Failure-Atomic Slotted Paging for Persistent Memory

Jihye Seo Wook-Hee Kim Woongki Baek Beomseok Nam * Sam H. Noh

UNIST (Ulsan National Institute of Science and Technology)

{jihye,okie90,wbaek,bsnam,samhnoh}@unist.ac.kr

Abstract

The slotted-page structure is a database page format commonly used for managing variable-length records. In this work, we develop a novel *"failure-atomic slotted-page structure"* for persistent memory that leverages byte-addressability and durability of persistent memory to minimize redundant write operations used to maintain consistency in traditional database systems. Failure-atomic slotted-paging consists of two key elements: (i) *in-place commit* per page using hardware transactional memory and (ii) *slot-header logging* that logs the commit mark of each page. The proposed scheme is implemented in SQLite and compared against NVWAL, the current state-of-the-art scheme. Our performance study shows that our failure-atomic slotted-paging shows *optimal* performance for database transactions that insert a single record. For transactions that touch more than one database page, our proposed slot-header logging scheme minimizes the logging overhead by avoiding duplicating pages and logging only the metadata of the dirty pages. Overall, we find that our failure-atomic slotted-page management scheme reduces database logging overhead to 1/6 and improves query response time by up to 33% compared to NVWAL.

Categories and Subject Descriptors H.2.7 [*Database Management*]:Logging and recovery

Keywords Non-volatile memory; Database buffer caching; Slotted page structure; Database recovery

1. Introduction

Emerging persistent memory (PM) such as phase-change memory and spin-transfer torque MRAM (STT-MRAM) is being considered as fast byte-addressable secondary storage. However, recent studies have shown that the legacy block I/O interface makes it difficult to take full advantage of the beneficial features of these byte-addressable PM [7, 10, 15, 18, 26].

PM has also been considered as (non-volatile) main memory (NVMM) [18, 44]. In such cases, durability can be guaranteed without employing journaling or logging methods as in secondary storage. However, when used as main memory, more sophisticated memory management is required for consistency and atomicity because memory write operations can be arbitrarily reordered in today's processors. Moreover, in order to guarantee store instructions flush data all the way down to NVMM, cache line flush operations must be carefully used. Since cache line flush operations are expensive, replacing DRAM with PM for general purpose applications still needs a significant amount of research effort.

Database transaction support is a promising use case of PM because database transactions require persistency, byte-addressability, and high performance. In traditional DBMS, journaling and logging techniques perform redundant write operations because write operations are not atomic and they are performed in volatile memory first and then subsequently in persistent secondary storage. When the DBMS commits a database page updated in volatile buffer cache, it first saves a journal page, that is, a back-up copy of the consistent page in secondary storage for recovery purposes. Essentially, the DBMS layer relies on redundant write operations for consistency.

Similarly, the amount of I/O is doubled again at the file systems layer because file systems also rely on journaling (or similar mechanisms) to protect the file system metadata. This *journaling of journal* anomaly is known to aggravate the notorious write amplification problem in flash memory [13]. In order to resolve the journaling of journal anomaly, several previous studies such as version-based recovery and write-ahead logging on PM have been considered [15, 16, 33]. These schemes have been shown to be effective in improving database transaction throughput by leveraging the property of PM.

In this work, we propose to use PM as a database buffer cache since PM is expected to deliver the high performance of DRAM. For the persistent database buffer cache, we present a novel *database slotted-page manage-*

* corresponding author

ASPLOS '17 April 8–12, 2017, Xi'an, China.

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

ment scheme that minimizes write operations by eliminating the necessity of logging, copy-on-write, and volatile copies of database pages while preserving atomicity and consistency. The *slotted-page structure* is a database page format commonly used for variable-length records [30, 34]. Numerous database systems including SQLite, PostgreSQL, and InnoDB are known to use database tables that make use of the slotted-page structure [4]. Two key ingredients of our persistent slotted-page management scheme are (i) *in-place commit* that eliminates redundant copies and (ii) *slot-header logging* that minimizes the logging overhead. We implement our novel database page management scheme in SQLite. Through extensive experiments using *Quartz* [38], a software-based PM latency emulator, we show that our slotted-page management scheme reduces the database transaction commit overhead to 1/6 compared to NVWAL [15], the current state-of-the-art management scheme.

In order to minimize the write operations, our scheme avoids copy-on-write or journaling that cause redundant write operations as much as possible. The scheme that we propose, which we refer to as *in-place commit*, directly updates the database pages in PM without creating an extra copy. We show that this can be done with transactional memory support without sacrificing consistency. Our in-place commit scheme provides optimal write performance for transactions that write a single record, which comprises the majority of database transactions in mobile applications.

For transactions that update multiple pages, we fall back to a logging method similar to legacy methods. However, the *slot-header logging scheme* that we propose does not duplicate an entire page but only logs the metadata of the page.

Next generation persistent memory devices are generally expected to guarantee failure-atomic writes in 8-byte word units [7, 24, 27, 39]. Several previous studies have also been conducted under the assumption that failure-atomic write instructions for such devices will be available at a higher granularity [8, 23]. In this work, we employ the failure-atomic cache line write operation that uses hardware transactional memory as done by Dulloor et al. [8] so that multiple slot offsets in a slot header can be simultaneously updated. We also evaluate our logging approach that can be used to consistently update slot headers when the atomic write granularity for PM is smaller than the cache line size.

The main contributions of this work can be summarized as follows.

- **Failure-atomic in-place commit scheme**
  We develop a failure-atomic in-place commit scheme for slotted-page structures in PM using hardware transactional memory that eliminates redundant write operations. We show that its performance is *optimal* in terms of the I/O performance because it removes the necessity of redundant writes for database transactions that insert a single record.
- **Failure-atomic slot-header logging scheme**
  We develop a Failure-Atomic Slot-Header logging (FASH) scheme for slotted-page structures in PM. For transactions that require manipulating more than a single record, logging becomes inevitable. FASH minimizes the logging overhead by avoiding page duplication and logging only the metadata of the dirty pages.
- **Performance analysis of PM-only database buffer caching**
  We implement FASH and the Failure-Atomic Slot-header logging with in-place commiT (FAST) scheme in the SQLite database buffer cache manager. We perform an exhaustive performance study and observe that making use of PM-only buffer caching with FAST significantly reduces unnecessary memory operations and reduces the database logging overhead to 1/6 and improves query response time by up to 33% compared to NVWAL, which employs hybrid PM+DRAM memory systems.

The rest of the paper is organized as follows: In Section 2, we present the background and other related efforts. In Section 3, we present our design and implementation of failure-atomic slotted paging with in-place commit and slot-header logging. In Section 4, we present the design and implementation of failure-atomic persistent slotted-paging in SQLite B-tree as a case study. Section 5 provides performance results and analysis. In Section 6, we conclude the paper.

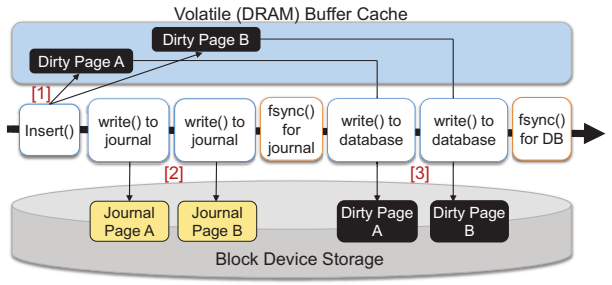
2. Background and Related Work

In this section, we first review background work that is needed to understand the contributions of our study. Specifically, we review how traditional database systems safeguards itself from system crashes. We then present recent state-of-the-art studies that leverage persistent byte-addressable persistent memory (PM) to improve database transaction performance while guaranteeing database consistency and durability.

