Energy Efficient IO stack Design for Wearable Device

Junghoon Kim* Sundoo Kim Juseong Yun Youjip Won**
*Samsung Electronics, Korea Hanyang University, Korea **KAIST, Korea
jhoon20.kim@samsung.com* {sksioi12,yjs05}@hanyang.ac.kr youjip.won@gmail.com**

ABSTRACT

In this paper, we visit the energy consumption issue of the IO subsystem for wearable device. We argue that the IO stack for wearable device is subject to extreme inefficiency in terms of the IO volume and the number of flush requests. The deficiency of the IO stack leaves a room for improvement from the energy consumption's point of view. In this work, we characterize the IO access patterns of the smartwatch device, develop a model to estimate the energy consumption from the given IO traces, and propose a set of methods to optimize the IO subsystem behavior for energy saving. In smartwatch, the amount of data written daily is 10× as large as the amount of data read daily. The amount of data written to the flash storage each day is approximately as large as the free space in the storage device. To minimize the energy consumption associated with the IO activities in smartwatch, we propose *Metadata Embedding* and *Selective Directory Sync* for SQLite DBMS and *Flushless Durability Guarantee* for the filesystem. We implement the proposed techniques in the commercially available smartwatch product. The proposed techniques reduce the energy consumption associated with the IO activities by 60%. It corresponds to 3% savings in the overall energy consumption. It is achieved solely via software optimization.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile devices**;

KEYWORDS

Smartwatch, SQLite, Tizen, Battery, Energy Consumption

ACM Reference Format:
Junghoon Kim, Sundoo Kim, Juseong Yun, and Youjip Won. 2019. Energy Efficient IO stack Design for Wearable Device. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC'19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3297280.3297491

1 INTRODUCTION

Thanks to the advancement in smartwatch, a variety of digital information such as the text messages, the emails, the weather forecast, and the stock price alert, becomes available at one's wrist. It is used not only to display the information but also to act as a surrogate to record every dimension of the user's physical behaviors such as the heart rate, the number of steps the user has taken, and the running distance. For the past five years, the unit sales of smartwatch have doubled every year [28]. Given this trend, the smartwatch may soon become one of the essential daily gadgets.

Despite the rapidly growing popularity of the smartwatch, it has been mere five years since the smartwatch first appeared on the market. It has not been long since the smartwatch has been under the technical exploration [11, 20, 21]. The smartwatch is not a scaled-down version of the smartphone with the inferior hardware specification mounted on the wrist. The smartwatch is fundamentally different device in its nature from the smartphone. One of the key concerns for the wearable device is battery life. The battery life of the smartwatch product lasts three to four days [29]. It is much longer than that of the smartphone, which is charged on daily basis. However, given that one replaces the battery of the wrist watch in every couple of years, we believe that the people's expectation on the battery life of the smartwatch is much higher than that of the smartphone. On the same token, the battery life of the smartwatch still leaves much to be desired and is subject to the further investigation [7, 24, 25].

In smartphone, CPU, GPU, network, and display components are the major sources of energy consumption [2]. The energy consumption associated with accessing the storage device is not significant [18]. In contrast, the smartwatch is mainly used for conveying the push notifications and tracking the fitness activities [21]. Wearable device has 5 to 6 sensors ranging from the heart rate monitor and gyro sensor to the GPS sensor [1, 8]. These sensors collect a wide variety of data. Wearable device first stores the collected data in its storage and asynchronously synchronizes it to the smartphone. We carefully argue that the energy consumption associated with the IO activities accounts for larger fraction of energy in wearables than in smartphone and that the IO activities account for non-negligible amount of total energy consumption in the wearable device.

In this work, we aim at reducing the energy consumption associated with IO activities in smartwatch. We modify the SQLite DBMS and EXT4 filesystem so that the smartwatch does not generate unnecessary IO and is free from generating unnecessary flush. We collect the IO trace from the daily worn smartwatch device and characterize the IO behavior of the smartwatch. We develop a model to estimate the energy consumption associated with the IO activities from the IO trace. We develop Metadata Embedding, Selective Directory Sync, and Flushless Durability Guarantee to reduce the energy consumption associated with the smartwatch IO. The contribution of this work can be summarized as follows.

