NVWAL: Exploiting NVRAM in Write-Ahead Logging

Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, Youjip Won

Ulsan National Institute of Science and Technology (UNIST)
{okie90,jwkim,wbaek,bsnam}@unist.ac.kr

Hanyang University
yjwon@hanyang.ac.kr

Abstract

Emerging byte-addressable non-volatile memory is considered an alternative storage device for database logs that require persistency and high performance. In this work, we develop *NVWAL* (NVRAM Write-Ahead Logging) for SQLite. The proposed NVWAL is designed to exploit byte-addressable NVRAM to maintain the write-ahead log and to guarantee the failure atomicity and the durability of a database transaction. The contribution of NVWAL consists of three elements: (i) byte-granularity differential logging that effectively eliminates the excessive I/O overhead of filesystem-based logging or journaling, (ii) transaction-aware lazy synchronization that reduces cache synchronization overhead by two-thirds, and (iii) user-level heap management of the NVRAM persistent WAL structure, which reduces the overhead of managing persistent objects.

We implemented NVWAL in SQLite and measured the performance on a Nexus 5 smartphone and an NVRAM emulation board - Tuna. Our performance study shows the following: (i) the overhead of enforcing strict ordering of NVRAM writes can be reduced via NVRAM-aware transaction management. (ii) From the application performance point of view, the overhead of guaranteeing failure atomicity is negligible; the cache line flush overhead accounts for only 0.8∼4.6% of transaction execution time. Therefore, application performance is much less sensitive to the NVRAM performance than we expected. Decreasing the NVRAM latency by one-fifth (from 1942 nsec to 437 nsec), SQLite achieves a mere 4% performance gain (from 2517 ins/sec to 2621 ins/sec). (iii) Overall, when the write latency of NVRAM is 2 usec, NVWAL increases SQLite performance by at least 10x compared to that of WAL on flash memory (from 541 ins/sec to 5812 ins/sec).

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

Keywords Write-ahead-logging; Non-volatile memory

1. Introduction

In the era of mobile computing, SQLite [3] is arguably the most widely deployed relational database management system because Android devices extensively use it to persistently manage application data for a wide range of apps such as contact managers, Twitter, and even web browsers. Due to its serverlessness, compactness, and the ability to form an integral part of a program, SQLite has become a core component of the Android software stack.

Despite its popularity, SQLite is far from being satisfactory in terms of efficiently exploiting the underlying hardware resources, mainly because the EXT4 filesystem journals the database journaling operation [17, 21, 23, 33, 38]. Beginning with SQLite 3.7, write-ahead logging (WAL) has become available as an alternative option to rollback journal modes. The WAL recovery scheme is designed in such a way that any update operation to a database page has to first be recorded in a permanent log file. WAL significantly improves the performance of SQLite because WAL needs fewer `fsync()` calls as it modifies a single log file instead of two, i.e., a database file and a rollback journal file [23]. However logging a single database transaction in SQLite WAL mode still entails at least 16 KBytes I/O traffic to underlying storage mainly due to metadata journaling overhead in the EXT4 file system [23].

The recent advances of byte-addressable NVRAM (Non-Volatile RAM) open up a new opportunity to reduce the high latency of NAND flash memory and the I/O overhead of block granularity journaling [10, 14, 35, 37, 41, 43]. As the log file is often much smaller than the database file and is accessed very frequently, high performance NVRAM is considered to be a promising alternative storage device for write-ahead logs. First, NVRAM can be used as a disk replacement; each page in a log file can be flushed to the NVRAM file system with low latency using a legacy block device I/O interface [15, 18, 22]. Alternatively, in order to take advantage of the fine granularity of byte-addressable

ASPLOS '16, April 2–6, 2016, Atlanta, Georgia, USA
Copyright © 2016 ACM 978-1-4503-4091-5/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2872362.2872392

NVRAM, each log record in a transaction can be stored in NVRAM as a persistent object [15, 36].

In this work, we take the latter approach and implement write-ahead logging for NVRAM (*NVWAL*) in SQLite. Our implementation of NVWAL allows reordering of memory write operations and minimizes the overhead of the cache line flush via byte-granularity differential logging. In addition, NVWAL reduces the overhead required to manage persistent objects via user-level heap management, while guaranteeing the failure atomicity. NVWAL is not a completely novel logging data structure, but it orchestrates SQLite, the OS persistent heap manager, and NVRAM layers to effectively leverage high performance NVRAM.

Over the past decades, processor design for volatile memory has evolved. Modern processors often do not preserve the ordering of memory write operations unless a memory barrier instruction is explicitly invoked. However, NVRAM must constrain the ordering of write operations to ensure correct recovery from system failures [37]. In order to avoid limiting NVRAM write concurrency and improve write throughput, several memory persistency models are being considered for emerging NVRAM technologies [37]. Assuming one of these NVRAM persistency models, NVRAM heap managers such as NV-Heaps [9] and Heapo [16], and NVRAM file systems such as SCMFS [46], PMFS [11], and BPFS [10] have been proposed. Using these NVRAM file systems or NVRAM heap managers, applications can deal with the ordering issues of NVRAM writes.

Database write-ahead logging does not have to strictly enforce the ordering of memory write operations as long as it guarantees that a commit flag is flushed only after all the dirty records are logged in WAL. Therefore, the proposed NVWAL does not call cache line flush instructions per log entry, but delays calling them in a lazy manner in order to reduce the overhead of the cache line flush by 1/3x.

It has been reported that flushing cached data to NVRAM can be extremely costly, especially when applications frequently call cache flush operations [47]. However, our performance study shows that the cost of data persistence in SQLite is no higher than 4.6% of query execution time, and the cost ratio even decreases when we insert a larger number of records in a transaction. SQLite throughput is governed both by I/O performance and by CPU performance. CPU overhead accounts for a dominant fraction of the transaction processing time. I/O constitutes about 30% of query processing time when SQLite WAL is stored in slow storage [23]. However, when the WAL is stored in fast devices such as NVRAM, the I/O overhead becomes much less significant than when the WAL log is located in the slow block device and when it is maintained through the expensive filesystem interface.

The main contributions of this work are as follows.

- **Byte-granularity differential logging**
  In stock SQLite WAL mode, write-ahead logging stores an entire B-tree page in a log file. In order to reduce the I/O overhead of SQLite logging on NVRAM, our NVWAL writes only the dirty portion of a B-tree page, so as to take advantage of the byte-addressability of NVRAM.
- **Transaction-aware memory persistency guarantee**
  In order to persist memory writes and to prevent incorrect reordering of the memory writes, cache line flush instructions and memory barriers must be called properly. We develop a transaction-aware persistency guarantee via *lazy synchronization*, which guarantees the atomicity and the durability of the database. This transaction-aware memory persistency guarantee enforces the persist order only between a set of log-write operations and the commit operation. It eliminates the unnecessary overhead of enforcing memory write ordering constraints.
- **User-level NVRAM management for WAL**
  The NVWAL structure is designed to minimize the overhead of managing non-volatile memory objects. Allocating and deallocating non-volatile memory blocks using a kernel-level NVRAM heap manager has high overhead due to ensuring consistency in the presence of failures. In order to avoid such overhead, we pre-allocate a large NVRAM block and manage the user-level heap inside the block where we store multiple WAL frames. The user-level NVRAM management helps reduce the number of expensive system calls to the NVRAM heap manager interface.
- **Effect of NVRAM latency on application performance**
  We observe that database transactions are less sensitive to NVRAM latency than expected. This is because the I/O overhead does not account for a large fraction of transaction execution time. In addition, we reduce the overhead of guaranteeing the failure atomicity via NVWAL. Consequently, we make the performance of SQLite more insensitive to NVRAM latency. This result holds a profound implication for NVRAM and CPU manufacturers because they need to invent and improve non-volatile material compounds and architectures to meet the expected application performance [34].

Combining all these elements, NVWAL on NVRAM yields transaction throughput 10x higher than that of WAL on flash memory (541 ins/sec vs. 5812 ins/sec).