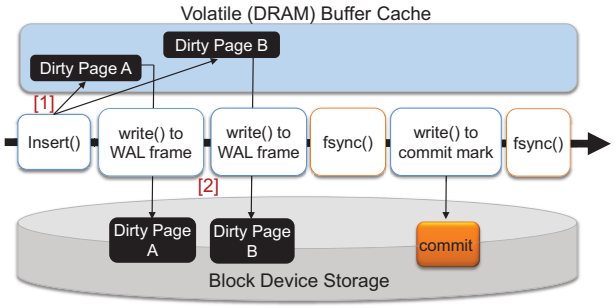
2.1 Recovery in Traditional Database Systems

If a system crashes while a transaction is making changes to a database file, the database file needs to be restored using the most recent consistent backup. For this purpose, traditional database management systems rely on multi-versioning or temporary backup files such as rollback journals and write-ahead logs [15, 16, 20]. In the following, we describe the workings of the latter in more detail as they are relevant to this study.

A rollback journal is a temporary backup file that saves the prior state of each modified page. Figure 1a shows the overall workings of the journaling mechanism as is done in SQLite [13, 18]. In the figure (as with Figure 1b), the top box is the volatile (DRAM) buffer cache, the middle sequence



(a) Journaling



(b) Write-Ahead Logging

Figure 1: Recovery using Temporary Backup Files in Traditional Database Systems

of boxes are the operations involved in journaling, and the circle in the bottom is the persistent store, that is, the disk. If a transaction attempts to modify data pages, the DBMS copies the data pages from persistent storage to the volatile buffer cache and updates the volatile copy (denoted [1] in Figure 1a). When the transaction is done and commits, the DBMS creates a copy of the untouched original page into the journal in persistent storage ([2]). After the journal file is synchronized, the DBMS overwrites the original pages in the persistent database file with the dirty pages in the volatile buffer cache ([3]). After all the dirty pages are flushed to the database file, the journal file can be deleted or truncated.

This journaling process creates a single copy in volatile memory and writes two copies in persistent storage - one for the journal and one for writing the dirty page to persistent store. Hence, it doubles the amount of I/O in the database layer. It has also been shown that the EXT4 file system layer amplifies the amount of I/O due to file system journaling [13, 16].

Write-ahead logging (WAL), depicted in Figure 1b, is another way of providing atomicity and durability. As with journaling, the DBMS copies the data pages to volatile buffer cache and updates the volatile copy (denoted [1] in Figure 1b). When the transaction commits, all dirty pages in the volatile buffer cache are first written to a log file as undo and redo information ([2]). The dirty pages in the log file are periodically moved to the database file via checkpointing process. Hence, most of the transactions make changes only to the log file. Unlike journaling that creates three copies in total, WAL creates a single volatile copy in DRAM and writes one updated page in persistent storage, although eventually the checkpointing process incurs writes to the original copy. Still, this helps mitigate the journaling of journal problem [13].

2.2 Non-Volatile Memory for Consistency

Next generation persistent memory seems to be in the horizon. Anticipating its realization, numerous studies have been conducted on ways of exploiting its beneficial features [21, 22, 44]. In particular, its use has opened up new opportunities and challenges in redesigning file systems and database management systems by leveraging the durability and high performance of persistent memories [9, 12, 15, 17, 18, 31, 32, 35, 37, 42, 43, 45]. In this section, we review some previous work we deem most relevant to our study.

A few studies have proposed methods for safeguarding against failures for particular data structures and components of systems. Venkataraman et al. propose CDDS, a version-based recovery method for PM [37]. Version-based recovery methods, though, have a limitation that it requires an expensive garbage collection process to recover dead versions. Yang et al. propose NV-Tree that makes updates to data structures in append-only manner so that it can roll back to its previous state without creating a journal [43]. FPTree, proposed by Oukid et al., is similar to NV-Tree in that it also stores leaf nodes in PM while keeping internal tree nodes in DRAM [27]. Chen and Jin propose the wB+-tree, which is another improved version of NV-Tree, that takes the same append-only approach [5]. The wB+-tree, however, stores both the leaf and internal tree nodes in PM. Our work is different from NV-Tree and FPTree in that they target systems that have both DRAM and PM. Also, our work is different from wB+-tree as the wB+-tree concentrates on the atomic update of a single tree node, while our study considers the logging overhead for the page splits and multiple insertions in a database transaction. Furthermore, wB+-tree works only for B-trees with fixed-sized records, but the optimization of the persistent slotted-page structure that we propose can be used not only for B+-trees (or any of its variants) but also for other hash-based indexes. Most importantly, we also implement and evaluate persistent buffer caching in SQLite, a full-featured database management system.

Lee et al. propose UBJ that provides the journaling effect in the buffer cache by unioning buffer caching and journaling in PM [18]. UBJ employs copy-on-write and converts a buffer cache block to a journal log by simply changing the status of the block to a frozen state. However, the contribution of UBJ is limited to management of the journal itself, whereas our study is on the comprehensive recovery of database transactions including multiple operations and objects in a full-featured database management system.

PM has been considered as an alternative secondary storage device for write-ahead logs. For SQLite in write-ahead logging mode, for example, application write operations will be concentrated on a small log file. Hence, the performance of write-ahead logging can be improved by managing undo/redo logs in PM, which can be done in two ways; one is to deploy a persistent file system such as PMFS [8] and BPFS [7] on PM and the other is to employ a persistent heap manager such as NV-Heaps [6], NVMalloc [23], and Heapo [11].

Kim et al. [14] take the former approach while NV-Logging [10], SQLite/PPL [25] and NVWAL [15] take the latter approach. SQLite/PPL is similar to our work in that they also implement a persistent database transaction logging technique in SQLite [25]. SQLite/PPL captures update operations as in write-ahead-logging and stores the log per each page in the PCM log sector. SQLite/PPL is different from our work in that they perform redundant write operations in the volatile buffer cache and the PCM log sector instead of eliminating redundant copies. NVWAL is the most recent scheme proposed, and as we compare the performance of our proposed work with NVWAL, we describe the workings of NVWAL in more detail in the following.

NVWAL and our work share the same goal of reducing the memory write operations and the granularity of memory management, while, at the same time, preserving atomicity and durability of transactions. This is done through differential logging in NVWAL while our work employs in-place update and metadata-only (slot-header) logging. Although NVWAL significantly reduces the amount of memory writes via differential logging, it creates redundant copies of the dirty portion of data pages as it employs a volatile buffer cache. When a transaction finishes all updates in the volatile buffer cache, it flushes the dirty portion of the page to PM in order to enforce durability. This becomes a critical limitation as PM performance approaches that of DRAM as described below. Our failure-atomic slotted-paging does not duplicate the dirty portion of data pages.

Traditionally, with slow legacy block storage devices, the time spent on updating records in volatile buffer cache was almost negligible compared to slow block device I/O time. With the advent of PM with latency in par with DRAM, the time spent on updating records in the volatile buffer cache and the time spent on copying dirty bytes from volatile buffer cache to PM will become similar. Hence, with PM as fast as DRAM, extra copies from volatile to non-volatile devices can now become significant. Then, it is logical to have transactions perform updates directly on persistent memory. This approach will eliminate redundant memory copies and improve the performance of database management systems. This is the key proposition of this work. The key technical challenge here is to do this while assuring that recovery is possible in case of system crashes. This is not simple as making modifications to existing data by overwriting can result in inconsistent states if crashes occur at inopportune times. This paper proposes an efficient PM-exploiting scheme that targets slotted-paging. The details are described next.

