qNVRAM: quasi Non-Volatile RAM for Low Overhead Persistency Enforcement in Smartphones

Hao Luo, Lei Tian and Hong Jiang
University of Nebraska, Lincoln

Abstract

The persistent storage options in smartphones employ journaling or double-write to enforce atomicity, consistency and durability, which introduces significant overhead to system performance. Our in-depth examination of the issue leads us to believe that much of the overhead would be unnecessary if we rethink the volatility of memory considering the battery-backed characteristics of DRAM in modern-day smartphones. With this rethinking, we propose quasi Non-Volatile Memory (qNVRAM), a new design that makes the DRAM in smartphones quasi non-volatile, to help remove the performance overhead of enforcing persistency. We assess the feasibility and effectiveness of our design by implementing a persistent page cache in SQLite. Our evaluation on a real Android smartphone shows that qNVRAM speeds up the insert, update and delete transactions by up to 16.33×, 15.86× and 15.76× respectively.

1 Introduction

Over the past decade, mobile devices, such as smartphones and tablets, have become ubiquitous. The latest smartphones are equipped with multi-core processors, large-capacity DRAM and flash storage. A recent study [6] points out that the I/O performance dominates the overall application performance in smartphones. In consideration of data consistency and integrity, Android applications rely on SQLite, the Shared Preference key-value store or the file system API to save persistent data in the local flash storage. In practice, all the three options employ journaling or file-level double-write, which adds significant overhead to and thus substantially degrade system performance by increasing I/O traffic with extra data written to flash storage. The SQLite database employs rollback journal or write ahead log to track the changes to the database table files. The Shared Preference (stored as an `xml` file in the file system) and some applications that store important data in files use file-level double-write to avoid data loss when modifying important files. The whole file, instead of the modified parts, will be written to a temporary file that will subsequently be renamed. Moreover, the underlying EXT4 file system uses metadata journaling to ensure data integrity. Another recent study shows that the performance of the SQLite suffers from the anomaly of Journaling of Journals (JOJ), which refers to the double-journaling phenomenon in which the file system is journaling the database journal activities [5].

Several different mechanisms have been proposed to reduce the overhead of enforcing persistency in smartphones. One solution [5] is to tune the I/O stack, by, for example, external journaling in the EXT4 file system and write-ahead logging in SQLite. Another solution [7] is to integrate the recovery information into the database file itself so that the journal file is omitted. Arguably, the best remedy would be to perform in-place update with a non-volatile memory (NVM) [9, 11], such as Phase Change Memory (PCM), and thus avoid almost all the overhead of enforcing persistency. But adding NVM into the smartphone will increase the size and cost. However, to the best of our knowledge, the fact that the memory in smartphones is battery backed, which makes it potentially non-volatile for practical purposes, has been overlooked.

Therefore, in this paper we propose qNVRAM, a quasi Non-Volatile RAM design for smartphones. qNVRAM takes advantage of the "battery-backed" nature of the smartphones to make the data in qNVRAM persistent under almost all the failure conditions. We implement persistent Page Cache (pPCache) in SQLite using qNVRAM to perform in-place updates to the database files. We also employ LazyFlush to absorb the repeated writes to table files to further improve the performance. Our experimental results based on a Samsung Galaxy S4 smartphone show that the pPCache outperforms the write ahead logging (WAL) mode in transactions of random inserts, updates and deletes by 1.98×, 2.25× and 1.80× respectively. More substantially, when LazyFlush is enabled, the speedup over WAL in the same three types of transactions goes up to 16.33×, 15.86× and 14.13× respectively.