- We find the essential characteristics of the smartwatch IO. Smartwatch IO is heavily write intensive. The amount of data written to the storage is 10× larger than the amount of data read from the storage on the daily average. The amount of data written to the storage each day is as much as the available storage space.

SAC'19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297491

- The proposed energy model enables us to estimate the energy consumption from the IO trace with simple voltage monitor. The energy model measures the energy difference between the different types of workloads and computes the energy consumption associated with the DMA transfer and writeback cache flush.
- We propose Metadata Embedding, Selective Directory Sync, and Flushless Durability Guarantee to mitigate the energy consumption overhead of the smartwatch IO. With Metadata Embedding, we reduce unnecessary IO traffic by eliminating the internal fragmentation of the SQLite journal organization. With Selective Directory Sync, we eliminate the unnecessary EXT4 journal transaction. With Flushless Durability Guarantee, we effectively eliminate the flush operation in `fsync()`and `fdatasync()`by exploiting the non-removable nature of the smartwatch battery installation. It saves the energy consumption associated with programming a flash cell.

With all these techniques, we reduce the energy consumption associated with the IO activities by 60%. Overall, the proposed optimization techniques save 3% of the energy consumption, which corresponds to 2.8 mAh per day, solely with software only optimization and without any performance overhead.

2 BACKGROUND

2.1 SQLite DBMS

Arguably, SQLite is the most widely deployed DBMS in the world [6]. SQLite is the default DBMS in all smartphone platforms including Android, iOS, Tizen, Firefox, etc. Most smartphone applications rely on SQLite DBMS to manage their data persistently. By default, SQLite uses B-tree for its database file. An SQLite database file consists of a database header page and the set of database nodes. By default, the node size is 4 KByte. Fig. 1 illustrates the structure of the database file. The database header page contains 100 Byte header and the database schema at the beginning and at the end of the database header page, respectively. The rest of the database header page remains unused. Database node takes heap like structure as shown in Fig. 2. There is an array of <key, pointer> pairs at the beginning of the node. They are sorted with a key. The pointer represents the location of the associated values. The values are placed from the end of the node. The values grow in the opposite direction to the <key, pointer> pairs.

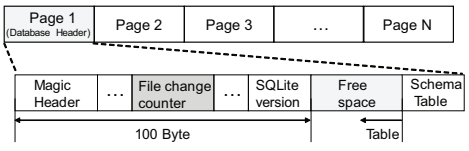


Figure 1: Database file structure in SQLite

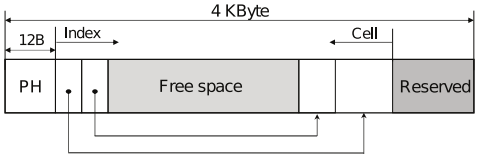


Figure 2: Database node structure in SQLite (PH: Page Header)

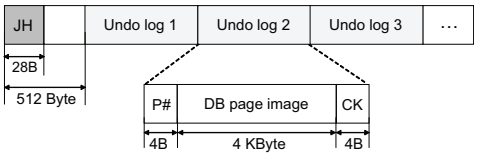


Figure 3: Rollback journal file structure in SQLite (JH: Journal Header, P#: Page Number, CK: Checksum)

Fig. 3 illustrates the structure of the rollback journal file of SQLite DBMS. The database journal file consists of 512 Byte journal header and a set of undo-logs. An undo-log consists of 4 Byte page number, the image of the database page before an update, and 4 Byte checksum. The actual journal header size is 28 Byte. SQLite appends the padding so that the journal header forms an entire sector. This is to make the update on the journal header robust against the *power crash* [27]. When the journal header and the undo-logs are on the same sector, a sudden power off when the journal header is updated may corrupt the undo-logs on the same disk sector. SQLite takes pessimistic approach to protect the database journal file against the corruption due to unexpected power failure.