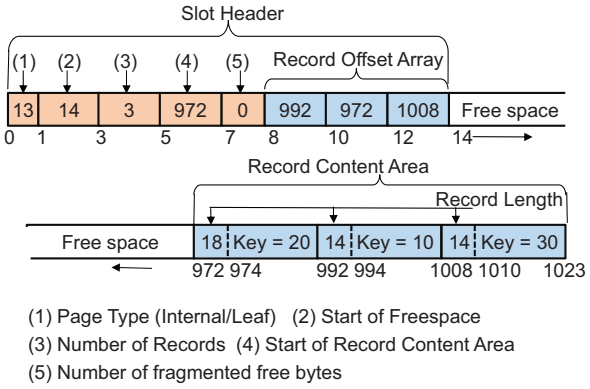


Figure 2: Slotted-Page Structure

3. Failure-Atomic Slotted-Paging

3.1 Slotted-Page Structure

Traditional disk-based database systems store database tables as heap file, hashing file, or B+-tree file formats. In a heap file format, any record can be placed in any page. In a hashing file format, the hash function determines the page in which the record is to be placed. In a B+-tree file format, records are stored in sorted order in B+-tree leaf pages. For variable-length records, these file formats need an efficient internal page representation so that individual records can be easily extracted.

The *slotted-page structure* is commonly used for organizing variable-length records in a fixed-sized block [4, 30, 34]. As shown in Figure 2, in a slotted-page structure, there is a *slot-header* at the beginning of each page, *free space* in the middle, and a *record content area* at the end of the page. The slot-header contains metadata about each page, including the number of records in the page, the end of free space in the page (the beginning of record content area), and an array whose entries contain the location of each record, which we refer to as *record offset array*

When a new record is inserted into a slotted-page, space is allocated for the record at the end of the free space extending the record content area. Note that the record content area also contains the length of the record and that it grows towards the beginning of the page. The offset and size of the new space allocated is added to the record offset array of the slot-header. The record offset array grows towards the end of the page. As the new records are always placed at the head of the record content area regardless of their keys, the record offset array is always kept sorted according to the ordering of the keys in B+-tree file format.

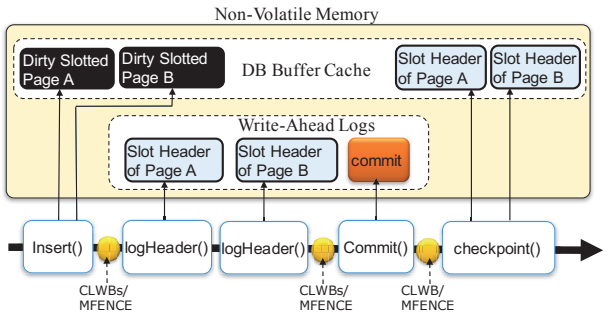


Figure 3: Failure-Atomic Slotted-Paging with Persistent Buffer Cache

3.2 Slotted-Paging and In-Place Commit

The slotted-page structure was originally designed for block device storage. When a transaction makes changes to a slotted-page, it copies the 4K or 8K byte sized page from the block storage device to volatile main memory (the buffer cache), updates the entire page, and flushes the page to the block storage device.

If we replace volatile main memory with PM, a transaction's commit operation does not have to flush updated slotted-pages to slow block storage. Instead, we can simply place a commit mark for any change that has taken place in the slotted-page. Such a scheme eliminates unnecessary redundant writes minimizing I/O and memory operations, which will help improve the performance of transactions and mitigate the endurance problems of PM.

The key idea behind the in-place commit scheme is to ensure that all modifications become committed and viewable with a single failure-atomic write operation. Let us see how this is done with persistent slotted-paging, that is, a slotted-page in persistent memory.

Failure-atomic write instructions are expected to be supported at 8 bytes or a larger granularity for PM [7, 23], and hardware transactional memory is one way of achieving the same goal. The advent of commercially available hardware transactional memory such as the Intel's Restricted Transactional Memory (RTM) and Hardware Lock Elision (HLE) has opened up a new means to support coarse-grained atomic operations via hardware support. Hardware transactional memory has spurred the growth of interest in using hardware-assisted transactional support in database systems [19, 40, 41]. In this work, we employ RTM as we need a user-defined fallback execution path if hardware transactions abort. RTM, in particular, provides three new instructions - `XBEGIN`, `XEND`, and `XABORT` that allow programmers to specify the start and end of a transaction, and to explicitly abort a transaction if the transaction cannot be successfully executed [8, 40]. RTM guarantees that the store operations within a single transaction are not visible outside the transaction until `XEND` is successfully completed. That is, a dirty cache line remains in the *write combining store buffer*, which combines multiple consecutive small 8 bytes writes, without flushing the cache line to the memory subsystem. If the system crashes before the transaction finishes, the dirty cache line will be lost and it does not hurt the consistency of the memory subsystem.

One of the major limitations of RTM is that an RTM transaction cannot successfully commit if its working set (i.e., read and write sets) size exceeds the hardware limit. Since RTM transactions could keep getting aborted due to consecutive transactional overflows, there is no guarantee for forward progress. Even if the working set size of persistent slotted-paging is smaller than the hardware limit, multiple dirty cache lines cannot be flushed to PM atomically. Hence, as in [8], we assume that the underlying hardware supports failure atomicity at cache line granularity, and we restrict the working set size of RTM to be no larger than the cache line size. [1]

In our in-place commit scheme, we make use of hardware transactional memory as follows:

Inserting a record: Suppose a transaction inserts a record into a persistent slotted-page. The actions that are involved with insertion are (i) writing the record along with its length into the record content area and (ii) writing to the slot-header, which involves writing the offset of a new record into the record offset array, updating the number of records, and updating the start of the record content area. Note that writing to the record content area need not be atomic as other transactions cannot see this addition until the slot-header is updated. So, if we can assure that the slot-header is no larger than the cache line size and failure-atomicity is ensured for a cache line write - (ii), then the insertion can be done atomically. [2] This is the premise behind our proposed in-place commit.

Hence, after writing the record, we update the slot-header in the RTM region by inserting the new record offset into its record offset array and increasing the number of records. Since the slot-header in the store buffer can be atomically written to PM by a cache line flush instruction, the slot-header acts as an in-place commit mark ensuring that all actions within the insert transaction has been committed and that the entire slotted-page will always be consistent, durable and fail-safe. Note that if a system crashes while adding a new record into the record content area, the partially written slot-header is, at best, in the RTM region. Thus, the slot-

[1] Nevertheless, best-effort HTM transactions are not guaranteed to succeed. Hence, if an RTM transaction fails, our fallback handler retries the RTM transaction until it succeeds. Alternatively, we can implement a handler that falls back to slot-header logging that we describe in section 3.3 if RTM transactions continuously fail.

[2] We use RTM for atomicity and consistency, but not for durability and isolation. RTM is used just to prevent a partially updated cache line from being written to PM. Durability is guaranteed when we call `clflush` after the RTM transaction ends. It should be noted that `clflush` cannot be called inside the RTM region since it violates the property of hardware transactions.

header in PM is not altered and the partially written data will simply be ignored.

Updating a record: When a record is updated in a persistent slotted-page, the record should not be overwritten for recovery purpose. Hence, we add an updated record in free space and atomically replace the offset of the previous record in the record offset array with the new offset so that the previous record is marked as deleted and the newly added record becomes accessible.

Deleting a record: If a transaction deletes a record, we can atomically invalidate the record by deleting its offset from the record offset array and decreasing the number of records in the slot-header via RTM transaction.

