact that full linear scan of leaves needs to read larger amounts of data; (2) Insert operations tend to suffer more from larger payloads, which stems from the need for larger, more expensive SCM allocations. In general however, the performance of the FPTree and wBTree vary only slightly with larger payloads, thanks to them having respectively constant and logarithmic average leaf scan costs.

B. LEAF GROUPS MANAGEMENT

In this appendix we discuss how insert and delete operations make use of leaf groups in the single-threaded version of the FPTree.

Insert

The insert operation using leaf groups differs from the concurrent version of the FPTree in the split operation. Indeed the non-concurrent version of the FPTree employs leaf groups and does not allocate memory on every split. The same recovery logic as for insert operations without leaf groups applies and only the leaf allocation step is replaced by a call to the *GetLeaf* function (described in Algorithm 10). This function requires a micro-log that contains a single persistent pointer, denoted *PNewGroup*. Instead of directly allocating a new leaf, *GetLeaf* first checks whether there is a leaf that is available in the queue of free leaves, and if not, it will allocate a leaf group, add it to the linked list of leaf groups, and insert its members into the transient queue of free leaves, except the one that is returned. The corresponding recovery function is shown in Algorithm 11: if *PNewGroup* is not null, this means that the allocation took place and we can continue the operation as indicated in the pseudo-code. Otherwise, no action is required since the queue of free leaves is transient and rebuilt at recovery time.

Algorithm 10 GetLeaf(PLeafNode PLeaf)

```
 1: μLog = Tree.GetLeafLog;
 2: if Tree.FreeLeavesQueue.IsEmpty() then
 3:     allocate(μLog.PNewGroup, sizeof(LeafNode)*GROUP_SIZE);
 4:     Tree.PGroupsListTail.Next = μLog.PNewGroup;
 5:     Persist(Tree.PGroupsListTail.Next);
 6:     Tree.PGroupsListTail = μLog.PNewGroup;
 7:     Persist(Tree.PGroupsListTail);
 8:     μLog.Reset();
 9:     insert Tree.PGroupsListTail in FreeLeavesQueue;
10: return Tree.FreeLeavesVector.Pop();
```

Algorithm 11 ReoverGetLeaf(GetLeafLog μLog)

```
1: if μLog.PNewGroup != NULL then
2:     if Tree.PGroupsListTail == μLog.PNewGroup then
3:         /* Everything was done except resetting the micro-log */
4:         μLog.Reset();
5:     else
6:         Continue from line GetLeaf:4;
```

Delete

The only difference between the Delete operation of the concurrent FPTree and the single-threaded FPTree is the leaf delete procedure. Indeed, the FPTree uses leaf groups and does not require a deallocation on every leaf deletion. Algorithm 12 shows the pseudo-code of the *FreeLeaf* function that replaces the deallocation function. This operation requires a micro-log that contains two persistent pointers, denoted *PCurrentGroup* and *PPrevGroup*. First, the procedure checks whether the leaf group to which the deleted leaf belongs is completely free. If it is, it will deallocate the leaf group and update the head of the linked list of groups if needed. Otherwise, it will push the deleted leaf into the transient queue of free leaves. The corresponding recovery function is depicted in Algorithm 13 and follows the same logic as the leaf delete recovery function that is detailed in Section 5: if *PCurrentGroup* is not null, this means that the micro-log was set but the deallocation did not take place. We can continue the operation starting by either updating the head of the linked list of groups, or by deallocating the group, as indicated in Algorithm 13. If *PCurrentGroup* is null, no action is needed since the queue of free leaves is transient and rebuilt at restart time.