The rest of the paper is organized as follows: In section 2, we briefly discuss the background. In section 3, we present our design and implementation of NVWAL on SQLite. In section 4, we discuss how the transaction-aware memory persistency guarantee can enforce minimal ordering of NVRAM write operations while guaranteeing durability and failure atomicity. Section 5 provides the performance results and analysis. In section 6, we discuss other research efforts related to this study. We conclude the paper in section 7.

2. Background: Write-Ahead Logging

In database systems, transactions make modifications to copies of database pages in volatile buffer cache memory and write the dirty pages to persistent storage when the transactions commit. In SQLite write-ahead logging (WAL) mode, the dirty pages are appended to a separate log file and the original pages remain intact in the database file. If a system crashes before a transaction commits or if a transaction aborts, dirty pages written by the aborted transaction can be simply ignored by the WAL recovery process because the original pages are available in the database file.

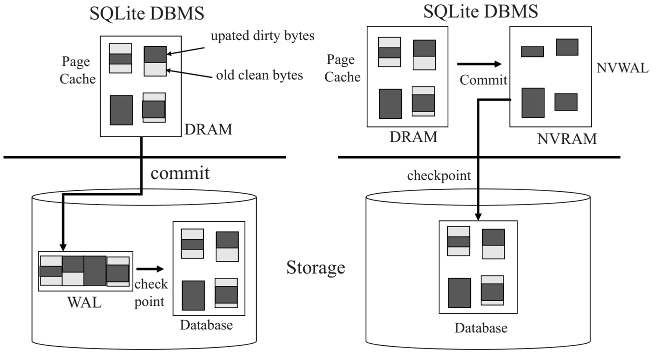
In WAL mode, the checkpointing process periodically batches the dirty pages in the log to the database file. When all the dirty pages in the log are written and checkpointed to the database file, the log is truncated so that it does not grow infinitely. Checkpointing occurs whenever all database sessions are closed or the number of log entries reaches the predefined limit (1000 pages in SQLite).

Write-ahead logging on block device storage guarantees the *atomicity* and *durability* of transaction commits and the ordering of transaction commits via `fsync()` system calls. In SQLite, a transaction commit mark is stored in the log frame header of the last appended dirty page, and the dirty pages and their checksum bytes are flushed to block device storage all together by `fsync()` system call. The order of transaction commits is guaranteed in SQLite by enforcing the commit marks of transactions, which are appended to the log file in the order of the commits. Because the commit mark and the dirty pages are flushed by the same `fsync()` system call in SQLite, SQLite is subject to the latent system failure issues pointed out by [48]. However, resolution of these problems lies beyond the scope of this paper.

3. NVWAL: Design and Implementation

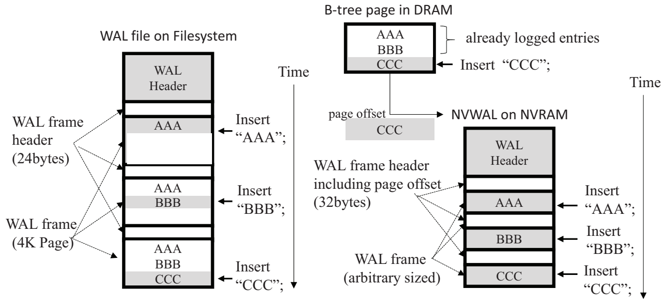
For the past several decades, memory technologies have evolved rapidly and now non-volatile memory devices such as phase-change RAM (PCRAM) and spin-transfer torque (STT) MRAM promise large capacity, high performance, low power consumption, and unlimited endurance. STT-MRAM is expected to meet the requirements of various computing domains because its performance is expected to be within an order of magnitude of that of DRAM [10, 11, 37].

However, as the price of NVRAM is not likely to be as low as that of flash memory, it is hard to expect that NVRAM will replace flash memory in the near future. On the other hand, it can also be inferred that NVRAM will not replace volatile DRAM without major changes in the software stack: volatility of memory is inevitable in current software and hardware design because system errors will be permanent if everything is non-volatile [6]. Therefore, we design and implement an SQLite write-ahead logging scheme for a system that has NVRAM along with DRAM and block device storage, as illustrated in Figure 1(b).



(a) Write-Ahead Logging (b) NVWAL

Figure 1: Write-Ahead Logging in NVRAM



(a) Write-Ahead Logging in SQLite (b) Byte Addressable Differential Logging

Figure 2: Byte Granularity Differential Logging

3.1 Write-Ahead Logging on NVRAM

A fair number of studies have proposed to store the database transaction logs in NVRAM [5, 8, 15, 42, 44]. When a transaction commits, the updated pages are either synchronized to a database file in journaling modes, or appended to a log file in WAL mode. Because transactions in WAL mode do not directly update the database file but buffer the frequent small updates in the write-ahead logs, storing write-ahead logs in high performance NVRAM can replace expensive block I/O traffic with lightweight memory write instructions; such a setup can take advantage of the byte-addressability of NVRAM as well. Figure 1(a) shows how NVRAM can be leveraged to accelerate database transactions.

3.2 Byte Granularity Differential Logging in NVWAL

In SQLite, write-ahead logging is designed to work with block device storage systems. It flushes an entire B-tree page (4 KBytes in normal SQLite configuration) to a WAL log file no matter how small a portion of the page is dirty. The size of the B-tree is aligned with the filesystem block size to avoid read-modify-write and torn-write problems in databases and file systems [31]. In most cases, a single database transaction yields changes in a small fraction of a B-tree page and leaves most of the contents of the B-tree page intact. In NVWAL, we exploit the byte-addressability of NVRAM and employ byte-granularity *differential logging* (also often referred to as *delta encoding*), which has been widely used in various systems including flash memory database systems [4, 13, 26, 28].

As illustrated in Figure 2(b), the NVWAL structure consists of an NVWAL header and pairs of a 32 bytes WAL frame header and arbitrary-sized WAL frames (log entries) created by differential logging. The NVWAL header contains database file metadata as in the legacy SQLite WAL header and a pointer to the available space for the next WAL frame to be stored (*next_frame*). Each WAL frame header consists of a commit flag, a checkpointing id number, a database page number, an in-page offset, a frame size, and checksum bytes for the WAL frame. For each WAL frame, we identify which portion of the B-tree page is dirty and truncate the preceding and trailing clean regions so that only the dirty portions of the B-tree page are flushed to NVRAM, as shown in Figure 2. By minimizing the memory copy, we can avoid the unnecessary overhead of cache line flush instructions.

3.3 User-Level NVRAM Heap Management

There exist several well-designed proposals on how to manage non-volatile memory pages under a persistent namespace and how to map the pages to application processes [9, 35, 41, 43, 46]. BPFS [10] is a transactional file system for NVRAM. NV-Heap [9] and Heapo [16] are heap-based persistent object stores that allow programmers to implement in-memory data structures without difficult reasoning about thread safety, atomicity, or memory access ordering.

For NVWAL, we employed Heapo [1] as our NVRAM heap manager so that (i) SQLite can map NVRAM to its address spaces, (ii) a non-volatile memory page can be identified by a persistent namespace and its address even when system reboots, and (iii) the NVRAM pages can be protected by access permission as in the file system. It should be noted that Heapo does not enforce memory persistency; the applications are responsible for properly adopting a persist barrier, memory barrier, and cache line flush to guarantee the failure atomicity.