A problem with in-place update and in-place deletion operations to the persistent slotted-page structure is that they can leave holes in the record content area. We discuss how we handle such fragmentation problem in Section 4.3.

These insert/update/delete operations in a persistent slotted-page structure guarantee failure atomicity of a single slotted-page, i.e., a slotted-page is atomically transformed from one consistent state to another consistent state even if a system crashes.

In Android applications, it is known that most write transactions insert just a single data item into the SQLite database as if it is a flat file interface [16]. For such single write transactions, the in-place commit scheme in persistent slotted-page structure is optimal in the sense that it minimizes memory write operations as it does not create any copy page. It requires a minimal number of `store` and `clflush` instructions for the dirty record in the record content area and only one `store` instruction and one `clflush` instruction for the commit mark.

3.3 Slot-Header Logging

The in-place commit scheme and cache line atomic write granularity is, unfortunately, far from satisfactory when a slotted-page structure needs to split or when a transaction updates multiple pages. Splitting a slotted-page must atomically update the slot-headers of two pages - the page that splits and its parent page. Even with RTM, two slot-headers cannot be updated atomically because the underlying PM does not support failure atomicity of writing two separate cache lines. Moreover, database transactions that insert more than one record are not uncommon in enterprise database systems. If they modify multiple pages, the in-place commit scheme alone cannot guarantee failure atomicity of transactions.

In order to provide failure atomicity for a transaction that modifies multiple pages, we have no choice but to fall back to logging methods. However, notice here that now as the database buffer cache is non-volatile, there is no reason to duplicate the dirty bytes in the journal or log because they are already persistent in the buffer cache. In the *slot-header logging* scheme that we propose, we do not duplicate the records in the record content area but do so only for the slot-header, that is, we write only the small slot-header in a separate persistent log, so that we minimize the memory write operations. Since the slot-header in the slotted-page structure behaves as a per-page commit mark, we write the slot-header in a separate log and postpone applying the log to the actual pages until the entire transaction is ready to commit.

Figure 3 illustrates how slot-header logging works. If a transaction modifies page A and page B, we perform in-place updates in the record content area of page A and B, but the slot-headers are not updated in-place. In slot-header logging, the ordering of memory write operations for the two pages does not have to be enforced as long as they are flushed to PM before we write the slot-headers to the log.

After calling cache line flush instructions and memory barrier instructions for the in-place updates in the record content area, we copy and update the slot headers (record offset arrays) of the two pages in a separate log, which we refer to as *slot-header log*. Since we do not overwrite the slot-headers of pages A and B in-place, the updated slot-headers in the slot-header log does not have to be atomically written. Furthermore, the slot-headers of pages A and B do not have to be stored in a specific order as long as they are flushed to PM before their transaction commit mark is written.

After the transaction commits and its commit mark is flushed to the slot-header log, we immediately start checkpointing the slot-header of each page from the slot-header log to the actual pages A and B. This is unlike legacy checkpointing where we generally postpone checkpointing until on opportune time. This eager checkpointing is done so that other transactions do not have to check the slot-header log. Once checkpointing is done, the updated records in the pages become accessible by other transactions, and we can safely remove the slot-header log. This is a feasible approach as our slot-header logging is lightweight since each log frame per slotted-page is just the metadata of each page and writes are happening in PM. In contrast, in legacy write-ahead logging, checkpointing is a very expensive operation because all dirty pages are being flushed to slow block device storage.

4. Persistent Slotted Paging in SQLite

To show the applicability of our failure-atomic persistent slotted-paging, we present two versions of our implementation.

4.1 FASH

The first persistent B-tree version, *FASH* (Failure-Atomic Slot-Header), employs slot-header logging for all modifications to the B-tree in SQLite. In this implementation, even if a transaction inserts a single record and modifies only a single leaf page, we perform slot-header logging. An advantage of this implementation is that the size of the record offset array can be arbitrarily large (and not limited to the cache line

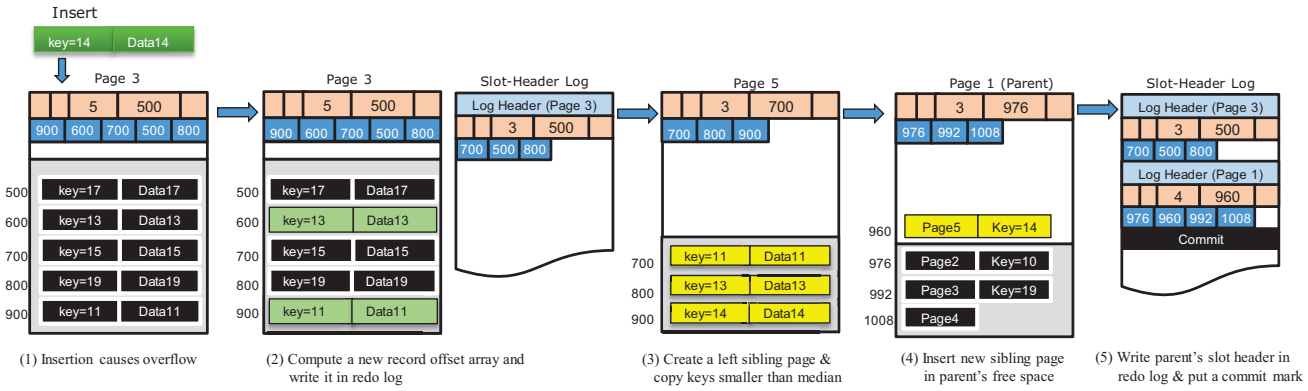


Figure 4: Phase 1: Slot-Header Logging in B-tree Split

size) allowing each page to hold a large number of records. Also, FASH does not require hardware transactional memory.

Figure 4 illustrates an example of how slot-header logging works when a B-tree page overflows. (1) As in a legacy B-tree split algorithm, we choose a median key and construct a new record offset array in the CPU cache, where we remove the offsets of the records whose keys are smaller than the median key. (2) This new record offset array is stored in a slot-header log without changing the original B-tree page's slot-header. (3) We allocate a new left sibling page, and the records whose keys are smaller than the median key are copied to the new left sibling page as in a legacy B-tree split algorithm. (4) Since a new left sibling page is created, we store the pointer to the left sibling page in its parent page. The largest key in the left sibling page and its pointer are written to the free space of the parent page. The slot-header of the parent page is updated accordingly in the CPU cache and written to the slot-header log. If the parent page overflows, the split is recursively propagated up to the root page. (5) Once a transaction completes, we put a commit mark of the transaction in the slot-header log.

4.2 FAST

As another implementation of our proposed scheme in the SQLite B-tree, we implement *FAST* (Failure-Atomic Slot-header with in-place commiT), the scheme that employs both slot-header logging and in-place commit with hardware transactional memory. Here, in-place commit is used when the transaction involves only a single page, while slot-header logging is used when the transaction involves multiple pages such as in the case requiring a B-tree page split operation.