Algorithm 12 FreeLeaf(LeafNode Leaf)

```
 1: get head of the linked list of Groups PHead;
 2: μLog = Tree.FreeLeafLog;
 3: (Group,PPrevGroup) = getLeafGroup(Leaf);
 4: if Group.IsFree() then
 5:     delete Group from FreeLeavesQueue;
 6:     set μLog.PCurrentGroup to persistent address of Group;
 7:     Persist(μLog.PCurrentGroup);
 8:     if μLog.PCurrentGroup == PHead then
 9:         /* Group is the head of the linked list of Groups */
10:         PHead = Group.Next;
11:         Persist(PHead);
12:     else
13:         μLog.PPrevGroup = PPrevGroup;
14:         Persist(μLog.PPrevGroup);
15:         PrevGroup.Next = Group.Next;
16:         Persist(PrevGroup.Next);
17:     /* the deallocate function resets PCurrentGroup */
18:     deallocate(μLog.PCurrentGroup);
19:     reset μLog;
20: else
21:     push Leaf in FreeLeavesVector;
```

Algorithm 13 RecoverFreeLeaf(FreeLeafLog μLog)

```
 1: get head of linked list of groups PHead;
 2: if μLog.PCurrentGroup != NULL and μLog.PPrevGroup != NULL then
 3:     /* Crashed between lines FreeLeaf:15-18 */
 4:     Continue from FreeLeaf:15;
 5: else
 6:     if μLog.PCurrentGroup != NULL and μLog.PCurrentGroup ==
        PHead then
 7:         /* Crashed at line FreeLeaf:10 */
 8:         Continue from FreeLeaf:10;
 9:     else
10:         if μLog.PCurrentGroup != NULL and
            μLog.PCurrentGroup→Next == PHead then
11:             /* Crashed at line FreeLeaf:14 */
12:             Continue from FreeLeaf:18;
13:         else
14:             reset μLog;
```

We note that to speed up *GetPrevGroup*, it can be implemented using a transient index on top of the linked list of free leaves.

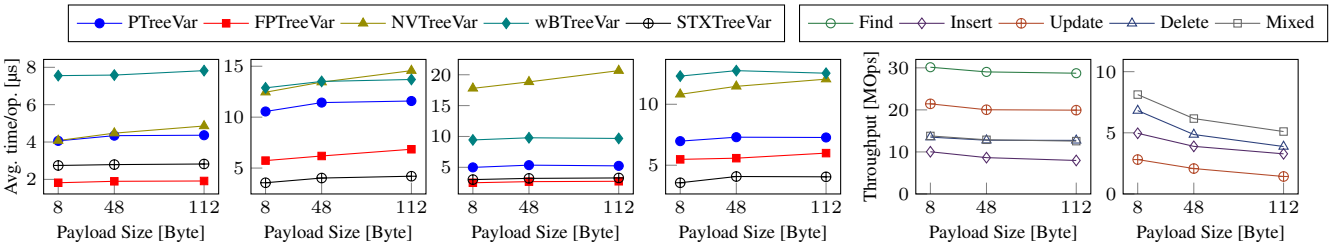


Figure 14: Impact of payload size on the single-threaded and multi-threaded performance of the evaluated trees.

Algorithm 14 ConcurrentInsert_String(Key K, Value V)

```
1:  Decision = Result::Abort;
2:  while Decision == Result::Abort do
3:      speculative_lock.acquire();
4:      (Leaf, Parent) = FindLeaf(K);
5:      if Leaf.lock == 1 then
6:          Decision = Result::Abort; Continue;
7:      Leaf.lock = 1;
8:      Decision = Leaf.isFull() ? Result::Split : Result::Insert;
9:      speculative_lock.release();
10: if Decision == Result::Split then
11:     splitKey = SplitLeaf(Leaf);
12: slot = Leaf.Bitmap.FindFirstZero();
13: allocate(Leaf.KV[slot].PKey, strlen(K));
14: set NewKey to key pointed to by Leaf.KV[slot].PKey;
15: NewKey = K; Persist(NewKey);
16: Leaf.KV[slot].Val = V; Leaf.Fingerprints[slot] = hash(K);
17: Persist(Leaf.KV[slot].Val); Persist(Leaf.Fingerprints[slot]);
18: Leaf.Bitmap[slot] = 1; Persist(Leaf.Bitmap);
19: if Decision == Result::Split then
20:     speculative_lock.acquire();
21:     UpdateParents(splitKey, Parent, Leaf);
22:     speculative_lock.release();
23: Leaf.lock = 0;
```