For atomicity and durability of a transaction, SQLite provides two journaling schemes: rollback journaling and Write-Ahead Logging (WAL). SQLite provides three different journaling modes for rollback journaling; DELETE, TRUNCATE, and PERSIST. We like to limit our discussion to PERSIST mode. PERSIST is the default journaling mode in SQLite. Algorithm 1 describes the pseudo code for SQLite transaction in PERSIST mode. An SQLite transaction consists of three phases: logging, database update, and log-reset.

Let us examine the details of each phase. In logging phase, SQLite records 0 to the number of log pages field (i.e. page count) in the journal header. This is to denote that the transaction has started. Then, SQLite logs the undo-logs. After logging, SQLite calls `fdatasync()`to make the results durable. Then, SQLite calls `fdatasync()`for the parent directory. This is to ensure that the

Algorithm 1: Pseudo code for transaction in PERSIST mode

```
1  jfd : journal file descriptor
2  fd : database file descriptor
3  dirfd : parent directory file descriptor
4  function PERSIST transaction(jfd, fd)
5      write (jfd,journal header);
6      for each logs[i] do
7          write (jfd, page number);
8          write (jfd , logs[i]) ;
9          write (jfd, checksum);
10     end
11     fdatasync(jfd);
12     fdatasync(dirfd);
13     write(jfd , page count);
14     fdatasync(jfd) ;                          /* Logging phase */
15     for each dirty_pages[i] do
16         write(fd , dirty_pages[i]);
17     end
18     fdatasync(fd) ;                  /* Database update phase */
19     journal reset();
20     fdatasync(jfd) ;                        /* Log-reset phase */
21     return;
22 end
```

newly created directory entry for the rollback journal file is persisted. This is required only when the rollback journal file is newly created, e.g. when the newly downloaded application is executed for the first time. The `fdatasync()`for persisting the directory block is prohibitively expensive. It is expensive in two aspects: IO volume and flush. A single EXT4 journal transaction consists of at least three disk blocks: a header block for EXT4 journal transaction, one or more blocks (e.g. updated metadata blocks) to be logged in the filesystem journal transaction, and a commit block of filesystem journal transaction. Committing a filesystem journal transaction accompanies two flush operations: one after writing the journal log blocks and the other after writing the journal commit block. After persisting the parent directory, SQLite writes the number of log pages to the SQLite journal header. After writing the number of log pages to the SQLite journal header, SQLite calls `fdatasync()`to persist the header page of the SQLite journal file.

In database update phase, SQLite writes one or more updated database node blocks and the associated database header to the database file. SQLite calls `fdatasync()`to persist the result of the database update. In log-reset phase, SQLite updates the journal header to 0. This is to denote that the transaction has made durable successfully. SQLite calls `fdatasync()`to make the result of the journal header reset to be durable.

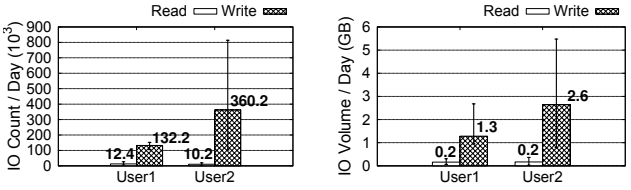
A single SQLite transaction creates five `fdatasync()`calls and writes at least six additional blocks for the journal header and EXT4 journal transaction. If an `insert` transaction updates a single database node only, then at least six blocks are written to the storage device excluding the undo-logs and the database updates. Even worse, these writes are interleaved by a number of flush operations.

2.2 PMIC/Shutdown

Mobile device has PMIC (Power Management Integrated Circuit). PMIC monitors the voltage level and protects the system against under-provisioning of the voltage. The PMIC consists of the monitor (i.e. fuel gauge), charger, and voltage regulator. The fuel gauge monitors the voltage level. It reads the register value of the PMIC to obtain the voltage level of the battery. The charger charges the battery. The regulator regulates the voltage level of the battery.