With the failure-atomic cache line write function, we restrict the size of the slot-header in a B-tree leaf page to be no larger than the cache line size due to the limitation of hardware support. With a 64-byte cache line, the slot-header of the B-tree leaf page can hold a maximum of 28 records $((64-8)/2)$. Also, note that the failure atomic cache line write function can atomically sort a maximum of 28 records

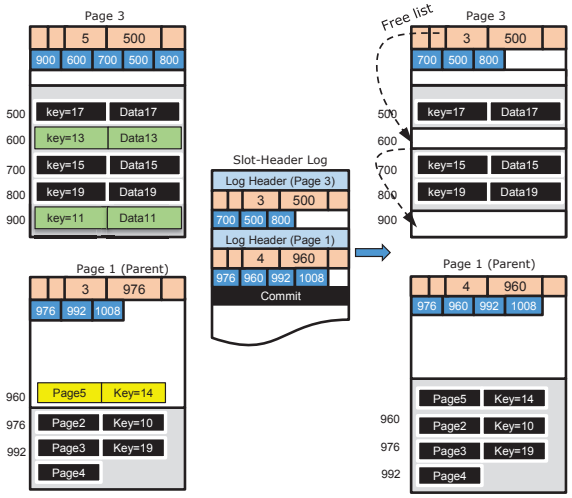


Figure 5: Phase 2: Checkpointing with Slot-Header Log in B-tree Split.

in a leaf page without making use of a redo log. Regarding the slot-header of the internal pages, since an internal B-tree page is updated only when a leaf page splits, note that the internal tree page updates are always performed along with a leaf page update. Hence, we use slot-header logging for all modifications involving internal tree pages. Consequently, the slot-header size of internal tree pages can be arbitrarily large.

When a transaction commits, the FAST implementation checks whether the transaction will modify multiple pages, whether it causes a page overflow, or whether it needs defragmentation (described in the next subsection). In these cases, that is, if it is not a single page modification, it simply falls back to slot-header logging.

4.3 Defragmentation

Failure-atomic slotted-paging does not overwrite any previous records until the transaction puts a commit mark. Oth-

erwise, recovery becomes impossible and consistency will not be guaranteed. Therefore, failure-atomic slotted paging is not free from the fragmentation problem as it does not have the freedom to shift record positions. This is unlike volatile buffer caching where records can be moved around without restriction.

Figure 5 illustrates how checkpointing is performed for the B-tree split shown in Figure 4. When the transaction puts a commit mark in the slot-header log, updated slot headers in the slot-header log are immediately checkpointed to the original pages. After the checkpointing is done, the parent page (page 1) will have a pointer to the new child (page 5), and page 3 will now have only 3 records and fragmented free spaces. Instead of compacting free spaces, we manage them as a linked list - *free list* as shown in Figure 5. The free list can be reconstructed from the record offset array from scratch if the total free space in the slot header does not match the size of the free list. Hence, updates to the free list do not have to be failure-atomic. Besides this, inconsistent free lists can be corrected in a lazy manner when we need to defragment.

Note that the slotted-page structure can accommodate variable-length records. If a transaction inserts a large record, which is larger than any available contiguous free space, we need to combine fragmented free spaces to store the large record. For example, if we insert a 200-byte record into *page 3*, we need to move around three valid records to make a contiguous space, which may require a large number of memory operations.

In order to resolve such a fragmentation problem, our failure-atomic slotted-paging employs the copy-on-write technique. If the total fragmented free space is large enough for a new record but none of them is large enough for the new record, defragmentation is done by allocating another slotted-page in PM space and contiguously copying the valid records to the new slotted-page. Once this is done, we update the pointer to the fragmented page in its parent page with the pointer to the new defragmented page.

Defragmentation is also often necessary for transactions that insert multiple records. Suppose an insertion causes an overflow and splits a page. The overflowing page needs to retain all the previous records until the transaction commits. However, if the next insert statement in the same transaction inserts a record into the same overflowing page, we perform a copy-on-write in order to avoid overwriting.

Defragmentation is performed on-demand in our implementation, but as we will show in Section 5, the overhead of defragmentation accounts for less than 0.02% of B-tree insertion time.

4.4 Crash Recovery for Slot-Header Logging

In this section, we discuss various system failures that can occur during slotted-paging and slot-header logging.

In our failure-atomic slotted-paging, we do not allow changing the original slotted-page until a transaction commits. Suppose a system crashes while splitting a page. As shown in Figure 4(2), the original slotted-page is not altered. Thus, recovery is trivial. We can simply ignore the slot-header logs. If the system crashes after a left sibling page is created (shown as Figure 4(3)), the sibling page can be safely garbage collected. If the system crashes after we insert a pointer to the left sibling page in the parent page as shown in Figure 4(4), we can ignore the pointer because the pointer is stored in the free space of the parent, which is not yet a valid entry since the slot header has not been updated. If the system crashes before or during checkpointing but after we put a commit mark in the slot-header log (Figure 4(5)), then we can recover by replaying the log entries. Recall that checkpointing is the process of copying the updated slot-headers in the log to the actual slotted-pages.

The only condition that must be guaranteed in slot-header logging is that all the records and their corresponding slot-headers be flushed to PM before we put a transaction commit mark in the log. That is, the ordering of memory write operations does not have to be preserved. This is because the log entries are all meaningless unless we have a valid commit mark in the log. This follows the approach taken in NVWAL [15].

Our logging scheme makes direct modifications to the actual persistent slotted-pages. However, our failure-atomic slotted-paging tolerates inconsistent updates caused by memory write failures, `mfence` failures, or `clflush` failures. This is because the updates are done in free space, which can be considered as perishable scratch space until the commits are guaranteed. For inconsistencies that arise in free space, the recovery process does not have to take any action as the partially written record can be ignored and again be taken in as free space.

5. Evaluation

Our testbed is a workstation that has four Intel Xeon Haswell-EX E7-8860 v3 processors (2.20GHz, 16x32KB instruction cache, 16x32KB data cache, 16x256KB L2 cache, and 40MB L3 cache) and 256GB of DDR3 DRAM. The Xeon Haswell-EX processors in our testbed also support TSX (Transactional Synchronization Extensions) including RTM. For our experiments, we set the scaling governor to performance to make the processors run at maximum frequency.

We implement our failure-atomic slotted-paging in SQLite 3.8. Since persistent memory is not yet commercially available, we conduct our evaluation with *Quartz* [3], a DRAM-based PM performance emulator that runs on the latest Intel Sandy Bridge, Ivy Bridge, and Haswell processors [38]. Quartz comprises a kernel module and a user-mode library that emulates PM by injecting software delays per each epoch and throttling the bandwidth of remote DRAM using thermal control registers. Quartz cannot emulate both latency and bandwidth at the same time. Hence,

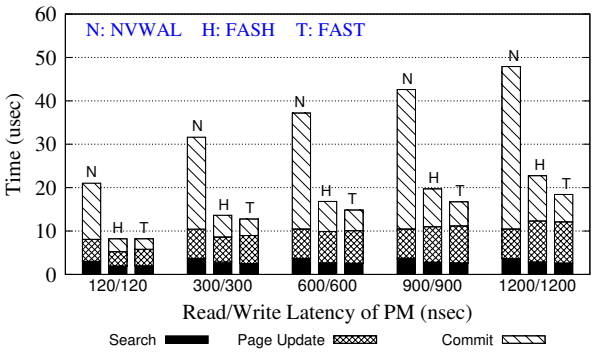


Figure 6: Breakdown of Time Spent for B-tree Insertion in SQLite as Read/Write Latency of PM is Varied

we emulate the read latency of PM using Quartz while assuming that the bandwidth of PM is equal to that of DRAM. Note that our experiments are more sensitive to latency than bandwidth.

Since write memory latency emulation is not yet supported in publicly available Quartz [3] implementation, we emulate PM write latency by introducing an additional delay after each `clflush` instruction, as was used to emulate PM latency in [10, 15, 39]. For `store` instruction, we do not insert the delay as the CPU cache can hide it. Quartz runs application threads on a PM-only mode or DRAM+PM mode. We use the PM-only mode for failure-atomic slotted-paging and DRAM+PM mode for NVWAL. In DRAM+PM mode, the emulator binds the NVWAL thread and its volatile memory to the first socket while the remote memory on the second socket is emulated as persistent memory. In NVWAL, we allocate virtual persistent memory for write-ahead logs via the `pmalloc()` and `pfree()` functions of Quartz.