Recovery

The recovery function of the non-concurrent FPTree is similar to that of its concurrent version, except that instead of iterating over the arrays of micro-logs, the non-concurrent version contains only one leaf split micro-log and one leaf delete micro-log. In addition, they differ in how the inner nodes are rebuilt. Indeed, the single-threaded FPTree traverses the linked list of leaf groups instead of traversing the linked list of leaves, which allows for more data locality. Besides, the keys that are retrieved from the leaves are unsorted and require to be sorted before proceeding to rebuilding inner nodes. Nevertheless, the benefits of increased data locality outperformed the cost of the additional sort, as shown in Section 6.2. We note that the transient vector of free leaves is rebuilt while traversing the list of leaf groups.

C. VARIABLE-SIZE KEY OPERATIONS

In this appendix we discuss the insert and delete operations of the concurrent FPTree with variable-size keys (string keys).

Insert

Algorithm 14 shows the pseudo code of the *concurrent Insert* operation. It is similar to its fixed-size keys counterpart. In particular, the leaf split operation is identical to that of fixed-size keys. The only difference is that for variable-size keys, we need to allocate persistent memory to store the insertion key. To ensure consistency, we do not need any additional micro-logging compared to fixed-size keys. When the insertion position is found, the new key is allocated by passing the key persistent pointer of the insertion position to the allocator that persistently writes the persistent address of the allocated memory to the provided persistent pointer. Then, the key to insert is persistently copied into the newly allocated key. Thereafter, the value and the fingerprint are persistently written in any order as they remain invisible until the bitmap is updated. Finally, the bitmap is persistently updated to make changes visible. If a crash occurs after the new key is allocated but before the bitmap is updated, the newly allocated key is a persistent memory leak. To detect this problem during recovery, it is sufficient to add the following additional check while traversing the leaves to rebuild inner nodes: for every unset bit in the bitmap of leaf, check whether its corresponding key persistent pointer is null (as it should be). If it is not, then we might have crashed during an insert operation, after having allocated the new key but before having updated the bitmap to reflect the insertion. Deallocating the key, as shown in the recovery procedure (Algorithm 17), is sufficient to set the tree back to a state of leak-free consistency.

Delete

Algorithm 15 shows the pseudo code of the *concurrent Delete* operation. It is similar to its fixed-size key counterpart. In particular, the leaf delete operation is identical to that of fixed-size keys. The only difference is that in this case, we need to deallocate the key that is deleted. Similarly to insert operations, there is no additional micro-log required to ensure crash-safety. Once the position of the key to be deleted is determined, the bitmap is updated and persisted to reflect the deletion. Finally, the deleted key is deallocated. A crash after updating the bitmap but before deallocating the key will lead to a persistent memory leak. Indeed, the key will be invisible since the bitmap has been updated and thus the key will never get deallocated. During recovery however, it is sufficient to conduct the same additional check as for insert operations: for every unset bit in the bitmap of leaf, check whether its corresponding key persistent pointer is null (as it should be). If it is not, then we might have crashed during a delete operation, after having updated the bitmap but before having deallocation the deleted key. All there is to do to recover to a consistent and leak-free state is to deallocate the keys pointed to by these non-null persistent pointers, as shown in Algorithm 17.

Update

Algorithm 16 shows the pseudo code of the *concurrent Update* operation. Similarly to the fixed-size key case, this operation is an optimized insert-after-delete operation where both the insertion and the deletion are made p-atomically visible by updating the bitmap. Compared to an insert operation, an update operation does not allocate memory, except when a split is needed. Indeed, instead of allocating a new key, the persistent pointer of the key is copied into a new location. After updating the bitmap, the persistent pointer of the old record location must be reset in order to ensure that a reference to a key exists only once. In the case of a crash after reflecting the update but before resetting the persistent pointer of the old record location, the recovery function might be mislead and