System call is expensive. It crosses the protection boundary and the parameters are copied. The system call overhead becomes even more expensive if we rely on a kernel when allocating and deallocating NVRAM pages. Thus, we develop a user-level heap management scheme for NVWAL. This scheme allows us to manage the log at the user level and to minimize the system call interference. Instead of relying on a kernel feature to protect an object against corruption and against the race condition, we implement a simple tri-state flag for each NVRAM block, i.e., *free*, *in-use*, and *pending*.

```
input: WalHeader wh, PagePtr p, bool commit
while p do
    available_space = wh.getAvailableSpace() ;
    dirty_frame = compute_WAL_frame(p) ;
    if available_space < dirty_frame.size then
        /* get an nvram block in pending mode */
        block=nv_pre_malloc(BLOCK_SZ) ;
        ptr = wh.add_new_block(block) ;
        dmb(); /* data memory barrier */
        cache_line_flush(ptr,ptr+sizeof(void*)) ;
        dmb(); /* data memory barrier */
        persist_barrier() ;
        /* mark in-use flag of nvram block */
        nv_malloc_set_used_flag(block) ;
    end
    next_nv_frame = wh.next_frame ;
    /* store a dirty WAL frame in NVRAM */
    memcpy(next_nv_frame, dirty_frame, ..) ;
    nvFramePtrList.add(next_nv_frame) ;
    p=p.next ;
end
f=nvFramePtrList.first() ;
dmb() ; /* data memory barrier */
while f do
    cache_line_flush(f,f+f.header.frame_size) ;
    f=nvFramePtrList.next() ;
end
dmb(); /* data memory barrier */
persist_barrier() ;
if commit==true then
    lastFrame = nvFramePtrList.last() ;
    lastFrame.commitMark |= COMMIT ;
    dmb() ;
    cache_line_flush(lastFrame.commitMark,...);
    dmb() ;
    persist_barrier() ;
end
```

Algorithm 1: `sqliteWriteWalFramesToNVRAM()`

To enable user level heap management, we implemented `nv_pre_malloc()` and `nv_malloc_set_used_flag()` system calls on top of Heapo so that the system could manage the status of the allocated NVRAM blocks. `nv_pre_malloc()` allocates a set of NVRAM pages that occupy a consecutive address space, maps them to a fraction of the virtual address space of a process, and sets the status of the block to *pending*. When NVWAL persistently saves the address of a newly allocated NVRAM block in another NVRAM block as in a linked list, it calls `nv_malloc_set_used_flag()` to change its status from

[1] Heapo is available at https://github.com/ESOS-Lab/HEAPO

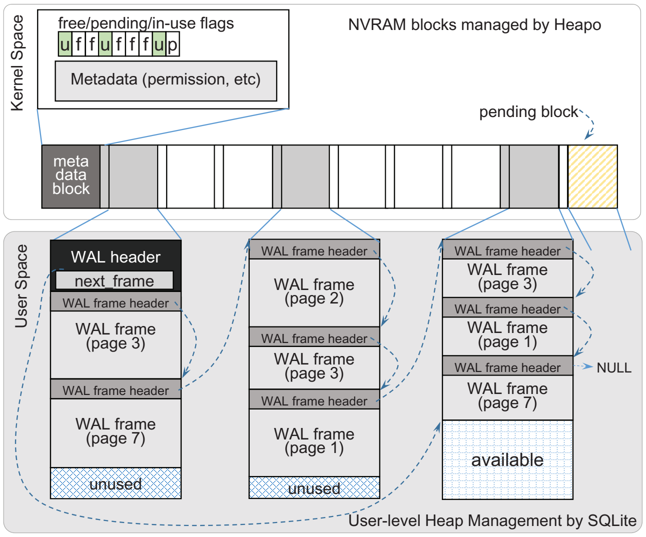
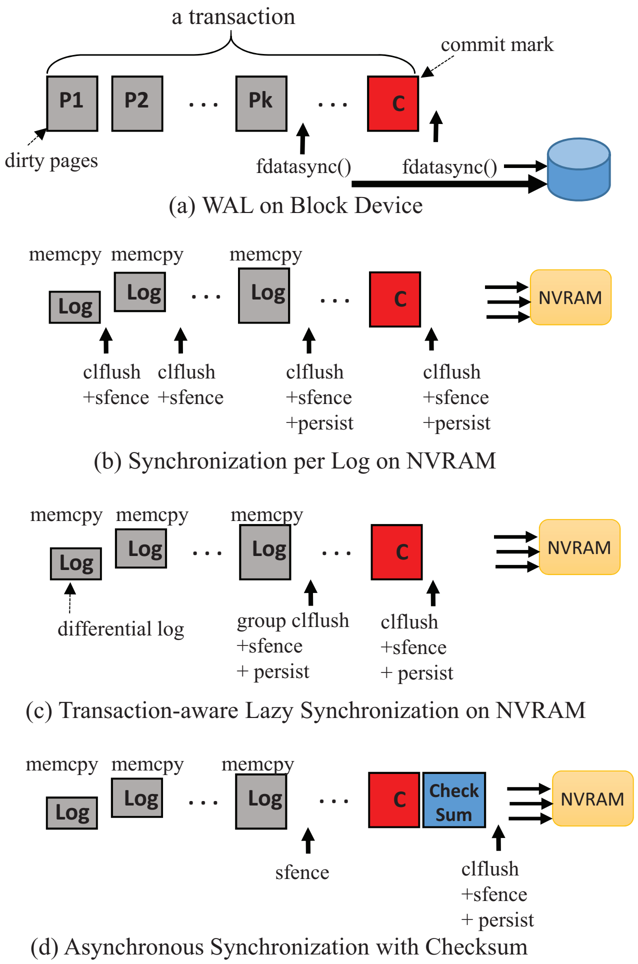


Figure 3: NVRAM Block Management in NVWAL



(a) WAL on Block Device

(b) Synchronization per Log on NVRAM

(c) Transaction-aware Lazy Synchronization on NVRAM

(d) Asynchronous Synchronization with Checksum

Figure 4: Transaction-Aware Persistency Guarantee

pending to *in-use*. If the system crashes while an NVRAM block is still in *pending* status, the SQLite NVWAL recovery process can safely deallocate the block.

Algorithm 1 shows how NVWAL allocates and manages NVRAM blocks. Figure 3 illustrates an example of NVWAL block management. The NVWAL heap consists of a metadata block and a set of pages from NVRAM. The metadata block contains the permission and the state of each block: free, pending or in-use.

In the example shown in Figure 3, if a transaction commits a new dirty page p, SQLite searches for page p's WAL frame in the write-ahead log. If page p's WAL frame is not in the log, the entire page p and its WAL frame header are copied to the available space of the last NVRAM block in the linked list if they fit. If the available space in the last NVRAM block is not large enough to hold them, we allocate another NVRAM block from Heapo, add the block to the linked list of NVRAM blocks, set the status of the block to *in-use*, and store the dirty WAL frame and its header in the newly allocated NVRAM block. If page p's WAL frames are found in the log, the differences between the page and the WAL frame are computed to construct a small WAL frame, as described in section 3.2; the small differential log entry is stored as a WAL frame in the NVRAM log. With the pre-allocated large NVRAM block, NVWAL reduces the number of calls to the expensive NVRAM heap manager's `nvmalloc()`. In the experiments we will describe in section 5, each NVRAM block stores 4.9 WAL frames on average when we fix the size of each NVRAM block to 8 KBytes.

4. Transaction-Aware Memory Persistency Guarantee

When NVRAM is used to replace block device storage, the atomicity and the durability of the database transactions must be ensured in the memory operations. NVWAL requires that we explicitly flush the appropriate cache lines to NVRAM to enforce the ordering constraints. The WAL recovery algorithm is based on the assumption that log entries are stored in the order of database transactions. Unfortunately, memory write operations can be arbitrarily reordered in today's processor designs. Without the hardware support `sfence` or `mfence` alone cannot guarantee the propagating of memory writes to NVRAM because writes can be cached in L1 or L2 caches. In order to guarantee that memory write operations flush data all the way down to NVRAM, cache line flush operations such as `clflush` should accompany `persist barrier` instructions such as Intel's `PCOMMIT`, as shown in Figure 4(b). `Persist barrier` ensures that the cache lines queued in the memory subsystem are persisted in NVRAM. Otherwise, a commit mark can be written to NVRAM before log entries are stored.

