Fast Transaction Logging for Smartphones

Hao Luo
University of Nebraska, Lincoln

Hong Jiang
University of Texas Arlington

Zhichao Yan
University of Texas Arlington

Yaodong Yang
University of Nebraska, Lincoln

Abstract

Mobile databases and key-value stores provide consistency and durability through write-ahead logging. The traditional logging scheme appends the log records to the end of the log file and flushes the records to durable storage using `fsync()`. Due to the large block size of the underlying file system and the Journaling of Journal anomaly, the logging latency becomes the main bottleneck of the mobile databases. Our experimental results indicate that the logging latency accounts for more than 90% of the overall insert latency on a Samsung Galaxy S4 smartphone. Moreover, we observe a significant write amplification (up to 122×) induced by the traditional logging scheme.

In this paper we present xLog, a fast transaction logging service leveraging qNVRAM, a nearly non-volatile memory for mobile devices. From our experimental results, xLog logs up to 77× faster than the traditional logging scheme, and speeds up the LevelDB `Put` operation by up to 10.7×. Moreover, xLog drastically reduces the write amplification of the traditional logging scheme, from 122× to less than 1.6×.

1 Introduction

The smartphones and tablets have become ubiquitous in the last five years. The databases in mobile devices (smartphones and tablets) have become a crucial part of data management in such mobile systems. The SQLite database serves as a persistent storage layer in the Android system. LevelDB [1], another NoSQL key-value database that only supports basic operations like `Get()`, `Put()` and `Delete()`, has been widely used by various applications [2]. Mobile databases employ write-ahead logging to ensure data persistency (atomicity, consistency and durability). Transaction logs are flushed to durable storage at commit time to prevent data loss. The transaction logging, however, is the main roadblock in achieving fast response time of write transactions due to the wimpy storage device in smartphones and the inefficiency of the Android I/O stack induced by the Journaling of Journal (JOJ) anomaly [5]. A recent study [4] points out that the JOJ anomaly, which refers to the double-journaling phenomenon in which the file system is journaling the database journal activities, drastically slows down the mobile databases.

Several solutions have been proposed to resolve the JOJ anomaly and improve the performance. One [4] is to optimize the Android I/O stack using a combination of different techniques, including log-structured file system, external journaling and pooling-based I/O. Another solution [5] is to integrate the recovery information into the database file itself so that the database journal is omitted. Shen et al. [9] argue that the JOJ is almost free through single-I/O data journaling in Ext4 file system. Nevertheless, while these solutions have been shown to boost the database transaction throughput, the inevitable data flushes to the flash storage upon transaction commit still cause a particularly long logging latency. Our previous work [6] proposed a new design leveraging the battery-backed nature of modern smartphones to enable the data in DRAM to survive almost all the failure conditions, which can be used to boost the performance of mobile databases.

In this paper, we propose xLog, a fast transaction logging service in Android smartphones. xLog uses qNVRAM as a persistent buffer to coalesce the small log records and significantly reduce the latency of transaction logging in SQLite and LevelDB. Our experimental results based on a Samsung Galaxy S4 smartphone show that xLog speeds up the transaction logging by up to 77× and the overall performance of the LevelDB `Put` operation by up to 10.7×, while cutting the write amplification from 122× to less than 1.6×.

2 Background and Motivation

2.1 Transaction Logging Overhead in Mobile Databases

The gold standard for transaction logging is ARIES [8], which uses fine-grained, record-oriented write-ahead logging (WAL) to recover the database. The ARIES-style physiological logging combines undo and redo logging: undo logging in the conventional ARIES systems

logs the description of the operation and redo logging logs the after-image of the database pages. Command logging [7], as an alternative to the ARIES-style logging, writes the transaction's logic (such as SQL query statements) to the write-ahead log. Mobile databases use variants of these two logging mechanisms.

SQLite's WAL uses value logging, which is different from the standard ARIES-style physiological WAL. It logs only the modified database pages, called frames in the SQLite WAL file, upon transaction commit. The log record for each transaction in the SQLite WAL file consists of multiple frames, of which the last frame serves as a commit marker. Upon transaction commit, the log record is flushed to eMMC flash storage using the `fsync()` / `fdatasync()` system calls. In Android smartphones, the default page size of SQLite databases is 4KB, which is the same as the block size of the underlying Ext4 file system.