2 Background and Motivation

2.1 Failure Mode in Android Smartphones

There are four different types of high-level manifestations of failures [4], or failure modes, in mobile devices, namely, (1) application crash (an app stops to work unexpectedly), (2) application hang (an app is still active but delivering a constant output, e.g., blocked in an infinite loop or deadlocked), (3) self-reboot (the system forces a reboot as a consequence of a severe problem, e.g., ker-

Table 1: **Android application benchmarks.**

Application	Description
Angry Birds	Open the app, play for the first three levels, close the app
Chrome	Open the app, load 30 pre-defined web pages one by one, close the app.
Facebook	Open the app, "drag" the screen 5 times to load news feeds, post 3 status, send 3 messages, close the app.
Gmail	Open the app, load 3 new emails, search emails for 3 different key words, compose and send 3 emails, close the app.
Google Maps	Open the app, enter origin and destination address and get directions, zoom into the maps 5 times, close the app.
Twitter	Open the app, "drag" the screen 5 times to load news feeds, "drag" the screen 5 times to load news feeds, post 5 new tweets, close the app.
Youtube	Open the app, play 5 videos, each for 1 minute, close the app

nel panic), and (4) system freeze (the system delivers a constant output and does not respond to the user's input). A recent study [8] on the issues, reported between *Nov. 2007* and *Oct. 2009* in the Android Open Source Project (AOSP), points out that the Web Browser and Multimedia applications are most error-prone, and the kernel layer of the Android platform is sufficiently robust (only 4% of all the bugs are in kernel).

When an application failure (i.e., failure mode (1) or (2)) happens, the memory used by the application will be released by the OS; when the self-reboot (i.e., failure mode (3)) happens, the data in memory is lost since the OS will re-initialize the page table in the virtual memory system after the reboot; when the system freezes (i.e., failure mode (4)), the user-initiated recovery will be performed and the phone is forced power-off by hard reset if it is still freezing [4], and hence all the data in DRAM is lost. From the application's point of view, the data in memory will be lost under all of the four failures. Therefore, the current persistent storage options in mobile devices use multi-versioning to ensure update atomicity by storing the original data and modified data in two different places (journal/temporary file and the real data) in the non-volatile storage, which is presumably only flash storage in the smartphones.

2.2 Overhead of Persistency Enforcement

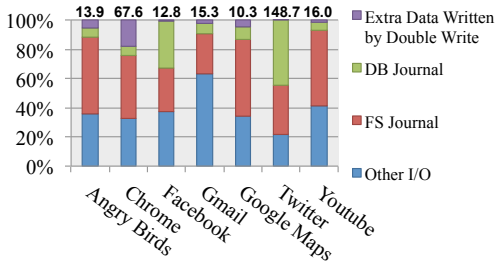
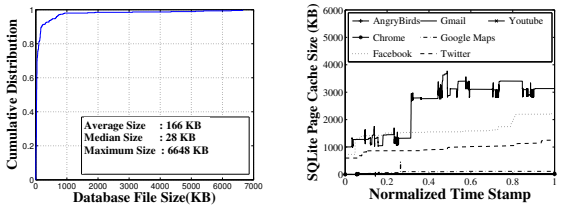


Figure 1: **The overhead of persistency enforcement. The amount of data (in MB) written to flash in each benchmark test is shown on top of each bar.**

The journaling and double-write schemes will introduce significant overhead to the storage system due to extra data transfers that result in additional I/Os. To better understand the reasons behind this, we carefully choose and run 7 top-ranked popular Android apps as application benchmarks, as described in Table 1, on a Samsung Galaxy S4 smartphone. We modify the block I/O path in the Linux kernel to collect information on block I/O requests, including logical block number (LBN), request size, inode number and the filename (if it is written to a file). The SHA-1 hash is also calculated for every 4KB chunk of a write request to identify the redundant data blocks written by the double-write scheme. The amount of extra data written to the flash storage resulting from maintaining atomicity is shown in Figure 1. In all these applications, 37% to 78% of the data written to the storage is exclusively for the purpose of atomicity. More than 75% of all data written by Twitter is going to the file system and database journals, while 18% of all data written by Chrome is unnecessary since it is not modified in the file updates. It is clear that the overhead of persistency enforcement is extremely high in smartphones.