In X86 processors, `clflush` instruction invalidates and flushes a cache line from the memory cache hierarchy. In the ARM v7 architecture, there exist two cache flush in-

```
input: u32 start, u32 end
u32 lineLen = getCachelineLen();
/* Align start to cache line boundary*/
start = start & ∼(lineLen - 1);
u32 mva = start;
while $mva < end$ do
    /* clean data cache line to PoC by MVA */
    dccmvac(mva) ;
    mva = mva + lineLen ;
end
```

Algorithm 2: cache_line_flush() system call for ARM v7

structions: `dccimvac` and `dccmvac`. The former invalidates the flushed cachelines while the latter does not. Android 4.4 has not implemented the `clflush()` system call. For NVWAL, we implemented a `cache_line_flush()` system call using a `dccmvac` instruction, as shown in Algorithm 2. `cache_line_flush()` calls `dccmvac` instead of `dccimvac` because write-ahead logging in SQLite does not, indeed, need cache invalidation due to its append-only nature. We implemented `cache_line_flush()` as a system call despite the overhead of the kernel mode switch because the `dccmvac` instruction needs to access register 15 in privileged mode.

4.1 Transaction-Aware Memory Persistency Guarantee

We exploit the ordering and persistency constraints in write-ahead logging and develop a transaction-aware memory persistency guarantee scheme. The idea is simple and straightforward. A log-commit operation consists of two phases: (i) logging: writing a sequence of logs to NVRAM and (ii) commit: putting the commit mark to NVRAM to validate the logs. We enforce the ordering and persistency guarantee requirement only between the two phases by calling the expensive `clflush`, `sfence`, and `persist barrier`. The reason behind this is that the ordering of writing WAL frames does not necessarily have to be preserved as long as a transaction's commit mark is written after all the WAL frames are written to NVRAM. It should be noted that (i) SQLite is a serverless DBMS that does not allow multiple write transactions to run concurrently and (ii) a write transaction requires an exclusive lock on the entire database file.

Under a transaction-aware memory persistency guarantee, so that the processors can better utilize caches and memory banks, NVWAL allows processors to flush the WAL frames to NVRAM in any order. That is, NVWAL separates the phase in which the logs are being copied to NVRAM (`memcpy()`) and the phase in which the respective logs are synchronized to NVRAM (`cache_line_flush(p)`). Figure 4(c) schematically illustrates the behavior of the transaction-aware memory persistency guarantee. The details of the implementation can be found in Algorithm 1.

There are a few implementation specific issues that deserve further elaboration. We implemented NVWAL in ARM based platform. The cache line flush instruction in an ARM, `dccmvac`, is non-blocking. In order to ensure that the memory operations that mark the commit flag (line 30–34) are not reordered with the following `cache_line_flush()`, the cache line flush instruction should be preceded by the data memory barrier, `dmb`. The `dmb` instruction completes only when any previous explicit memory access instructions are all completed. The `dmb()` in line 22 ensures that all WAL frames are written at least to the L1 or L2 cache. In the proposed *lazy synchronization*, a batch of non-blocking `dccmvac` instructions are called in line 23–26 in order to flush the cache lines. The `dmb()` in line 27 is also needed in order to block until the preceding `cache_line_flush()` flushes all the WAL frames to NVRAM. `dmb()` in line 32 also ensures that the memory operation that marks the commit flag is not reordered with `cache_line_flush()`.

The commit mark resides at the WAL frame header and is one bit long. As in a prior work [10], we assume that NVRAM devices guarantee atomic writes for 8 bytes. This means that even if a random power failure occurs, capacitors in DIMM guarantee no corruption of 8 bytes. Because the commit mark is just a bit flag, the commit mark can be safely flushed to NVRAM with 8 bytes padding. If atomic writes can be done in the unit of a cache line, as in [35], cache line padding is necessary to prevent `cache_line_flush()` from flushing an unintended memory region.

4.2 Asynchronous Commit

We can further allow processors to utilize caches and memory banks by compromising the consistency using checksum bytes. Under this mechanism, we do not explicitly enforce the barrier between the logging phase and the commit phase. Instead, we compute the checksum of the logs that must precede the commit mark and store the checksum along with the commit mark. Figure 4(d) illustrates NVWAL with asynchronous log-commit; NVWAL asynchronously writes log entries, a commit mark, and checksum bytes without calling the cache line flush instructions.

Suppose a system crashes after it flushes a commit mark and checksum bytes but before all log entries are written to NVRAM. In the recovery phase, the checksum bytes will be found to be inconsistent with the written log entries and this will invalidate the commit mark of the transaction. Thus, the database recovery process can consider that the transaction has been aborted. However there is a chance that the written checksum bytes accidentally match the unwritten log entries. Hence, although the chance is very low, a system crash may corrupt a database file. We consider this asynchronous synchronization NVWAL as a performance comparison target because it minimizes the overhead of enforcing memory constraints.

4.3 Checkpointing and Crash Recovery

In WAL, the committed dirty pages in NVRAM are written back to the original database file in block device storage via periodic checkpointing. In NVWAL, we reconstruct the dirty pages by combining the log entries in the NVWAL log and the respective original database pages. When all the dirty pages are flushed to the database file via `fsync()`, the NVRAM blocks for the write-ahead logs can be safely truncated from the end of the list to the beginning. For each NVRAM block, we first set the tri-state flag of the NVRAM block to *free* and then call Heapo's `nvfree()` system call.

In SQLite WAL mode, the log file contains a complete history of dirty pages committed by transactions. When a system crashes, SQLite can recover the transactions by replaying the logs. The recovery algorithm of NVWAL is not very different from SQLite's stock recovery algorithm except that the dirty pages are reconstructed from byte-addressable WAL frames. However, complications can arise if the system crashes while updating the metadata of the NVRAM blocks.

NVWAL is for a forth-coming architecture, which is not available yet. Without a persist barrier, we are unable to perform a physical power-off test for NVWAL because `clflush` and the emulated persist barrier do not guarantee the failure atomicity. Instead, we resort to discussing the NVRAM recovery algorithm under various failure cases that can occur while a transaction is executing `sqliteWriteWalFramesToNVRAM()`.

- If a system fails while allocating an NVRAM block (line 6 of Algorithm 1), the allocated NVRAM block is marked as *pending*, but SQLite may not have written its reference in another persistent NVRAM block. When the system recovers, the heap manager can reclaim any pending NVRAM blocks to prevent a memory leak.
- If a system crashes after an NVRAM reference was stored in the WAL header but before the block was marked as in-use, the SQLite recovery process will find that the block was freed by the NVRAM heap manager's recovery process and the block's reference can be safely deleted.
- If system crashes while copying a dirty WAL frame to NVRAM using `memcpy()` (line 22 of Algorithm 1), SQLite can easily recover from the failure because the frame's transaction has not written a commit mark, thus the transaction is considered to have been aborted. In database transactions, writing a commit mark is the last operation of the transaction commit process.
- Recovery from checkpointing process' failure is also trivial and not different from legacy checkpointing recovery. Because the write-ahead logs will not be deleted before all the dirty pages are persistently stored in the database file, the SQLite recovery process can simply replay the checkpointing process to recover from the failure.

4.4 NVWAL and Persistency Model

In this section, we briefly discuss NVWAL implementation for strict and relaxed persistency models.

Memory persistency, proposed by Pelley et al [37], is a framework that provides an interface for enforcing the ordering constraints on NVRAM writes. The ordering constraints are referred to as "persists" to distinguish them from regular writes to the volatile memory [37]. Similar to memory consistency, memory persistency is largely classified into two classes – *strict persistency* and *relaxed persistency*.

Strict persistency integrates memory persistency into the memory consistency [37]. Under strict persistency, persist order must match the volatile memory order specified by the memory consistency model. Strict persistency is a simple and intuitive model in the sense that it provides a unified framework to reason about possible volatile memory and persist orders. In addition, it requires no additional persist barriers because the existing memory barriers can be used to specify persist ordering constraints. Strict persistency, however, may significantly limit persist performance because it enforces strict ordering constraints between persist operations.