For all the experiments in this section, we report and compare the performance of three schemes, namely, NVWAL, FASH, and FAST. NVWAL is the current state-of-the-art scheme proposed by Kim et al. [15]. As NVWAL makes use of volatile buffer cache, we use the DRAM latency numbers for this part of memory access. For FASH and FAST, all accesses are to PM so PM latencies are used. Also, all numbers reported except the last query processing throughput experiments, shown in Figure 11 and Figure 12, are the time spent on database buffer caching (SQLite pager) and B-tree code. The time for SQL parsing and SQLite bytecode processing are not included as they are irrelevant to our modifications. All results reported in this section are the average of 5 runs where for each run we take the average of 100,000 insertions each invoked through an INSERT database transaction statement with randomly generated keys, unless otherwise stated.

Figure 6 shows the first set of experiments, that is, the average time to insert a single 64 byte record per transaction, as we vary both read and write latencies of emulated PM. In our testbed, local DRAM access latency is measured as 120 nsec. Overall, the total insertion time increases as PM latency increases for all three schemes. While the latency increases from 300 nsec to 1.2 usec, the insertion time does not increase by 4 times because of the computation time and CPU cache effect. Although failure-atomic slotted-paging does not benefit from low DRAM latency, failure-atomic slotted-paging is 2.5 $\sim$ 2.6x faster than NVWAL even when PM latency is 1.2 usec. We believe 1.2 usec is a conservative number, considering that previous studies have assumed PM latency to be around 500 nsec [15, 28] and as persistent memory in the horizon is anticipated to show shorter latency [1, 36].

In detail, the insertion time is composed of three parts, `Search`, `Page Update`, and `Commit` times. (i) `Search` time is the time to traverse a B-tree from its root page to a leaf page where we insert a new record, and this is affected only by the read latency. (ii) `Page Update` time is the time from when we have found the leaf page to insert the record to the time when we finish updating the leaf page or parent pages in the database buffer cache, not including the commit operation. (iii) `Commit` time in NVWAL is the time from when the transaction starts copying dirty pages in volatile buffer cache to write-ahead log in PM to the time when it puts a commit mark in the write-ahead log. In FASH, `Commit` time is the time from when the transaction starts writing the updated slot-headers to the slot-header log in PM to the time when it finishes checkpointing and deletes the log. In FAST, `Commit` time is no different from FASH if slot-header logging takes place. But if in-place commit occurs, `Commit` time is the time to write an updated slot-header in-place via RTM transaction.

Overall, `Page Update` time of NVWAL is slightly slower than that of FAST and FASH when NVRAM latency is low even though this time in NVWAL does not include the time to create and flush WAL frames. This is because NVWAL performs copy-on-write processing when a page overflows and splits, while FAST and FASH perform in-place updates that minimize memory writes. However, as NVRAM latency increases, `Page Update` time of FAST and FASH takes longer than NVWAL. This is because in FAST and FASH the overhead of flushing the updated new record to the free space of a slotted-page is included, while in NVWAL only the time to simply overwrite B-tree pages in the volatile buffer cache is included without considering failure atomicity, consistency, and durability. This result is noteworthy because it shows that the copy-on-write processing overhead can be higher than the cache line flush overhead with in-place updates if NVRAM latency becomes similar to DRAM latency.

Figure 7 further breaks down the time spent on `Page Update` in order to show the benefits of persistent buffer caching. Since the size of a record is smaller than the page size, FASH and FAST needs a fewer number of `store` instructions, which explains why *in-place record insert* in FASH and FAST is faster than *volatile buffer caching* in

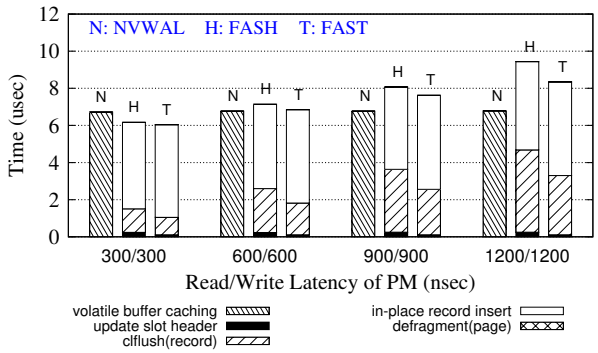


Figure 7: Breakdown of Page Update Time Spent for B-tree Insertion in SQLite as Read/Write Latency of PM is Varied

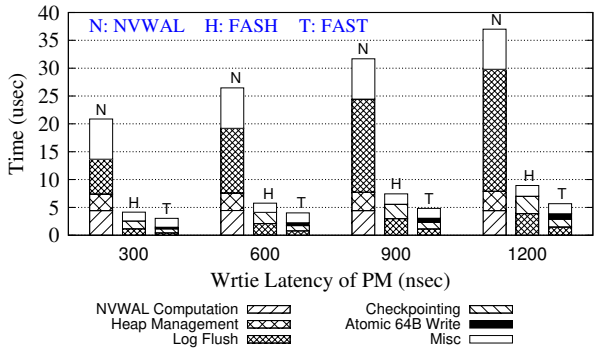


Figure 8: Breakdown of Commit Time Spent for B-tree Insertion in SQLite as Write Latency of PM is Varied

NVWAL. However, FASH and FAST call cache line flushes in order to persist the new record in free space, denoted as *clflush(record)*. This cache line flush overhead increases as the write latency of PM increases. `Page Update` time includes the overhead of copying slot headers to the slot-header log, denoted as *update slot header*. But compared to other overhead, it is almost negligible because its size is small and we do not call cache line flushes in this phase. The last element of `Page Update` time is the overhead of on-demand defragmentation, which we also find to be very small.

The `Commit Time` is where FAST improves significantly over NVWAL. Figure 8 shows FAST is up to 6x faster than NVWAL in terms of logging and commit overhead. For these experiments, the read latency of PM is set to 300 nsec while we vary the write latency of PM. Note that the commit time is independent of read latency of PM.

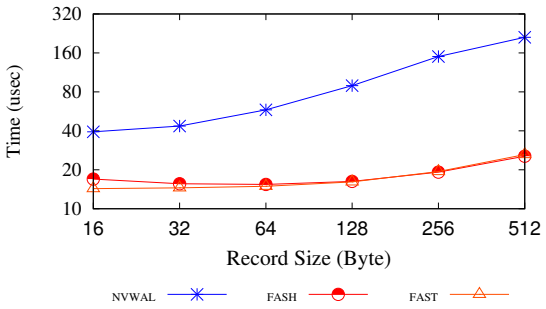
NVWAL, as denoted in Figure 8, has four distinct points that incur overhead. Let us compare these overhead in NVWAL with those of FAST. First of all, NVWAL computes differential logging in order to reduce the amount of memory writes, which accounts for about 4 usec (*NVWAL Computation*). In FAST, no such computation is involved. Hence, this portion does not appear for FAST. Second, NVWAL employs a user-level heap manager that manages PM address space, which also costs about 3 usec (*Heap Management*). On the other hand, FAST does not need a separate heap manager because everything is non-volatile. Hence, this portion also does not appear for FAST.