The logging in LevelDB is somewhat similar to command logging. For the `Put()` operation, LevelDB logs the operation type (*kTypeValue*), the key and value; for the `Delete()` operation, LevelDB writes the operation type (*kTypeDeletion*) and the key to the log file. Generally, command logging will write substantially fewer bytes per transaction than logging the modified database pages, making the former a potentially much better performer than the latter.

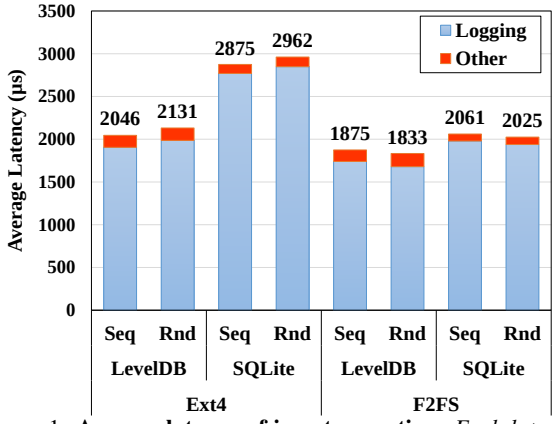

Figure 1: **Average latency of insert operation.** *Each latency value is the average of 10 repetitions with a $<$ 5% standard deviation.*

The transaction logging has imposed a great deal of overhead on mobile database systems due to the wimpy storage device and the inefficiency of the Android I/O stack. To assess the overhead of logging in Android smartphones, we run benchmark tests against SQLite and LevelDB on a Samsung Galaxy S4 smartphone. SQLite is configured to use write-ahead logging, and LevelDB is configured in the synchronous write mode. In each run of the benchmark test, 1000 records, each of which consists of an integer key and a 100-byte string value, are inserted to the SQLite or LevelDB in sequential or random key order. The benchmark tests run on the Ext4 and F2FS file systems respectively.

A breakdown view of the measured insert latency from the tests is shown in Figure 1 and the corresponding log file size and block-level I/O statistics are shown in Table 1. The *Actual Writes* refers to the amount of data writen to log file and total writes to eMMC storage device. As shown in the figure, the logging latency accounts for more than 90% of the average insert latency. The bottleneck lies in the `fsync()` / `fdatasync()` system calls. From Table 1 we can see that the log file of SQLite is more than 20$\times$ larger than that of LevelDB. For each of the insert transaction, SQLite will modify 2 to 3 database pages, each of which is 4KB, and write them to the log files. LevelDB, however, composes a much smaller log record of approximately 130KB (4-byte key + 100-byte value + record header) for the same operation.

Nevertheless, the gap between logging latency in LevelDB and SQLite is not as big as the log file sizes would suggest. On the Ext4 file system, command logging (LevelDB) is only 38% faster than the value logging (SQLite); on the F2FS file system, the gap shrinks to only 12%. This counter-intuitive result is due to the fact that the underlying file system and flash devices use large block size (usually 4KB). Whenever the log record is appended to the end of the log file, the data are flushed at the file system block boundaries. In our benchmark test, a 4KB block can hold about 30 LevelDB log records, thus for each of the 30 `Put()` operations the same block will be flushed to the storage devices 30 times. This results in drastically more data written to the flash storage than the size of log file. In the LevelDB benchmark test, the amount of data written to the log file is 30.8$\times$ the size of the log file. This issue also exists in SQLite when the log frame is not aligned with the file system block size. Even worse, more data are written to file system metadata or journals to ensure file system persistency. The F2FS file system exhibits a total data write amplification of up to 60.7$\times$; in the Ext4 file system, due to the Journaling of Journal anomaly [4], the write amplification of total data written is up to 122$\times$.

Another interesting observation from the benchmark test is that the log-structured file system (F2FS) does not significantly help with either the logging latency or the write amplification. The sequential and random insert latencies of LevelDB on the F2FS file system are only 12% and 18% shorter than those on the Ext4 file system, while these latencies of SQLite on F2FS are 28% and 31% shorter than those on the Ext4 file system. The reason behind this is that the inevitable `fsync()` / `fdatasync()` system calls prevent the transaction from

			Log file size (KB)	Actual Writes (KB)		Write Amplification	
				Log file	Total	$\frac{\text{Log file writes}}{\text{Log file size}}$	$\frac{\text{Total writes}}{\text{Log file size}}$
Ext4	**LevelDB**	**Seq**	134	4132	16420	30.8	122.5
		Rnd	134	4132	16428	30.8	122.6
	SQLite	**Seq**	2722	6720	23796	2.5	8.7
		Rnd	3028	7028	24656	2.3	8.1
F2FS	**LevelDB**	**Seq**	134	4132	8136	30.8	60.7
		Rnd	134	4132	8136	30.8	60.7
	SQLite	**Seq**	2722	6720	10764	2.4	4.0
		Rnd	3028	7020	11100	2.3	3.7

Table 1: **Log file size and block I/O statistics in the benchmark test.**

committing more quickly due to the non-sequential I/O pattern and the high write amplification.