Algorithm 17 RebuildInnerNodes_String()

```
1: let MaxVector be a vector of keys;
2: for for Leaf_i in linked-list of leaves do
3:     let MaxKey be a key set to the smallest value of its domain;
4:     for each slot in Leaf_i.Bitmap do
5:         if Leaf_i.Bitmap[slot] == 1 then
6:             /* Find max key */
7:             set K to key pointed to by Leaf_i.KV[slot].PKey;
8:             MaxKey = MAX(K, MaxKey);
9:         else
10:            /* Test whether there is a memory leak */
11:            if Leaf_i.KV[slot].PKey != NULL then
12:                if KeyExists(Leaf_i, Leaf_i.KV[slot].PKey) then
13:                    Leaf_i.KV[slot].PKey.reset();
14:                else
15:                    deallocate(Leaf_i.KV[slot].PKey);
16:        MaxVector[i] = MaxKey;
17: rebuild inner nodes using MaxVector;
```

Algorithm 15 ConcurrentDelete_String(Key K)

```
1: Decision = Result::Abort;
2: while Decision == Result::Abort do
3:     speculative_lock.acquire();
4:     /* PrevLeaf is locked only if Decision == LeafEmpty */
5:     (Leaf, PPrevLeaf) = FindLeafAndPrevLeaf(K);
6:     if Leaf.lock == 1 then
7:         Decision = Result::Abort; Continue;
8:     if Leaf.Bitmap.count() == 1 then
9:         if PPrevLeaf->lock == 1 then
10:            Decision = Result::Abort; Continue;
11:        Leaf.lock = 1; PPrevLeaf->lock = 1;
12:        Decision = Result::LeafEmpty;
13:    else
14:        Leaf.lock = 1; Decision = Result::Delete;
15:    speculative_lock.release();
16: slot = Leaf.FindInLeaf(K);
17: Leaf.Bitmap[slot] = 0; Persist(Leaf.Bitmap);
18: deallocate(Leaf.KV[slot].PKey);
19: if Decision == Result::LeafEmpty then
20:    DeleteLeaf(Leaf, PPrevLeaf);
21:    PrevLeaf.lock = 0;
22: else
23:    Leaf.lock = 0;
```

might treat this as a crash during an insert or a delete operation, and deallocate the key, as described previously. To avoid this issue, whenever a deleted record with a non-null key persistent pointer is found, the recovery function checks whether there is a valid record in the same leaf that points to the same key. If there is one, the key persistent pointer simply needs to be reset. Otherwise, this means that the crash occurred during an insert or a delete operation and the key needs to be deallocated.

Algorithm 16 ConcurrentUpdate_String(Key K, Value V)

```
1: Decision = Result::Abort;
2: while Decision == Result::Abort do
3:     speculative_lock.acquire();
4:     (Decision, prevPos, Leaf, Parent) = FindKeyAndLockLeaf(K);
5:     speculative_lock.release();
6: if Decision == Result::Split then
7:     splitKey = SplitLeaf(Leaf);
8: slot = Leaf.Bitmap.FindFirstZero();
9: Leaf.KV[slot].PKey = Leaf.KV[prevSlot].PKey;
10: Leaf.KV[slot].Val = V;
11: Leaf.Fingerprints[slot] = Leaf.Fingerprints[prevSlot];
12: Persist(Leaf.KV[slot]); Persist(Leaf.Fingerprints[slot]);
13: copy Leaf.Bitmap in tmpBitmap;
14: tmpBitmap[prevSlot] = 0; tmpBitmap[slot] = 1;
15: Leaf.Bitmap = tmpBitmap; Persist(Leaf.Bitmap);
16: Leaf.KV[prevSlot].PKey.reset();
17: if Decision == Result::Split then
18:     speculative_lock.acquire();
19:     UpdateParents(splitKey, Parent, Leaf);
20:     speculative_lock.release();
21: Leaf.lock = 0;
```

Recovery

The recovery procedure of the FPTree with variable-size keys is identical to that with fixed-size keys, except for rebuilding inner nodes where the additional checks described above to catch memory leaks are added. Algorithm 17 shows the pseudo-code of the inner node rebuilding function.