The third source of overhead for NVWAL is denoted as *Log Flush*. This is the time spent on calling cache line flush and memory fence instructions to store WAL frames. Similar overhead is incurred with FASH and FAST. However, we observe that the *Log Flush* overhead of FASH is considerably smaller compared to that of NVWAL and the *Log Flush* overhead of FAST is even smaller. This is due to the slot-header and in-place commit schemes that are key to FAST. Although NVWAL significantly reduces the size of WAL frames via differential logging, WAL frame and WAL frame header size is about 4x$\sim$8x larger than the size of slot-headers. With the atomic cache line write function, the FAST scheme writes the slot-header log only when a leaf page splits. Hence, the FAST scheme minimizes the cache line flush overhead for the slot-header log. The checkpointing overhead of FAST is 49% smaller than FASH (0.72 usec vs 1.42 usec) because of the in-place commit. As for NVWAL, we did not include the checkpointing overhead in Figure 8 because NVWAL performs checkpointing in a lazy manner while FASH and FAST performs checkpointing immediately, which affects the response time of each individual query. We find that *Log Flush* in FASH accounts for approximately 27.8% of its `Commit` time while it accounts for approximately 14.2% of the `Commit` time in FAST.

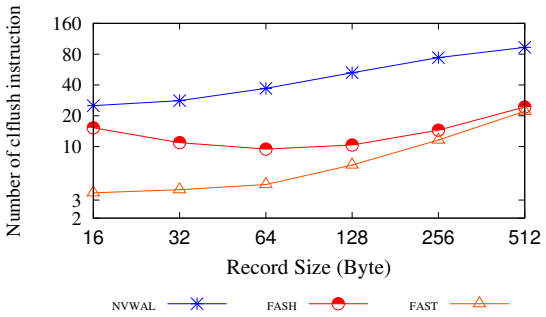
Finally, NVWAL and FAST share a miscellaneous computation portion denoted as *Misc*. However, we see from the figure that the difference between the two is significant. The main reason is that for NVWAL, considerable time is spent constructing indexes for WAL frames, while for FAST such management of a separate index is unnecessary as the slot-header itself is an index.

Figure 9 shows the comparative performance of FASH and FAST against NVWAL as we vary the size of the records that each transaction inserts when PM latency is set to 300 nsec and PM bandwidth is no different from DRAM. Figure 9(a) shows the average insertion time. Overall, FAST and FASH consistently outperform NVWAL. We see that the performance gap widens between FAST and NVWAL as the record size increases. This is because FAST does not duplicate write operations for large data. As the record size gets larger, NVWAL generates larger WAL frames while FAST writes the fixed-sized slot-header. In addition to the redundant memory writes to the volatile buffer cache, NVWAL incurs considerable overhead from differential logging, user-level heap manager, and WAL index construction.

Figure 9(b) compares the number of cache line flush instructions per insertion. A peculiar observation here is that for FASH, the number of cache line flush instructions are



(a) Insertion Time



(b) Number of clflush per Insertion

Figure 9: Insertion Time as Record Size is Varied

high for small record sizes. The reason behind this is that with smaller records, the slotted-page can hold more records, which results in a larger slot-header. As slot-header logging always duplicates the slot-header irrelevant to the record size, these duplicates may result in larger writes than the record itself. For example, a 1 KByte slotted-page can have a maximum of fifty 16 byte records. The slot-header, in this case, will be 108 bytes ($2 \cdot 50 + 8$). So FASH will write this 108-byte slot-header to the slot-header log and then checkpoint it to the actual page. This results in considerably larger writes due to the slot-header than the record itself, resulting in a larger number of cache line flush instructions.

In the case of FAST, when the record size is smaller than 64 bytes, FAST calls, on average, about 3 cache line flush instructions per insert transaction: (i) one cache line flush for writing the record in free space, (ii) one cache line flush for writing the slot-header, and (iii) one cache line flush that is the amortized overhead of a page split. If the record size is larger than a cache line, FAST calls multiple cache line flush instructions to write the record.

We now see how the number of insertions per transaction affects performance. Figure 10 shows the results as the number of 64 byte records inserted per transaction is increased (x-axis). Note that both the x- and y-axes are log scale. We also use 300 nsec PM latency for these experiment. Since each transaction inserts more than one record in these experiments, we do not evaluate FAST as the in-place commit scheme cannot update multiple pages at once.

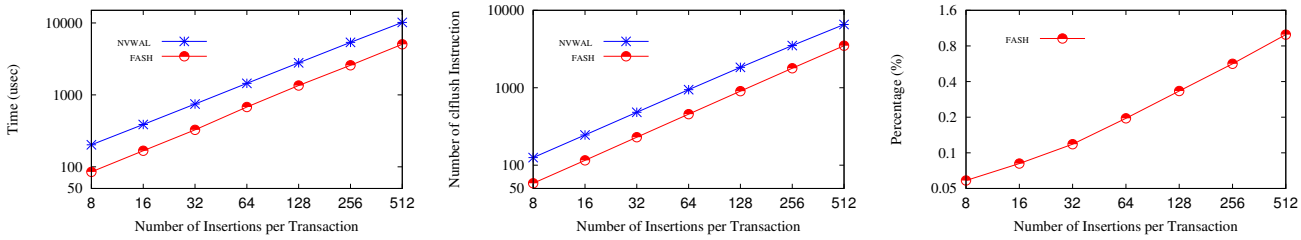
Figure 10a and Figure 10b show that the query execution time and the number of cache line flush instructions increases linearly with the increase in insertions per transaction, respectively. As we insert more records in each transaction, we need more cache line flushes. More specifically, when a transaction inserts 8 records, FASH calls 58 cache line flush instructions on average, i.e., about 7 cache line flushes per record, while NVWAL calls 125 cache line flush instructions, i.e., about 16 cache line flushes per record. As the transaction size increases, the probability of inserting multiple records into the same page increases. If we insert a record into the page that has just split in the same transaction, the copy-on-write that was described in Section 4.3 occurs in slot-header logging. Since the copy-on-write duplicates redundant records, it increases the number of cache line flush instructions. Thus, the number of cache line flush instructions in slot-header logging increases at a slightly faster rate than NVWAL as the transaction size grows. Even so, even when each transaction inserts 512 records, FASH still calls only about half as many cache line flush instructions than NVWAL.

The copy-on-write also occurs when a page is fragmented. In Figure 10c, we show the results when we measure the proportion of cache line flushes caused by defragmentation out of the total cache line flushes in FASH. As shown in the figure, when each transaction inserts 8 records, only 0.06% of cache line flush instructions are called due to defragmentation. Even when each transaction inserts 512 records, the defragmentation overhead accounts for only 1% of the total cache line flush overhead.

We now consider the real world effects of our proposed schemes. In Figure 11, we run Mobibench [2] that inserts 1000 records of 136 bytes and measure the end-to-end throughput of FAST, FASH, and NVWAL including SQL parsing overhead and SQLite bytecode processing overhead.

The transaction throughput of NVWAL is 26,890 transactions/sec when PM latency is set to 200 nsec. Compared to NVWAL, the transaction throughput of FASH and FAST are 31% (35,251 transactions/sec) and 33% (35,754 transactions/sec) higher, respectively. When we increase PM latency up to 1.4 usec (7x), the transaction throughput of FAST decreases slightly by around 15% (30,365 transactions/sec). These results are in line with a previous study that shows database transactions are less sensitive to PM latency [15].

Although FAST and FASH take advantage of eliminating volatile buffer cache, PM-only buffer caching is not favorable for read-only transactions when PM latency is higher than DRAM latency. In the experiments shown in Figure 12, we measure the query processing throughput with varying number of read and write transactions. We initialize a table with 100,000 records, and submit 10,000 transactions that insert or search 64 byte records. We set both read and write latencies of PM to 1.2 usec, which is unfavorable to FAST and FASH.