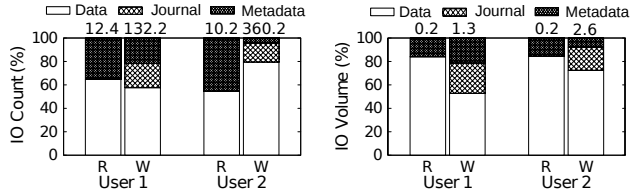
We examine Gear S3 Tizen, a target wearable platform for this study. In Tizen wearable platform, the Linux kernel driver for PMIC polls the voltage level in every 30 seconds. Based upon the voltage state of the battery, a kernel daemon, `deviced`, takes the different actions. Tizen defines five battery states subject to the remaining battery life: Normal, Warning (<15%), Critical (<5%), Poweroff (<1%), and Realoff (0%). When battery state changes from Poweroff to Realoff, the `deviced` performs powerdown. The detailed power off sequence is as follows. The device daemon flushes the dirty page cache entries to prevent the data loss. The kernel stops all daemons. Then, the kernel shutdowns the individual devices and executes the shutdown functions associated with each device driver if any such functions are registered.

Unless the battery is physically detached from the device or unless the PMIC and the associated software components are buggy, the PMIC protects the system from any data loss in case of the normal power outage. It takes less than a few milliseconds to flush the writeback cache of the mobile storage device. When PMIC reads the remaining battery capacity to be 0%, the battery still can operate the device for a few more seconds. It gives the device drivers sufficient time for persisting the dirty pages.



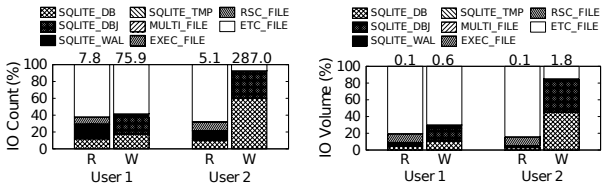
(a) IO Count (b) IO Volume

Figure 4: Average daily IO



(a) IO Count ($\times 10^3$) (b) IO Volume (GB)

Figure 5: Daily IO count and volume (block type)



(a) IO Count ($\times 10^3$) (b) IO Volume (GB)

Figure 6: Daily IO count and volume (file type)

3 IO CHARACTERISTICS

3.1 Collecting IO Traces

Acquiring a correct understanding on the underlying IO traffic is an essential part of the study. We perform limited study on IO characteristics of the smartwatch. Comprehensive analysis on IO characteristics requires the trace collection from the extensive set of users and the users have to be chosen to properly represent diverse sectors of the society such as gender, profession, age, etc. The extensive study of the IO characteristics of the smartwatch is beyond the scope of this work. In this study, we collect the IO trace from the two male software developers. They wear Samsung Gear S3 smartwatch as the normal daily worn device. We collect the trace for one month. We use a trace collection tool, `Androtrace` [19]. We port Androtrace to Tizen platform.

3.2 Primitive Analysis

We first examine the average daily IO count and volume. As shown in Fig. 4, the IO behavior of the smartwatch is heavily write intensive. On the average, the user 1 and user 2 write 1.3 GByte and 2.6 GByte per day. The number of writes is 11× and 35× higher than the number of reads in user 1 and user 2, respectively. The smartwatch device has only 1.4 GByte of storage space left. The smartwatch has 4 GByte flash storage and the pre-installed software occupies

	Med.	75%	99%	99.9%	Max
Transaction	68	104	160	184	1256
.db	16	36	60	60	336
.db-journal	28	52	64	64	352
EXT4 journal	16	20	56	100	1204

Table 1: IO size of an SQLite transaction, DB updates, journal updates, and filesystem journal (KB)

Transaction count (%)	Med.	75%	99%	Max
.wnoti-service.db (76.1%)	76	116	160	1256
.shelath.db (9.9%)	72	76	128	336
.helathShare.db (6.7%)	64	64	88	220
.rua.db (4.2%)	68	76	128	304
.msg-consumer-server.db (1.5%)	56	72	168	588

Table 2: Quantiles of SQLite transactions in Top 5 DB (KB)

2.6 GByte of the storage capacity. For user 1, the amount IO written to the storage in a day equals the amount of the available storage space. For user 2, the situation is worse. The amount of IO written to the storage is nearly 2× to the available storage space. We are not aware of any computing systems that write entire available storage space for worth of data every day.