2.2 qNVRAM Overview

qNVRAM [6] is a nearly non-volatile memory in smartphones that takes advantage of the "battery-backed" nature of mobile devices to make data in qNVRAM non-volatile under almost all the failure conditions. There are four different failure modes in smartphones, namely, (1) application crash, (2) application hang, (3) self-reboot, and (4) system freeze. qNVRAM manages a piece of the physical memory at fixed location, and maps the physical memory into application's address space upon application's request. When the application restarts after crash or getting killed by the user (i.e., failure mode (1) or (2)), the qNVRAM that the application allocates is remapped into its address space so that the application can recover from the qNVRAM; when the self-reboot (i.e., failure mode(3)) happens, the data in the physical memory is not lost since the DRAM in smartphone does not lose power thus data in qNVRAM can be retrieved; when the user performs hard reset (i.e., failure mode (4)), qNVRAM will flush the data to flash storage so that next time the smartphone restarts the data will be reloaded to the qNVRAM. As long as the battery is not physically pulled out, data in qVNRAM can be considered non-volatile. Given that the batteries in more and more smartphones and tablets are made non-removable, the qNVRAM can be considered non-volatile for practical purposes.

3 xLog Design

The architecture of xLog is shown in Figure 2. xLog runs as a system service process in the Android platform and manages a piece of qNVRAM as a flush buffer to coalesce the writes to the log files. Application processes communicate with xLog through Android Binder interprocess communication (IPC). A small header with the information about the log record, including a universal sequence number and checksum, is added to each record

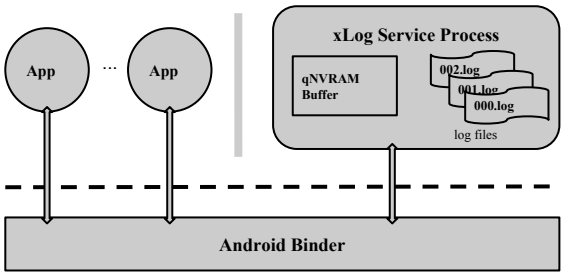

Figure 2: **Architecture of xLog.**

for recovery. qNVRAM is used as a ring buffer, and the offset of the buffer head and the size of valid data in the ring buffer are stored in the first eight bytes of qNVRAM, of which each is a 32-bit integer. These two variables are the key to recovering the log records in the ring buffer after a crash. Updating a 32-bit variable requires a single `store` instruction, which can be considered atomic [10].

xLog has a logger thread that services the logging requests from applications, and a flush worker thread that periodically flushes the log records in the ring buffer to the log files. All incoming requests are serialized by the logger thread and inserted to the ring buffer one by one (with corresponding CPU caches flushed to qNVRAM). During xLog's recovery, the log records in the log files and the ring buffer will be concatenated using the universal sequence number.

xLog logs for all applications that need transaction logging to persist data. The reason for choosing such a centralized design is two-fold. First of all, the burst mode is very common in mobile devices [3], and the bursty I/Os are usually induced by a single application. Since the mobile databases are embedded in the application's own process, every application can only allocate a small piece of qNVRAM as a flush buffer due to the overall qNVRAM size limitation. However, the qNVRAM in xLog can be made much bigger to better handle bursty I/Os. On the other hand, the centralized logging design helps improve the efficiency and qNVRAM utilization. The background applications are subject to being killed by `lowmemkiller`, thus a death-notification mechanism is needed to ensure that the qNVRAM allocated by the

application is reclaimed after the application process is dead. xLog, however, does not have such an issue: the xLog service process is always running and will restart immediately if it is terminated unexpectedly.

xLog manages a set of log files, one for each of the clients. Using separate log files for a different logging clients, instead of a single log file for all clients, simplifies the design and provides a bounded recovery time, although it might require more `fsync() / fdatasync()` system calls when flushing the qNVRAM buffer. Considering the recovery scenario where one application that uses the xLog service restarts after a crash, it will request the log records from xLog since the last checkpoint and replay them to recover from the inconsistent state. To fetch the log records for this application, xLog will only need to replay the log file corresponding to this application, instead of replaying a single large log file associated with all applications.