(a) Database Table Size (b) SQLite Page Cache Size

Figure 2: **Characteristics of the SQLite databases in Android smartphones: (a) The cumulative distribution of the SQLite database file size in Android Smartphones; (b) The memory consumption of the SQLite page cache in different applications.**

2.3 Rethinking Memory Volatility in Smartphones

A fundamental assumption in most transactional systems that maintain the durability property, not limited to database systems, is that the memory is volatile. Thus they have to pay a very high cost to enforce data integrity and consistency through either journaling or copy-on-write [11]. Some modern systems incorporate non-volatile memory, such as NVDIMM [1] and PCM, to help eliminate the burden of persistency and boost the performance [9, 11]. However, the adoption of the NVM for mobile platforms is infeasible for reasons of cost and size, which are two of the most important design metrics for smartphones. Nevertheless, the battery in smartphones, especially the non-removable battery, can potentially make the DRAM non-volatile for practical purposes as explained next.

Despite of the possibility that some unknown bugs may power off the smartphone unexpectedly, the probability of that happening is considered extremely small. From our analysis of the AOSP issue reports (as of Feb. 2014), there are only 10 reports, i.e., 0.05% of all the 19670 issue reports related to defects in Android systems, indicate unexpected/random power-off, which infer a very small (though non-zero) chance that unexpected power failure may occur. Another study [2] on 600,000 technical support calls handled by WDS shows that 6% to 14% of these calls were assigned to hardware. The study also points out that hardware faults (if within a warranty period) usually result in the device being returned and entered into a reverse logistics process for repair or replacement. While physically pulling out the battery will lead to memory data loss, the fact that increasingly more smartphones are installed with irremovable batteries makes this a rare event. Therefore, in practice, the devices are more likely to suffer some deadly hardware faults, thus making recovery from data loss in these cases essentially superfluous.

Therefore, without increasing the cost and size of smartphones, we argue that it is possible to trade the data persistency in very rare cases for the performance boost in all cases. Some small changes to the current memory system design can preserve data persistency against the aforementioned four common failure modes, as follows. When an application failure (mode (1) or (2)) happens, the persistency is automatically achieved if the OS can preserve the application data that is critical to the transactions, such as the page cache in database, and provide the data to the application when it restarts; When self-reboot (mode (3)) happens, the data can be preserved and retrieved over reboot if the data is stored in a piece of physical memory at a fixed known location since DRAM does not loss power during reboot; When system freezes (mode (4)), it may result in hard reset in the worst case. The hard reset requires pressing the power button for 10 seconds, which is long enough for both capturing the action and flushing the important data to the flash storage, just like the battery-backed DIMM.

3 quasi Non-Volatile Memory

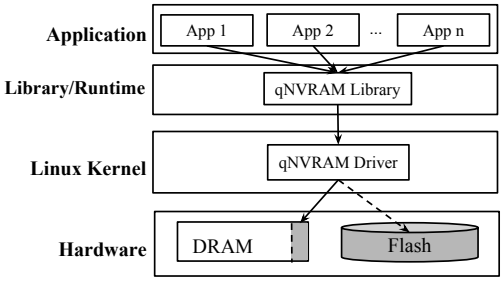


Figure 3: **The design of qNVRAM.**

Based on the above observations, we propose qNVRAM, a new design that makes data in DRAM non-volatile against almost all of the failures in smartphones. The architecture of qNVRAM is shown in Figure 3. qNVRAM consists of two major components: a library, which provides a programming interface to the applications, and a device driver, which manages the physical memory.

The library implements a simple memory allocator, which provides an easy-to-use memory interface for applications to obtain and release a piece of qNVRAM memory. The interface is shown in Table 2. The application can allocate and free qNVRAM using `qnvmalloc` and `qnvmfree` during normal execution, and retrieve the memory content using `qnvmretrieve` upon failure recovery.