We examine how the filesystem partition is accessed. We partition the filesystem partition into three regions: data, metadata, and filesystem journal. We analyze the IO count and volume for each region. Fig. 5 illustrates the result. For both users, journal writes account for substantial fraction of IOs written to the storage. The journal writes account for 26% and 20% of the total IO volume in user 1 and user 2, respectively. The IO requests to the data region account for 53% and 72% of the total IO volume in user 1 and user 2, respectively. Metadata writes account for remaining fraction of IOs. We examine how the data region is accessed. We categorize the files into eight types. They are SQLite database (`*.db`), SQLite journal (`*.db-journal`), SQLite WAL (`*.db-wal`), SQLite temp (`*.db-shm`), multimedia file (`*.jpg,*.mp3,*.mp4,*.MOV,*.avi`), executable file (`*.so`), resource file (`*.dat,*.xml,*.cache`), etc. Fig. 6 illustrates the result. The write volume for SQLite database file accounts for 10% and 44% of total file IOs and the write volume for SQLite journal file accounts for 15% and 38% of total file IOs in user 1 and user 2, respectively.

3.3 SQLite Transaction Analysis

The IO requests associated with SQLite DBMS account for as much as 90% of total IOs written to the storage device from IO volume as well as IO count's point of view as shown in Fig. 6. We parse the collected trace and identify the beginning and the end of a transaction from the collected trace [6]. We analyze the size of the SQLite transactions. The average size of the SQLite transactions is 80.8 KByte. In PERSIST mode, an SQLite transaction updates the rollback journal file and the associated database file. The results of the updates are made durable through `fdatasync()`. An `fdatasync()` occasionally accompanies EXT4 journal transaction. Table 1 summarizes the size of an SQLite transaction and the sizes of the associated database updates, rollback journal updates, and filesystem journal, respectively. In median, an SQLite transaction

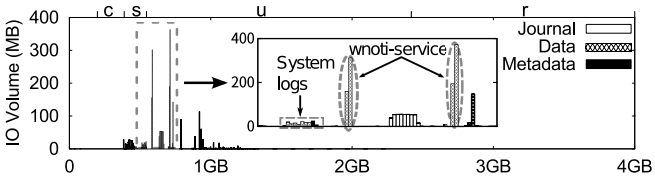


Figure 7: Storage access footprint of user 2 (daily average), bin size = 4 MB (c : csa, s : system-data, u : user, r : rootfs)

writes four pages to the database file while it writes eleven pages to the SQLite journal file as well as the filesystem journal.

The database accesses are extremely skewed. We examine the frequency of database updates in Gear S3. Table 2 presents the top 5 frequently updated databases. The IO requests to the top 3 most frequently updated databases account for over 90% of the total SQLite transactions. Heavy SQLite IOs in Gear S3 are mainly driven by the push notification and fitness activity tracking. The `wnoti-service.db` is the most frequently updated database; it accounts for 76.1% of total SQLite transactions. The `wnoti-service.db` database file is updated when Gear S3 smartwatch receives push notifications such as an instant message and an email from smartphone's applications. Also, the `shealth.db` and `healthShare.db` are one of the top 5 most frequently updated databases; they account for 9.9% and 6.7% of total SQLite transactions, respectively. These two databases are for health application. The `rua.db` records the application launch history. It manages recently used applications. The access to `rua.db` accounts for 4.2% of total SQLite transactions. The `msg-consumer-server.db` is managed by message application; it accounts for 1.5% of total SQLite transactions.