4 Preliminary Results

4.1 Micro-benchmarks

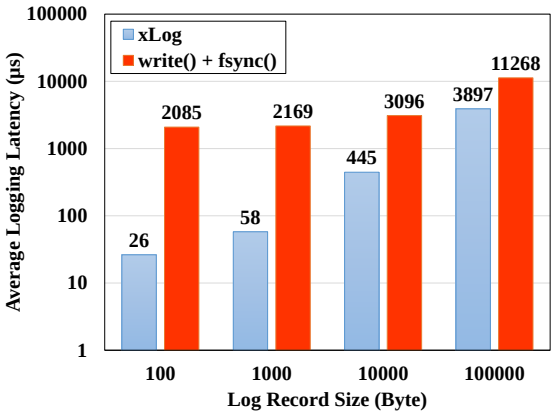

Figure 3: **Micro-benchmark performance of xLog.** *This figure shows the logging latency of native xLog. The latency is plotted in the logarithmic scale, and the value is shown on the top of each bar.*

We run a set of micro-benchmark tests on a Samsung Galaxy S4 smartphone to evaluate the raw performance of xLog. The xLog service process in the micro-benchmark allocates 10MB qNVRAM as the ring buffer. The log files are stored in a separate partition, formatted as an Ext4 file system. A total amount of 100MB log records is logged using xLog or the traditional logging scheme, which writes log records directly to the log file using `write() + fsync()`. The size of a single log record varies from 100 bytes to 100 kbytes. We measure the logging latency of native xLog inside the service process without involving the Android Binder IPC.

The average latency of both the xLog and baseline logging schemes is shown in Figure 3. From the figure we can see that, when the log record is small in size, xLog is significantly superior to the traditional logging method: the average logging latency of a 100-byte record in xLog is only 1.3% of that of the traditional scheme. As the size of the log record increases, the gap between the two methods narrows because the write amplification decreases. Yet, xLog still yields much better performance than `write() + fsync()`: xLog's logging performance is 37×, 7× and 3× faster than that of the baseline scheme using `write() + fsync()` when the log record size increases to 1KB, 10KB and 100KB respectively.

Note that in the micro-benchmark, there is no interval time between two consecutive logging requests. Therefore, the flush worker thread cannot keep pace with the logger thread: the logger thread needs to wait for the flush worker thread to make room in the ring buffer for the new log record. In real world scenarios, the I/O requests arrive in bursty patterns, which leaves plenty of time for xLog to flush the log records to log files before the next I/O burst comes. Thus the logging latency is expected to be much smaller even when the log record is big as long as the qNVRAM buffer is large enough to handle the I/O burst.

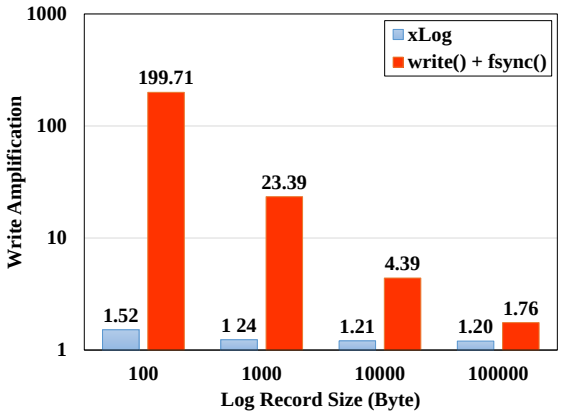

Figure 4: **Write amplification of xLog.** *The figure shows the write amplification in the micro-benchmark test. The write amplification is calculated by dividing the total amount data written to flash storage by the actual log size, and is shown in the logarithmic scale.*

xLog also addresses the write amplification issue by batching the small writes into a large I/O. The results from the micro-benchmark tests are shown in Figure 4. When the log record is small in size (100 byte per record), the write amplification induced by the traditional logging scheme (`write() + fsync()`) is exceedingly high (199.7×). xLog, however, exhibits a write amplification of only 1.5×. Most of the overhead comes from

the record header added by xLog. This overhead is amortized as the record size increases.