The kernel-space device driver reserves a small chunk of physical memory (qNVRAM pool) when the kernel sets up the memblocks. The physical memory will be mapped into the user space when the application requests/retrieves memory from the qNVRAM pool. The driver is also responsible for flushing data from memory to the flash storage under certain circumstances. The first scenario is when the user tries to perform hard reset. To capture this action, we modify the power button interrupt handler so that it can notify the driver of the pressing event. When the button is pressed for 5 seconds, the driver will start flushing data to the flash storage. Our flush-on-fail mechanism is similar to but different from the Whole System Persistence (WSP) [10] approach. While upon failure the latter is triggered by a dedicated power monitor to flush all data in CPU registers and caches to NVRAM, the former flushes data in DRAM to flash without relying on dedicated hardware for signaling. The other scenario is when the qNVRAM pool is under memory pressure. When the allocator cannot find enough space in the qNVRAM pool, it will trigger the flush operation to swap out the memory associated with the processes that have already been killed. When the application restarts, it will be loaded into the memory for recovery.

The size of the qNVRAM pool can be adjusted at compile time. In our current implementation, the size of the qNVRAM pool is set to 20 MB (only 1% of the 2 GB physical memory in the Samsung Galaxy S4 smartphone). The reason why we reserve such a small piece of DRAM is twofold.

First of all, the persistent memory needed to enforce atomicity (SQLite page cache and file update buffer) is usually small. For example, unlike enterprise databases, the database tables in smartphones are usually very tiny [7], so are the page caches used by SQLite. To get the characteristics of databases in smartphones, we extract 453 well-populated database table files from 3 active Android smartphones and tablets. Figure 2(a)

Table 2: **qNVRAM application interface.**

Function	Description
`void *qnvmalloc(int uid, int magic, int size);`	Allocate a piece of qNVRAM memory, return the pointer value to the memory in application's address space.
`void qnvmfree(int uid, int magic);`	Free a piece of qNVRAM memory.
`void *qnvmretrieve(int uid, int magic, int *size);`	Retrieve a piece of qNVRAM memory using the uid and the magic number, return the pointer to the memory in application's address space.

shows the cumulative distribution of the database file size. More than 90% of the database files are less than 200 KB, and the median size is only 22 KB. We also rerun the application benchmarks of Table 1 in an actively used *Google Nexus 7* tablet with a modified SQLite library to monitor the dynamic behavior of the page cache memory usage in different applications. The overall page cache size in each application (one application may open multiple database files, thus creating multiple page cache instances) is plotted in Figure 2(b). Note that the page cache sizes of Facebook, Gmail and Twitter all start at a non-zero value because they are already running in the background as an Android service. Considering the fact that most smartphones run one user-facing app at a time and the background apps consume much less page cache, 20MB should be more than enough in practice.

Secondly, the qNVRAM pool needs to be flushed to the flash storage before the smartphone is powered off by hard reset. The sequential write bandwidth in Samsung Galaxy S4 is 15.1 MB/s, which means that we can flush the 20MB data in 1.3 seconds, well within the 5-second window between capturing the hard button press event and power-off.

4 Case Study: Persistent Page Cache in SQLite

4.1 Persistent Page Cache and LazyFlush

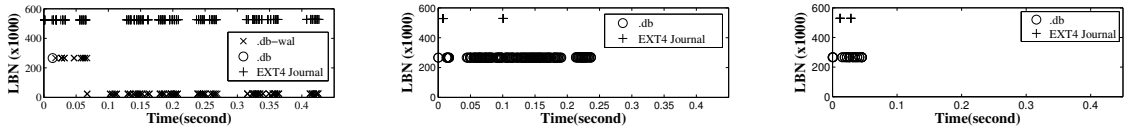
A good use case for qNVRAM is the page cache in SQLite. Making the page cache persistent will significantly reduce the volume of reliablity-induced writes [3] to the flash storage because database journaling is no longer needed and the JOJ anomaly is also eliminated. We have implemented a persistent page cache in SQLite 3.7.12 using the qNVRAM API. SQLite will allocate a piece of memory from the qNVRAM pool that will be used as the persistent page cache. When a write transaction is committed, SQLite will perform in-place update to the `.db` table file without logging the changes to journal files since the latest copy is already persistent. And when the transaction aborts, the modified pages in the page cache can be rolled back to the version in the `.db` file. Note that the pPCache does not support transactions that cannot fit in the page cache, which should not be a big problem since the transactions and table files are very small. To further exploit the locality of page accesses, the page cache can temporarily defer flushing dirty pages to table files, a process called LazyFlush, so that the repeated writes can be absorbed by the page cache. When LazyFlush is enabled, an undo log in qNVRAM is implemented to quickly revert to the old version of the page content of a transaction when it aborts. The undo log is dynamically allocated during a transaction, so that it is transient and will not consume too much qNVRAM memory. In our current implementation, we set a threshold on the number of dirty pages in the page cache such that when the number of dirty pages is larger than the threshold it will flush all dirty pages to the files.