In contrast, relaxed persistency decouples the memory consistency and persistency models [37]. Under relaxed persistency, persist order may deviate from the volatile memory order specified by the memory consistency model. Relaxed persistency requires persist barriers to enforce the order of persist operations. A representative relaxed persistency model is *epoch persistency* [10, 37]. In epoch persistency, persist barriers are used to divide persist operations into different persist epochs. Persist barriers ensure that all the persists before the barrier occur strictly before any persist after the barrier. Persists in the same epoch, however, are allowed to concurrently execute, potentially improving persist performance. A major disadvantage of relaxed persistency is the increased programming complexity; programmers must correctly annotate their code using the two types of barriers (i.e., memory and persist barriers).

With memory persistency, it is the responsibility of the underlying hardware to enforce the ordering constraints of NVRAM writes specified in the (annotated) code. Therefore, no extra code is required to explicitly flush appropriate cache lines to NVRAM, easing the programmer's burden.

Strict persistency significantly simplifies the NVWAL implementation shown in Algorithm 1 because all the cache-flush operations and persist barriers can be safely removed. However, we conjecture that strict persistency may degrade the performance of NVWAL because it enforces strict (but unnecessary) ordering constraints between persists when writing the log entries to NVRAM.

Relaxed persistency simplifies the NVWAL implementation shown in Algorithm 1 because all the cache-flush oper-

# of insertion per txn	1	2	4	8	16	32
# of cache line flushes	139.49	153.32	181.224	236.52	349.168	574.464

Table 1: Average number of cache line flushes per transaction for the experiments shown in Figure 5

ations can be safely removed. We expect that relaxed persistency, because it can dynamically reorder persist operations when copying the WAL frames from DRAM to NVRAM, will induce a level of performance for NVWAL higher than that possible when using strict persistency. Due to the unavailability of real hardware that can implement strict and relaxed persistency, we leave a performance evaluation of NVWAL under various memory persistency models to our future work.

5. Experiments

We implemented NVWAL in SQLite 3.7.11 and integrated it with an NVRAM heap manager - *Heapo* [16]. We first measure the performance of NVWAL using a *Tuna* NVRAM emulation board [2, 29].

Tuna is an NVRAM emulation board with Xilinx Zynq XC7Z020 ARM-FPGA SoC. The Tuna board consists of an ARM Cortex-A9 processor with L2 caches and FPGA programmable logic that controls the read/write latency of one of the two DRAMs in order to emulate NVRAM. The emulated NVRAM has a separate power switch for non-volatility emulation. The frequency of the emulated NVRAM is 400MHz (DDR3-800) while that of the volatile DRAM is 533MHz (DDR3-1066); the data width of the emulated NVRAM is 64 bits while that of DRAM is 32 bits. The size of the cache line is 32 bytes. The write latency of the emulated NVRAM can be adjusted between 400 nsec and 2000 nsec.

5.1 Overhead of Ordering Constraints

First, we quantify the overhead of enforcing the ordering constraints in SQLite write-ahead logging. In the configuration denoted as *E* (eager synchronization), we make SQLite write-ahead logging call `cache_line_flush()` system call and memory barrier immediately after each `memcpy()` function call per log entry as illustrated in Figure 4(b). In the other configuration, denoted as *L* (lazy synchronization), SQLite calls a batch of `cache_line_flush()` system calls for all the dirty log entries in a transaction right before its commit mark is stored, as illustrated in Figure 4(c).

We run the experiments on a Tuna NVRAM emulation board and set the write latency of the NVRAM to 500 nsec as in [37]. Figure 5 shows the benefits of lazy synchronization. As we insert more records into the database table in a single transaction, more dirty bytes are written to NVRAM and the time spent for memory writes increases. Table 1 shows how many cache lines are flushed per transaction (the number of called dccmvac instructions) with varying the number of insertions per transaction. When each transaction inserts a single data record, the time spent on `dccmvac`, `dmb`, and kernel mode switching is only 19.3 usec per transaction. Considering that the query execution time is 424 usec, this overhead is just 4.6% of the query execution time as shown in Figure 6. However, when a transaction inserts 32 data records, the query execution time is 5828 usec while the overhead of the write ordering constraints is 46.5 usec (only 0.8 % of the execution time). SQLite throughput is governed more by the computation performance than by the I/O performance, especially when a WAL file is located in fast NVRAM. Hence, improving the I/O performance by reordering the memory writes does not significantly affect the ratio of the ordering constraint overhead to the query execution time, as shown in Figure 6.

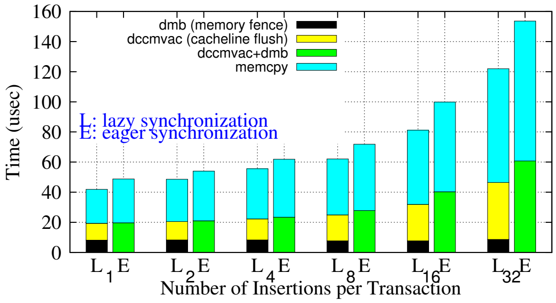


Figure 5: Quantifying the benefit of allowing processors to reorder memory writes

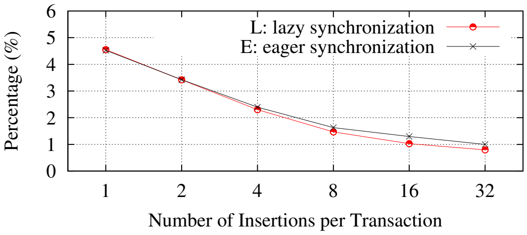
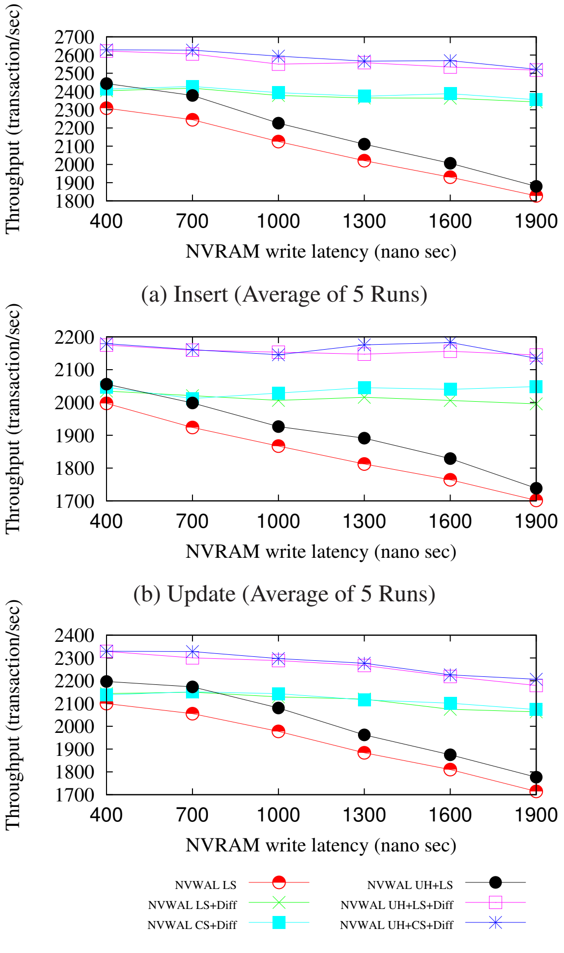


Figure 6: Proportion of ordering constraint overhead to query execution time

In eager synchronization, denoted as *E*, we do not allow reordering of memory writes. But, in lazy synchronization denoted as *L*, we call `memcpy()` for all dirty portions of the WAL frames in a transaction before NVWAL explicitly calls `dccmvac`. Therefore the dirty bytes in the L1 or L2 cache could have been already evicted. I.e., in lazy synchronization, the overhead of `dccmvac` is masked by the overhead of `memcpy()`. As in epoch persistency, the lazy synchronization decouples the volatile memory order from the persist order, which fits quite well with the database transaction concept.