We examine the storage footprint of the smartwatch IOs. Fig. 7 illustrates the footprint of the storage IOs of user 2. Storage access is heavy tailed. 4.5% of the storage space accounts for 90% of the data blocks written. We partition the entire 4 GByte storage space into 4 MByte unit. We compute the amount of data written to each 4 MByte partition each day. The storage capacity is 4 GByte and the pre-installed binaries such as Linux kernel and the various applications occupy 2.6 GByte of the storage. There are 1.4 GByte of space available in the storage. The smartwatch writes the data block to all free pages in the flash storage each day. The frequently written storage regions correspond to the database file of the frequently used applications and the filesystem journal. This characteristic is observed in both user 1 and user 2.

4 MODELING IO ENERGY CONSUMPTION

We develop a model to obtain the energy consumption of the IO operations from a given IO trace. The objective of this model is to use minimal instrumentation equipment so that anyone with a simple voltage instrumentation device can use this model. We use a Monsoon monitor to measure the aggregate energy consumption of the IO request [26]. Our modeling consists of two parts: identifying a session and computing the energy consumption.

4.1 Identifying a Session

IO trace is a sequence of IO operations. We partition the IO trace into a sequence of sessions. There are two types of sessions: SQLite session and non-SQLite session. An SQLite session denotes a sequence

id	op	bt	size (KB)	pname	fname	
76692	S	D	4	Music-control	Music-control-service	S1
76693	S	JD	60	jbd2		
76694	S	JC	4	jbd2		S2
76695	S	D	4	shealth-service	Shealth.db-journal	S3
76696	S	D	16	shealth-service	Shealth.db-journal	
76697	S	JD	8	jbd2		
76698	S	JC	4	jbd2		
76699	S	D	4	shealth-service	Shealth.db-journal	
76670	S	D	4	shealth-service	Shealth.db	
76671	S	D	12	shealth-service	Shealth.db	
76672	S	D	4	shealth-service	Shealth.db-journal	

Figure 8: Sessions in the IO trace (S1 : Non-SQLite session [DMA 4 KB, 60 KB and Flush 64 KB], S2: Non-SQLite session [DMA 4 KB and Flush 4 KB], S3: SQLite session [3 updated database nodes])

of IO operations associated with executing an SQLite transaction. One or more transactions can be interleaved with each other. A multiple SQLite transactions can form a single SQLite session. Non-SQLite session is a set of IO operations that is irrelevant to the SQLite transaction. A non-SQLite session is delimited by the flush operation. Fig. 8 illustrates an example. There are three sessions in the trace, S_1, S_2 and S_3. S_1 and S_2 are associated with committing an EXT4 journal transaction. In EXT4 journaling, JBD thread first persists the journal log blocks and then it persists the journal commit block. JBD thread issues a flush command after it transfers the log blocks of the filesystem journal transaction. The first non-SQLite session is delimited (S_1). After it writes the journal commit block to the storage, it issues a flush command again. The second non-SQLite session is delimited (S_2). An SQLite session starts with writing the undo-logs to the SQLite journal file. In Fig. 8, an SQLite session starts at the trace id of 76695. With regular pattern of the SQLite journal transaction, we can unambiguously identify the beginning and the end of the SQLite session [6]. Here, the SQLite session ends at id 76672 (S_3).

For each session, we count the number of blocks written to the storage. For the non-SQLite session, we count the number of updated pages. Session S_1 writes 4 KByte and 60 KByte, respectively. Session S_2 writes 4 KByte. For the SQLite session, we count the number of updated database pages. In SQLite transaction, the IO operation amplifies. We only count the number of updated pages in the database file. Session S_3 updates a database header page (id 76670) and three database nodes (id 76671).

4.2 Computing the Energy Consumption

Computing the energy consumption of the IO trace consists of two parts: measuring the baseline energy consumption for per-session and computing the total energy consumption. First, we measure the baseline energy consumption for varying SQLite session sizes. The behavior of an SQLite is highly sensitive to the various configuration options [6]. Gear S3 uses PERSIST mode journaling and FULLSYNC option in SQLite. We create the transaction that updates N pages and repeat the transaction 10,000 times. We measure the energy consumption when the device is in the idle state and when the device is performing a transaction. We take the difference between the two and compute the average amount of energy consumption in updating N database pages.