4.2 Macro-benchmark: A LevelDB Example

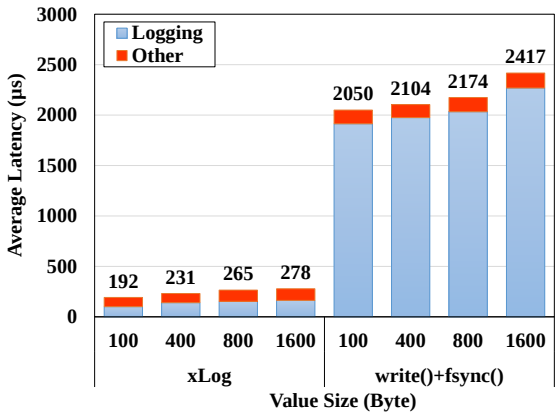

Figure 5: **LevelDB performance with xLog and traditional logging scheme.** *The figure shows the performance of LevelDB using different logging schemes in the benchmark test. In the test, 1000 key-value pairs are randomly inserted to the LevelDB in the synchronous mode. The average insert latency is shown on the top of each bar.*

To evaluate the performance benefits of xLog for mobile databases, we modify the LogWriter in LevelDB so that it writes log records using the xLog service instead of appending them to log files. The performance of LevelDB using the xLog and baseline traditional logging is shown in Figure 5. We vary the value size of the key-value pairs in LevelDB to evaluate the performance with different log record size. From the figure, when the value size is small (100 bytes), xLog speeds up the overall performance of LevelDB by 10.7×. As the value size increases, the speedup decreases a little as the logging latency increases. Nevertheless, xLog still boosts the LevelDB performance by 9.1×, 8.2× and 8.6× when the value size is 400 bytes, 800 bytes and 1600 bytes respectively.

Compared to the latency of the native xLog from the micro-benchmark, xLog logging via the Android Binder IPC induces significant latency. When the value size is 100 bytes, the overhead of the Android Binder IPC call is about 100 μs for a two-way round trip, excluding the logging latency of native xLog. A Binder IPC call will involve multiple context switches and task switches/wakeups as both the client (LevelDB) and server (xLog) fetch/pass data from/to the Binder kernel driver. The IPC call becomes the major bottleneck in the overall logging latency considering that the native xLog takes only 20 μs to log a 100-byte record, thus diminishing the performance benefits from xLog. Nonetheless, the logging latency of xLog over Binder IPC is still one order of magnitude lower than that of writing to the log files.

5 Discussion and Future Work

In this paper we present xLog, a fast transaction logging service that uses qNVRAM as a buffer, for Android smartphones. Our previous work has demonstrated that a qNVRAM-based persistent buffer cache can significantly boost the performance of the SQLite database. The xLog, however, is a more general solution to the overhead of enforcing persistence. The persistent buffer cache can only be applied to applications that use paged storage and perform in-place update. The LevelDB, for example, is not able to benefit from persistent page cache because it employs a log structured merge tree (LSM-Tree), and all new records are appended to a sorted table file. xLog, however, can be used to speed up both SQLite and LevelDB. xLog can be used to accelerate existing databases, as well as building new types of persistent storage. With xLog, we can build a high performance persistent in-memory database and do not need to worry about the overhead of enforcing persistence. We plan to explore the performance benefits of xLog in more applications, including the conventional databases (e.g., SQLite) and the in-memory database that usually achieve persistency through a combination of checkpointing and logging.

References

[1] LevelDB. https://github.com/google/leveldb.

[2] AGRAWAL, N., AND ET AL. Mobile data sync in a blink. In *USENIX HotStorage* (2013).

[3] JEONG, D., AND ET AL. Boosting quasi-asynchronous i/o for better responsiveness in mobile devices. In *USENIX FAST* (2015).

[4] JEONG, S., AND ET AL. I/O stack optimization for smartphones. In *USENIX ATC* (2013).

[5] KIM, W.-H., AND ET AL. Resolving journaling of journal anomaly in android i/o: Multi-version b-tree with lazy split. In *USENIX FAST* (2014).

[6] LUO, H., AND ET AL. qnvram: quasi non-volatile ram for low overhead persistency enforcement in smartphones. In *USENIX HotStorage* (2014).

[7] MALVIYA, N., AND ET AL. Rethinking main memory oltp recovery. In *IEEE ICDE* (2014).

[8] MOHAN, C., AND ET AL. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems* (1992).

[9] SHEN, K., AND ET AL. Journaling of journal is (almost) free. In *USENIX FAST* (2014).

[10] VOLOS, H., AND ET AL. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News* (2011).