Recovery in the persistent page cache is fairly simple. The SQLite will retrieve the page cache from the qNVRAM pool when the application restarts. Then it will scan all the pages in the page cache and perform checkpointing by flushing all the committed but unflushed pages to the database table file.

4.2 Preliminary Evaluation

We evaluate and compare the performance of pPCache and LazyFlush against the baseline WAL mode on the *Samsung Galaxy S4 Google Edition* with a quad-core CPU, 2GB DRAM and 16GB eMMC flash memory formatted with the EXT4 file system. The smartphone is running Android 4.3 and Linux kernel 3.4. Our tests first initialize the SQLite table with 2000 records, each of which consists of an integer key and 100-character value. Then we sequentially and randomly, respectively, insert 1000 records, update 1000 records and delete 1000 records, of which each corresponds to a transaction (i.e., for a total of 6000 transactions). For each test case, we run 10 times and report the average throughput (transactions per second). The page cache size is configured to be 400 KB (100 4096-byte pages), while the table size is around 300KB. A smaller page cache will not affect the result since our tests only perform write transactions, and the database can always find a clean page to accommodate the new data.

Figure 5 illustrates the performance of pPCache with and without LazyFlush. The pPCache without LazyFlush speeds up the WAL mode in random `insert`, random `update`, random `delete`, sequential `insert`, sequential `update` and sequential `delete` by 1.98×, 2.25×, 1.80×, 2.04×, 2.17× and 2.64× respectively. When LazyFlush is enabled with a small threshold of 5 pages, the random `insert`, `update` and `delete` get 3.29×, 4.56×, 4.37× performance boost respectively, while the sequential `insert`, `update` and `delete` achieve 8.80×, 13.93× and 9.24× speedup respectively. The performance of sequential operations drastically benefits from the write locality to the database



(a) WAL (b) Persistent Page Cache (c) Persistent Page Cache + LazyFlush

Figure 4: **Block I/O pattern of the Persistent Page Cache and WAL. The threshold of LazyFlush is set to 5 pages.**

files. In a database with several thousands of records, which is very common in smartphones, one `insert`, `update` or `delete` will only modify a small number of pages (i.e., 3 per `insert`, 2 per `update` and 3 per `delete`). And two consecutive operations in the sequential `insert`, `update` or `delete` are very likely to modify the same page since they are stored in the same leaf nodes in the B+ tree. Moreover, every transaction will modify the file change counter in the first page of the database file. As we increase the threshold, the throughput also increases. When the threshold is set to ∞, the database becomes an in-memory database. In this scenario, the speedup is 16.33×, 15.86× and 14.13× in random `insert`, `update` and `delete` transactions respectively, and 15.40×, 15.09× and 15.76× in sequential `insert`, `update` and `delete` transactions respectively.