# of operation per txn	1	2	4	8	16	32
Insert	4431.8	4874.2	5767.1	7536.6	11141.1	18350.0
Insert (Diff)	726.8	863.0	1139.7	1684.0	2776.0	4984.0
Delete	4890.6	5685.2	7274.4	10452.9	16842.7	22282.2
Delete (Diff)	1555.2	2236.8	3576.9	5863.0	9412.6	11452.9
Update	4096	4210.6	4440.0	4882.4	5832.7	7733.2
Update (Diff)	647.7	830.3	1186.5	1857.9	3191.5	5529.6

Table 2: Average number of bytes written to NVRAM



(a) Insert (Average of 5 Runs)

(b) Update (Average of 5 Runs)

(c) Delete (Average of 5 Runs)

Figure 7: Transaction throughput with varying latency of NVRAM (Average of 5 runs on Tuna)

The amounts of time spent on memcpy() in both schemes are similar in the experiments, as shown in Figure 5. However, in *E*, the dccmvac and dmb together perform up to *23% slower* than does dccmvac in *L* because *E* does not allow the reordering of memory writes. By separating memcpy() and dccmvac, write-ahead logging can eliminate about 2∼23% of the total overhead of enforcing persistency (19.3 ∼ 46.4 usec vs. 19.7 ∼ 60.7 usec). The overhead of dmb in *L* is negligible because dmb is called at most only four times in our implementation.

5.2 Differential Logging and I/O

Table 2 shows the I/O volume written to NVRAM with byte granularity differential logging and with legacy block granularity logging. For the insert, update, and delete operations, byte-granularity differential logging eliminates 73∼84%, 29∼85%, and 49∼69% of the unnecessary I/Os to NVRAM, respectively. In SQLite, each B-tree page appends a newly inserted set of data to the end of the used region. Thus, the size of a WAL log entry generated by byte-granularity differential logging is often small. But, for the update and delete operations, because it has to shift some log entries to avoid fragmentation issues, the write-ahead logging may touch a large portion of the B-tree page. Therefore, the insert transaction gets the most performance benefit from byte-granularity logging while the performance benefits for the update and delete transactions are moderate.

5.3 Transaction Throughput and NVRAM Latency

For the experiments shown in Figure 7, we measure the throughput of NVWAL with the SQLite benchmark app Mobibench [1] with varying the latency of NVRAM. Using Mobibench, we submit 1,000 transactions that insert, update, or delete a single 100-byte data record per transaction. In the experiments, we do not include the time for periodic checkpointing that sporadically flushes the logged WAL frames to the slow block device storage. It should be noted that checkpointing affects the performance of only one out of hundreds of transactions; this overhead is not relevant to the write latency of NVRAM.

The write latency of NVRAM is likely to be higher than the write latency of DRAM. For example, phase change memory, because it has access latencies in the hundreds of nanoseconds, is about 2∼5 times slower than DRAM [10, 24]. In the experiments, we vary the write latency of NVRAM from 400 nsec to 1900 nsec. Overall, as the write latency increases, the transaction throughput decreases in a linear fashion. However, the latency adjustment affects the throughput of some NVWAL schemes more severely than it does others. Due to the unavailability of persist barrier instruction, we simulate the persist barrier overhead by introducing a 1 usec delay using nop instructions. We also assume the PoC is the system main memory - DRAM and NVRAM. In most cases, the PoC is the system main memory and dccmvac flushes data to the main memory. However, ARM architecture does not prohibit the implementation of outer caches beyond the PoC. We believe the outer cache should be disabled when NVRAM is used. Otherwise, outer caches should be explicitly flushed before the persist barrier instruction is called.

NVWAL LS denotes the NVWAL scheme with *lazy synchronization*, which calls cache line flush instructions in a lazy manner without hurting the correctness of the database transactions as illustrated in Figure 4(c). *NVWAL LS* does not employ byte-granularity differential logging and user-level heap; it calls Heapo's `nvmalloc()` system call for every dirty WAL frame.

In *NVWAL LS+Diff*, SQLite performs lazy synchronization. Additionally, it adopts *byte-granularity differential logging*. When the B-tree node size is 4 Kbytes, *NVWAL LS* calls 128 cache line flush instructions, one memory barrier instruction, and one persist barrier instruction for each WAL frame. It then calls another cache line flush instruction, memory barrier instruction, and persist barrier instruction for a commit mark. However, in *NVWAL LS+Diff*, the average number of cache line flush instructions per transaction is reduced, and hence *NVWAL LS+Diff* consistently outperforms *NVWAL+LS*, yielding up to 28% higher throughput.

In *NVWAL CS+Diff*, NVWAL asynchronously flushes log entries without explicitly calling cache line flush instructions, as illustrated in Figure 4(d). In this variant of NVWAL, we call the cache line flush instruction and memory barrier only for a commit mark and checksum bytes. In this way, we minimize the overhead of the ordering constraints. However, the performance benefit comes with a potential inconsistency.

In *NVWAL UH+LS*, NVWAL employs the user-level heap, which reduces the number of calls to the expensive NVRAM heap manager's `nvmalloc()` function. In the experiments shown in Figure 7, we call Heapo's `nvmalloc()` and allocate 8Kbytes NVRAM block that can store two WAL frames. By saving the overhead of calling an expensive NVRAM heap manager's function, *NVWAL UH+LS* achieves a 6% performance gain over *NVWAL LS*.

In *NVWAL UH+LS+Diff*, NVWAL employs user-level heap, lazy synchronization, and byte-granularity differential logging. Interestingly, the performance of *NVWAL UH+LS+Diff* is comparable to that of *NVWAL UH+CS+Diff*, which uses an user-level heap, byte granularity differential logging, and checksum bytes. Because *NVWAL UH+CS+Diff* minimizes the number of bytes to be written to NVRAM and also minimizes the overhead of the `cache_line_flush()` system calls and memory barrier, it shows the highest transaction throughput. Considering that *NVWAL UH+CS+Diff* is vulnerable to the inconsistency problem that is inherent in probabilistic checksum bytes, *NVWAL UH+LS+Diff*'s comparable performance makes it a promising write-ahead logging scheme for NVRAM, because it does not compromise the correctness of database transactions.

As the write latency of NVRAM increases, the benefit of using the proposed schemes becomes more significant. When the write latency of NVRAM is set to 1942 nsec, *NVWAL UH+LS+Diff* yields throughput up to 37% higher than that of *NVWAL LS*.

5.4 The Real Numbers: Nexus 5

In the last set of experiments, we examine the performance of NVWAL using a commercially available smartphone - the Nexus 5, which has a 2.26 GHz Snapdragon 800 processor, 2 GB DDR memory and 16 GB SanDisk iNAND eMMC 4.51 SDIN8DE4 formatted with an EXT4 file system. As for the NVRAM emulation on Nexus 5, we assume that a specific address range of DRAM is NVRAM, i.e., that NVRAM is attached to the memory bus. NVRAM write latency is varied by inserting `nop` instructions. The stock SQLite WAL implementation does not use the emulated NVRAM but stores log pages in flash memory instead. We fix the CPU frequency to the maximum 2.26 GHz to reduce the standard deviation of the experiments. The Snapdragon 800 processor's cache line size is 64 bytes. We run Mobibench using SQLite 3.7.11 and Android 4.4 on Nexus 5.

Recent studies [17, 23, 30] have reported that SQLite on flash memory suffers from unexpected I/O volume and that SQLite write-ahead logging unnecessarily doubles the I/O traffic in the current implementation (SQLite 3.7.11). In order to fix these problems and compare the performance of SQLite WAL on flash memory against that on NVWAL in a fair way, we implemented two optimizations that help avoid the unnecessary I/O traffic and that significantly improve the performance of SQLite on flash memory.

For each dirty page, SQLite creates a 24-byte frame header that consists of page number, commit mark, checksum values, etc., and that appends the dirty page to it. Due to the additional 24-byte frame header of the WAL frames, each WAL frame becomes larger than the page size, which causes the WAL frames not to be aligned with the page boundaries. With such misaligned pages, a write operation for a single database page causes at least two pages to be written to the block device storage. In order to resolve the misaligned WAL frame problem, we modified SQLite's B-tree split algorithm so that it splits an overflow page early and the last 24 bytes of all B-tree pages are not used. By doing so, 24 bytes of WAL frame header and WAL frame can be merged and stored in a single page of a WAL file. We implemented the same split algorithm for NVWAL.