(a) Insertion Time

(b) Average Number of clflush instructions per Transaction

(c) Proportion of cache line flush instructions due to defragmentation

Figure 10: Insertion Performance as the Number of Insertions per Transaction is Varied

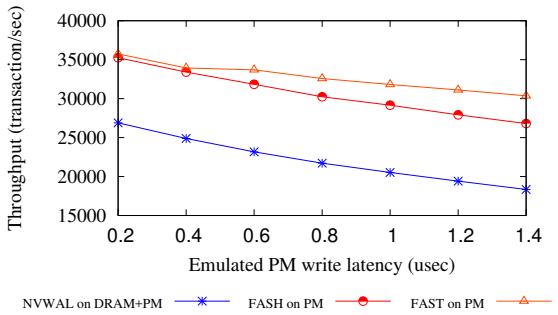


Figure 11: End-to-end Transaction Throughput for NVWAL, FASH, and FAST

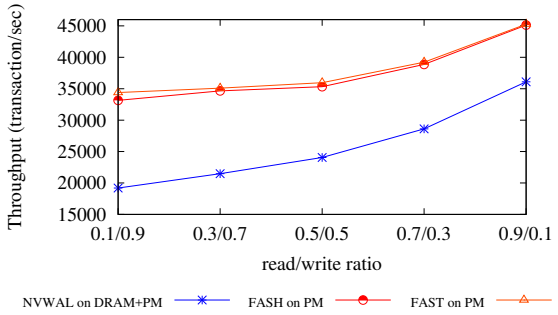


Figure 12: Mixed Workload

Since read transactions are often much faster than write transactions as database buffer caching helps avoid disk I/O, the transaction throughput increases as the percentage of read transactions increases. Although FAST and FASH perform faster than NVWAL for write transactions, NVWAL is favorable for read-heavy transactions as it can benefit from volatile buffer caching on faster DRAM. However, as shown in Figure 12, we observe that FAST and FASH perform faster than NVWAL even when more than 90% of the transactions are read-only transactions. This is because write transactions have a higher performance effect on overall query processing throughput [16, 43]. This result is noteworthy because it shows that PM-only data buffer caching performs faster than hybrid memory systems with PM and DRAM.

6. Conclusions

Emerging byte-addressable persistent memory enables various novel recovery methods. In this work, we proposed a novel *"failure-atomic slotted-page structure"* for persistent memory that minimizes redundant write operations. Our failure-atomic slotted-page structure uses free space of each slotted-page as scratch space without violating the consistency of the page. When the transaction commits, it converts the dirty portion of the page to valid records by atomically updating the header of the page. Our in-place commit and slot-header logging schemes effectively eliminates unnecessary redundant copies of database pages and minimizes the number of write operations when PM is used as the database buffer cache.

We evaluated our proposed failure-atomic slotted-page structure against NVWAL, which is the state-of-the-art logging method for PM. Our extensive performance evaluation showed that failure-atomic slotted-paging shows *optimal* performance - only 3 cache line flushes for database transactions that insert just a single record. Even for larger transactions that update more than one database page, slot-header logging reduces the logging overhead to at least 1/4 and up to 1/6 compared to NVWAL. These results imply that PM-only memory systems may perform faster than hybrid memory systems with PM and DRAM.

7. Acknowledgement

We thank the anonymous reviewers for their suggestions and comments on the early draft of this paper. This research was supported by MKE/KEIT (No.10041608, Embedded System Software for New Memory based Smart Devices), MSIP (No. R0190-15-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development) and National Research Foundation of Korea (No. 2014R1A1A2058843).

References

[1] Intel and Micron produce breakthrough memory technology. *https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology*.

[2] Mobibench. *https://github.com/ESOS-Lab/Mobibench*.

[3] Quartz. *https://github.com/HewlettPackard/quartz*.

[4] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB Journal*, 11(3):198–215, November 2002.

[5] Shimin Chen and Qin Jin. Persistent B+-Trees in non-volatile main memory. *Proceedings of the VLDB Endowment (PVLDB)*, 8(7):786–797, 2015.

[6] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persisten objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[7] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

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

[9] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. High performance database logging using storage class memory. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)*, pages 1221–1231, 2011.

[10] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. Nvram-aware logging in transaction systems. *Proceedings of the VLDB Endowment*, 8(4), 2014.

[11] Taeho Hwang, Jaemin Jung, and Youjip Won. Heapo: Heap-based persistent object store. *ACM Transactions on Storage (TOS)*, 11(1), 2014.

[12] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages (ASPLOS))*, 2016.

[13] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2013.

[14] Junghoon Kim, Changwoo Min, and Young Ik Eom. Reducing Excessive Journaling Overhead with Small-Sized NVRAM for Mobile Devices. *IEEE Transactions on Consumer Electronics*, 6(2), June 2014.

[15] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. Nvwal: Exploiting nvram in write-ahead logging. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[16] Wook-Hee Kim, Beomseok Nam, Dongil Park, and Youjip Won. Resolving journaling of journal anomaly in android i/o: Multi-version b-tree with lazy split. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2014.

[17] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 399–411, 2016.

[18] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2013.

[19] Victor Leis, Alfons Kemper, and Thomas Neumann. Exploiting hardware transactional memory in main-memory databases. In *Proceedings of the 30th International Conference on Data Engineering (ICDE)*, 2014.

[20] D. Lomet and B. Saltzberg. Access methods for multiversion data. In *Proceedings of 1989 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1989.

[21] Sparsh Mittal and Jeffrey S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, June 2015.

[22] Sparsh Mittal, Jeffrey S. Vetter, and Dong Li. A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1524–1537, June 2015.

[23] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Nathan Binkert, and Parthasarathy Ranganathan. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the ACM Conference on Timely Results in Operating Systems (TRIOS)*, 2013.

[24] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[25] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. Sqlite optimization with phase change memory for mobile applications. *Proceedings of the VLDB Endowment (PVLDB)*, 8(12):1454–1465, 2015.

[26] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys 16)*, 2016.

[27] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.

[28] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 265–276, 2014.

[29] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. Storage management in the NVRAM era. *Proceedings of the VLDB Endowment*, 7(2):121–132, October 2013.

[30] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 2005.

[31] Andy Rudoff. Programming models for emerging non-volatile memory technologies. *;login*, 38(3):40–45, June 2013.

[32] Priya Sehgal, Sourav Basu, Kiran Srinivasan, and Kaladhar Voruganti. An empirical study of file systems on nvm. In *Proceedings of the 31st International Conference on Massive Stroage Systems (MSST)*, 2015.

[33] Kai Shen, Stan Park, and Meng Zhu. Journaling of journal is (almost) free. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2014.

[34] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, 2005.

[35] Craig A. N. Soules, Garth. R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX conference on File and Storage Technologies (FAST)*, pages 43–58, 2003.

[36] Kosuke Suzuki and Steven Swanson. The non-volatile memory technology database (nvmdb). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, May 2015. http://nvmdb.ucsd.edu.

[37] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *9th USENIX conference on File and Storage Technologies (FAST)*, 2011.

[38] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A lightweight performance emulator for persistent memory software. In *15th Annual Middleware Conference (Middleware '15)*, 2015.

[39] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[40] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. Using restricted transactional memory to build a scalable in-memory database. In *ACM SIGOPS/Eurosys European Conference on Computer Systems (EuroSys)*, 2014.

[41] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[42] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[43] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, and Khai Leong Yong. NV-Tree: reducing consistency const for NVM-based single level systems. In *Proceedings of the 13th USENIX conference on File and Storage Technologies (FAST)*, 2015.

[44] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *Proceedings of the 31st International Conference on Massive Stroage Systems (MSST)*, 2015.

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