Figure 4 shows the block I/O access pattern of 100 `insert` transactions. In the WAL mode, a single `insert` transaction will result in 16KB data written to the log file and 20KB data written to the EXT4 journal. In the pPCache mode, each `insert` transaction will only write 12KB data to the `.db` file. When the size of the database file remains unchanged, the `fdatasync` will not trigger file system journaling. As shown in Figure 4(b), there are only two EXT4 journal commits in the 100 `insert` transactions. In the LazyFlush mode, most of the repeated writes are absorbed by the persistent page cache and thus much less data is written to the database file. The WAL mode writes 1.72MB (1.05 MB EXT4 journal blocks and 0.65 MB WAL log blocks and 0.02 MB table file blocks) to the flash storage, while it is only 1.16 MB (1.05 MB table file blocks and 0.11 MB EXT4 journal blocks) for pPCache and 0.07 MB (0.03 MB EXT4 journal blocks and 0.04 MB table file blocks) for LazyFlush. It clearly indicates that the persistent page cache can significantly reduce the number of journal commits, as well as the volume of I/Os destined to the database files when LazyFlush is enabled.

5 Discussion and Future Work

qNVRAM provides a nearly persistent memory in smartphones, which can be used to speed up different applications. The applications that store important data in files can allocate a piece of qNVRAM as write buffer for the modified blocks; the file system can also employ qNVRAM as a writeback buffer for metadata updates. The current design of the qNVRAM pool uses reserved physical memory with a fixed size. We plan to incorporate the qNVRAM into the virtual memory system so that the size of qNVRAM pool can be dynamically changed and hence increase the memory efficiency. We also plan to further study the various failure characteristics of smartphones to better understand and overcome qNVRAM's limitations.

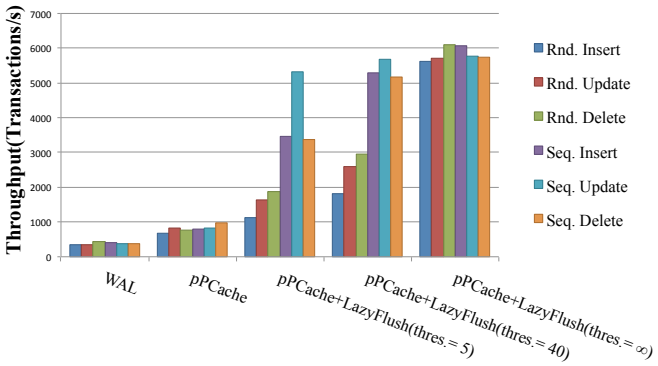


Figure 5: **The Performance of Persistent Page Cache and LazyFlush.**

6 Acknowledgments

We are grateful to anonymous reviewers and our shepherd, Anirudh Badam, for their feedback and guidance. This work is supported by the US NSF under Grant No. NSF-CNS-1116606, NSF-CNS- 1016609, NSF-IIS-0916859, and NSF-CCF-0937993.

References

[1] AGIGARAM Non-Volatile System. `http://www.agigatech.com`.

[2] Controlling the Android. `http://itcafe.hu/dl/cnt/2011-11/78806/controlling-the-android.pdf`.

[3] Peter M Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. *The Rio file cache: Surviving operating system crashes*, volume 31. ACM, 1996.

[4] Marcello Cinque. Enabling on-line dependability assessment of android smart phones. In *Dependable Systems and Networks Workshops*. IEEE, 2011.

[5] Sooman Jeong and et al. I/O stack optimization for smartphones. In *USENIX ATC*, 2013.

[6] Hyojun Kim and et al. Revisiting storage for smartphones. *ACM Transactions on Storage*, 2012.

[7] Wook-Hee Kim and et al. Resolving journaling of journal anomaly in android i/o: Multi-version b-tree with lazy split. In *USENIX FAST*, 2014.

[8] A Kumar Maji and et al. Characterizing failures in mobile oses: A case study with android and symbian. In *Software Reliability Engineering*. IEEE, 2010.

[9] Eunji Lee and et al. Unioning of the buffer cache and journaling layers with non-volatile memory. In *USENIX FAST*, 2013.

[10] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 401–410. ACM, 2012.

[11] Jishen Zhao and et al. Kiln: closing the performance gap between systems with and without persistence support. In *MICRO*. ACM, 2013.