Another ad-hoc improvement we made on SQLite WAL is *pre-allocation of WAL log pages* as in WALDIO [30]. For each dirty page, a transaction calls `write()` that appends one or two pages to the end of a log file. As the appended pages increase the size of the WAL log file, the increased file size must be updated in an inode. Because the writes in WAL mode are all sequential and because the size of the log file keeps increasing until checkpointing truncates it, pre-allocating multiple pages once will help the next transactions write log frames without increasing the file size. The only negative aspect of this pre-allocation is that it may waste several disk pages if there is no next transaction. In terms of EXT4 journal overhead, allocating two new pages causes overhead slightly higher than that induced by allocating only one page. However, that overhead is much smaller than the overhead for allocating another page later. The size of the pre-allocated pages can be fixed at a specific number based

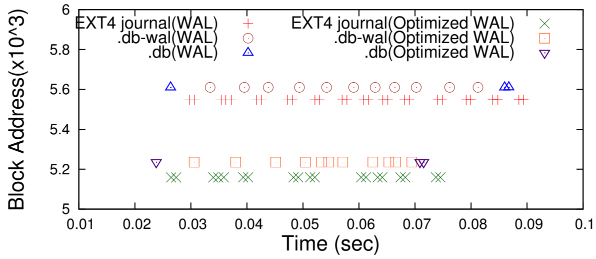


Figure 8: Block Trace of SQLite Insert Transaction with Optimizations

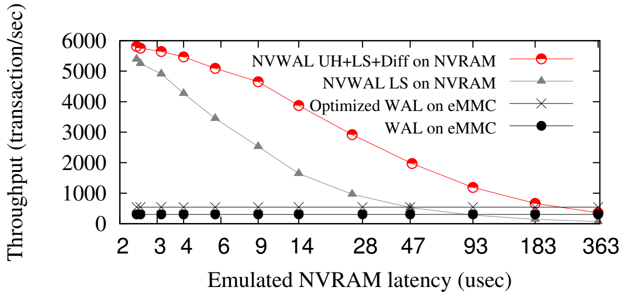


Figure 9: Transaction Throughput of NVWAL on Emulated NVRAM vs. Optimized WAL on eMMC Memory (Average of 5 Runs)

on the database access patterns or the size can be doubled every time the pre-allocated pages fill up.

Figure 8 shows a block I/O trace of 10 transactions in stock SQLite WAL mode and another block I/O trace of 10 transactions in our optimized WAL mode. For a single insert transaction, one block (4KB) is written to the `.db-wal` file but two blocks(16KB, 4KB) are written to the EXT4 journal to update the metadata of the log file in the ordered-mode EXT4 journal. In our optimized WAL mode, SQLite pre-allocates 8 pages when a transaction writes a log frame for the first time, even if the system needs just one page. If another transaction needs to write log frames while there is no available pre-allocated page, we double the number of pages to be pre-allocated each time and SQLite allocates 16 new pages to the log file. Compared to the block trace of the WAL mode, allocating multiple log pages in advance reduces EXT4 journal accesses by 40% (172 KB vs. 284 KB), and the batch execution time in our optimized WAL mode decreases from 90 msec to 74 msec.

In Figure 9, we measure the transaction throughput of NVWAL on Nexus 5. We run 1000 insert transactions with an empty SQLite database table. Each transaction inserts a single 100-byte record into the table. The NVRAM write latency on the Nexus 5 is varied by adding `nop` instructions after each `clflush` instruction, which is also used to emulate the NVRAM write latency in [15]. We set the checkpointing interval to 1000 dirty WAL frames, which is the default setting in SQLite. As for the sporadic checkpointing overhead, we amortize the overhead across 1000 transactions.

Our optimized WAL on flash memory yields 541 transactions/sec while *NVWAL LS* and *NVWAL UH+LS+Diff* on NVRAM (DRAM with 2 usec write latency) exhibit 5393 and 5812 transactions/sec, respectively. As we increase the write latency, the throughput of NVWAL decreases. When the write latency becomes 47 usec, *NVWAL LS* on the emulated NVRAM shows throughput similar to that of WAL on flash memory. When the write latency is set to 230 usec, *NVWAL UH+LS+Diff* also shows performance similar to that of WAL on flash memory. We need to stress that an NVRAM write latency of 230 usec is very conservative.

The experiments on the Nexus 5 are not precisely identical to the experiments on the Tuna board because the former do not include the overhead of checkpoint operation, whereas the latter include the checkpointing overhead. Hence, the experiments on the Nexus 5 stand for the sustained throughput, while the experiments on the Tuna stand for the peak throughput.

6. Related Work

Condit et al. [10] designed a file system *BPFS* and a hardware architecture for persistent, byte-addressable memory. BPFS proposes a new *epoch barrier* instruction that specifies an ordering of groups of persists. BPFS uses short-circuit shadow paging, which provides atomic, fine-grained updates to persistent storage. However, *epoch barrier* instruction requires a modification of the memory hierarchy because it considers cache-bypassing writes, and because cache flushes cause inefficiency.

Fang et al. [12] proposed to write log records directly into storage class memory (SCM), which provides an *epoch barrier* without caching them in volatile DRAM. In their logging scheme, log records stored in SCM are periodically archived to a WAL file in block device storage. Unlike our NVWAL, their logging scheme requires log records to be stored twice - once in NVRAM and once in block device storage before they are applied to the database file.

SCMFS proposed by Wu et al. [46] is another file system designed for non-volatile memory; it uses only `clflush` and `mfence` instructions without any modification to the hardware; however, it is unclear if this design guarantees consistency if the system crashes.

With non-volatile memory, memory write errors from application bugs can cause serious and permanent system corruption. Moraru et al [35] proposed *NVMalloc* - a memory allocator for NVRAM that prevents memory wear-out and increases robustness against erroneous memory writes. *NV-Heaps* [9], Mnemosyne [43], *Heapo* [16], and Atlas [7] are persistent object management systems for NVRAM. These works are complementary to our work as NVWAL can utilize persistent and byte-addressable heap managers or file systems.

In the future, when NVRAM and a good implementation of the transactional memory support such as Mnemosyne [43] become available, it may be possible to eliminate some use cases of SQLite because some application developers use SQLite as a failure atomic persistent storage. However, RDBMS provides the same level of failure atomic persistent storage, with the added benefit of using declarative query language (SQL), pre-implemented relational operators, indexing, and query optimization, to name just a few of them. Hence, we expect that NVWAL and persistent heap will be complementary to each other, rather than in competition.

In Android smartphones, due to the overhead of journaling activity, one of the major performance bottlenecks has been shown to be the storage of the device [17, 21, 27]. Recently, a fair amount of work has been done to leverage the non-volatility of NVRAM to resolve the performance and scalability issues of database logging [5, 8, 12, 15, 19, 20, 22, 25, 32, 39–42, 44, 45]. Kim et al. [23] proposed a version-based B-tree for flash memory. The version-based B-tree (*CDDS*) has also been studied for NVRAM by Venkataraman et al. [41]. CDDS allows atomic updates on NVRAM without requiring logging. *NV-tree* is another B-tree for NVRAM, specifically designed to achieve data consistency on NVRAM while reducing the number of cache line flush operations.

NV-Logging proposed by Huang et al. [15], proposes a per-transaction logging method for OLTP systems using NVRAM; this allows us to avoid the bottleneck of centralized log buffers. Unlike NVWAL, NV-Logging [15] calls `clflush` immediately after every `memcpy()` operation. Huang et al. [15] compared their per-transaction logging against that of an *NV-Disk* [20, 22] - NVRAM accessed via standard file I/O interface; they showed that replacing the disk or flash memory with NVRAM without redesigning the file system cache and I/O interface can cause the system to suffers from high overhead [15]. Our NVWAL is different from NV-Logging in that NV-Logging focuses on scalability issues of processing concurrent queries, while SQLite does not allow concurrent transactions.

The related work most similar to ours is SQLite/PPL, recently proposed by Oh et al. [36]. They proposed a per-page logging scheme for SQLite; this system stores the disk-based WAL frames in PCM memory, which takes no account of byte-addressability, reordering of memory writes, or the overhead of managing a persistent heap.

Lee et al. [25] proposed to merge the buffer cache and the journal for byte-addressable NVRAM by converting the buffer cache blocks to journal logs simply by changing the status of the blocks to a frozen state. The method of *Blurred Persistence*, proposed by Lu et al. [32], blurs the boundary between volatility and persistency by allowing processors to persist uncommitted data in NVRAM and by detecting the uncommitted data later via *execution in log* if necessary. Our work is different from theirs in that NVWAL targets database systems that retain DRAM, not NVRAM, as a volatile buffer cache.

Pelley et al. [37] conducted a memory trace analysis and showed that replacing a disk with NVRAM, which supports a *persist barrier*, can achieve significantly higher throughput than that of disks. To further improve the performance of OLTP, they proposed a *group commit* protocol that persists the committed transactions in batches in order to reduce the number of required barriers. Our method of transaction-aware lazy synchronization is different from their idea of group commit in that they assumed that the ordering of memory writes would be enforced through the *persist barrier*, which does not exist yet, while we evaluated SQLite WAL on real hardware an emulated NVRAM board with ARM v7 instructions and no modifications to the processor design.

7. Conclusions

Emerging non-volatile memory devices are expected to substitute slow block device storage so as to persistently store frequently accessed data. In this work, we design and implement write-ahead-logging on NVRAM (NVWAL) and show that NVWAL can minimize the overhead of managing and synchronizing log entries on NVRAM via i) transaction-aware lazy synchronization, ii) a user-level NVRAM block manager, and iii) byte-granularity differential logging.

Through our extensive performance evaluation, we show that database logging can be optimized to be insensitive to NVRAM latency, and that the overhead of guaranteeing the failure atomicity is almost free. NVWAL with the optimizations exhibits transaction throughput up to 37% higher than that of non-optimized WAL on NVRAM; when NVRAM latency is smaller than 2 usec, NVWAL achieves at least 10x higher than that of legacy WAL on flash memory.

We can draw three lessons from this study. First, employing an NVRAM in database logging is not an option but a necessity. Second, the aggregate overhead of the persist barrier instructions is insignificant from the application's point of view. Third, NVRAM latency is barely propagated to SQLite performance. NVWAL significantly decreases the logging overhead in SQLite and makes the workload further CPU bound. As a result, SQLite performance becomes relatively insensitive to NVRAM latency.

8. Acknowledgments

We would like to thank our shepherd Shankar Pasupathy and the anonymous reviewers for their suggestions on early drafts of this paper. This research was supported by MKE/KEIT (No.10041608, Embedded System Software for New Memory based Smart Devices). The corresponding author of this paper is Beomseok Nam.

References

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

[2] OpenNVRAM. *http://opennvram.org/*.

[3] Sqlite. *http://www.sqlite.org/*.

[4] S. Agarwal, R. Garg, M. S. Gupta, , and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*, 2004.

[5] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722. ACM, 2015.

[6] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Implications of cpu caching on byte-addressable non-volatile memory programming. *http://www.hpl.hp.com/techreports/2012/HPL-2012-236.pdf*, 2012.

[7] D. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceeding of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 433–452, 2014.

[8] A. Chatzistergiou, M. Cintra, and S. D. Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile datastructures. *Proceedings of the VLDB Endowment*, 8(5):497–508, 2015.

[9] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. 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.

[10] 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.

[11] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys)*, pages 15:1–15:15, 2014.

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

[13] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini. Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers. In *Proceedings of the ACM/IEEE SC2005 Conference*, 2005.

[14] G. Graefe. A survey of B-tree logging and recovery techniques. *ACM Transactions on Database Systems*, 37(1), Feb. 2012.

[15] J. Huang, K. Schwan, and M. K. Qureshi. Nvram-aware logging in transaction systems. *Proceedings of the VLDB Endowment*, 8(4), 2014.

[16] T. Hwang, J. Jung, and Y. Won. Heapo: Heap-based persistent object store. *ACM Transactions on Storage (TOS)*, 11(1), 2014.

[17] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O stack optimization for smartphones. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2013.

[18] J. Jung and Y. Won. nvramdisk: A transactional block devie driver for non-volatile ram. *IEEE Transactions on Computers*, http://dx.doi.org/10.1109/TC.2015.2428708, 2015.

[19] J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon. Frash: Exploiting storage class memory in hybrid file system for hierarchical storage. *ACM Transactions on Storage (TOS)*, 6(1):3, 2010.

[20] D. Kim, E. Lee, S. Ahn, and H. Bahn. Improving the storage performance of smartphones through journaling in non-volatile memory. *Consumer Electronics, IEEE Transactions on*, 59(3):556–561, 2013.

[21] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2013.

[22] J. Kim, C. Min, and Y. I. Eom. Reducing Excessive Journaling Overhead with Small-Sized NVRAM for Mobile Devices. *IEEE Transactions on Consumer Electronics*, 6(2), June 2014.

[23] W.-H. Kim, B. Nam, D. Park, and Y. Won. Resolving journling 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.

[24] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, 2009.

[25] E. Lee, H. Bahn, and S. 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.

[26] J. Lee, K. Kim, and S. Cha. Differential logging: A commutative and associative logging scheme for highly parallel main memory database. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, 2001.

[27] K. Lee and Y. Won. Smart layers and dumb result: Io characterization of an android-based smartphone. In *Proceedings of the 12th International Conference on Embedded Software (EMSOFT 2012)*, 2012.

[28] S.-W. Lee and B. Moon. Design of flash-based dbms: An in-page logging approach. In *Proceedings of 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2007.

[29] T. Lee, D. Kim, H. Park, and S. Yoo. Fpga-based prototyping systems for emerging memory technologies. In *Proceedings of the 25th IEEE International Symposium on Rapid System Prototyping (RSP)*, 2014.

[30] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won. Waldio: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proceedings of the 2015 USENIX Anual Technical Conference (ATC)*, 2015.

[31] M. Li and P. P. C. Lee. Toward i/o-efficient protection against silent data corruptions in raid arrays. In *Proceedings of the*

30th International Conference on Massive Storage Systems and Technology (MSST), 2014.

[32] Y. Lu, J. Shu, and L. Sun. Blurred persistence in transactional persistent memory. In *Proceedings of the 31st International Conference on Massive Storage Systems and Technology (MSST)*, 2015.

[33] H. Luo, L. Tian, and H. Jiang. qNVRAM: quasi non-volatile ram for low overhead persistency enforcement in smartphones. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[34] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale research letters*, 9(1):1–33, 2014.

[35] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, N. Binkert, and P. 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.

[36] G. Oh, S. Kim, S.-W. Lee, and B. Moon. Sqlite optimization with phase change memory for mobile applications. *Proceedings of the VLDB Endowment (PVLDB)*, 8(12):1454–1465, 2015.

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

[38] K. Shen, S. Park, and M. Zhu. Journaling of journal is (almost) free. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2014.

[39] M. Son, S. Lee, K. Kim, S. Yoo, and S. Lee. A small non-volatile write buffer to reduce storage writes in smartphones. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 713–718, San Jose, CA, USA, 2015. EDA Consortium. ISBN 978-3-9815370-4-8.

[40] C. A. N. Soules, G. R. Goodson, J. D. Strunk, and G. 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.

[41] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *9th USENIX conference on File and Storage Technologies (FAST)*, 2011.

[42] S. D. Viglas. Data management in non-volatile memory. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1707–1711. ACM, 2015.

[43] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[44] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, 2014.

[45] Q. Wei, J. Chen, and C. Chen. Accelerating file system metadata access with byte-addressable nonvolatile memory. *ACM Transactions on Storage (TOS)*, 11(3):12, 2015.

[46] X. Wu and A. L. N. Reddy. SCMFS: A file system for storage class memory. In *Proceedings of the ACM/IEEE SC2011 Conference*, 2011.

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

[48] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 449–464, Oct. 2014.