Second, we obtain the baseline energy consumption for non-SQLite session with varying sizes. Non-SQLite session consists of

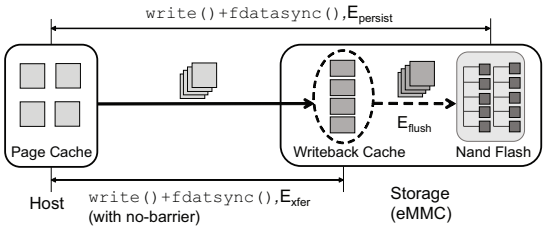


Figure 9: Energy consumption for different IO phases

two phases: DMA and flush. In DMA phase, the host transfers the data blocks to the storage device. In this case, the data blocks do not necessarily reach the disk surface. The data blocks are delivered to the writeback cache of the storage device unless the write command has FUA option. We denote the energy consumption associated with transferring the data blocks to the storage device as E_{xfer}. A non-SQLite session is delimited by flush operation. The flush command is not captured in the block trace. We can indirectly infer the flush command since the last write request for writing the log blocks in the filesystem journal transaction or the write request for writing the journal commit block in the filesystem journal transaction carries flush flag such as REQ_FLUSH and REQ_FUA. The energy consumption for flushing the writeback cache is denoted by E_{flush}. We measure the baseline energy consumption for the non-SQLite session under varying the number of blocks written. Under varying the number of data blocks, N, we write N blocks and call `fdatasync()`. With this experiment, we obtain the energy consumption associated with persisting N blocks, $E_{persist}$. We perform non-allocating writes not to trigger filesystem journaling. Then, we set the `no-barrier` mount option and perform the same experiment. With `no-barrier` option, the filesystem omits issuing the flush command in `fsync()` and in `fdatasync()`. It corresponds to the energy consumption required to transfer the data blocks to the writeback cache, E_{xfer}. Taking the difference between the two, we obtain the amount of energy for flushing N blocks that reside in the writeback cache, E_{flush}. Fig. 9 schematically illustrates the concept of $E_{persist}$, E_{xfer} and E_{flush}.

The total IO energy consumption for a given IO trace can be obtained through the sum of energy derived from individual IO sessions. We determine the number of updated pages in the non-SQLite sessions and the number of updated database pages in the SQLite sessions. We apply the baseline energy consumption measurement.

5 OPTIMIZATION

5.1 Metadata Embedding

SQLite journal structure is not aligned with 4 KByte block size. A journal file consists of a 512 Byte journal header and a sequence of undo-logs. Each 4 KByte undo-log page is prepended and appended with 4 Byte page number and 4 Byte checksum, respectively. Logging a single 4 KByte database page to the journal file corresponds to writing 512 Byte journal header, 4 KByte database node, 4 Byte page number, and 4 Byte checksum. This fragmented structure entails severe write amplification in undo-logging. When a transaction needs to log two database pages, SQLite writes three pages for the journal header update and the undo-logs. Then, it writes the journal header page again to record the number of undo-log

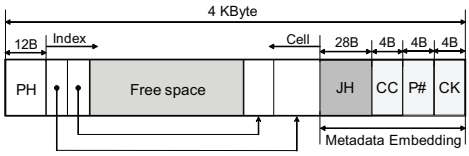


Figure 10: Rollback journal page structure with Metadata Embedding (PH: Page Header, JH: Journal Header, CC: File Change Counter, P#: Page Number, CK: Checksum)

pages. In this scenario, the actual amount of data blocks written to the storage is twice as many as the amount of the updated database pages in logging phase.

We modified the structure of journal file and the SQLite log structure to eliminate the internal fragmentation. This technique is called Metadata Embedding. Metadata Embedding has been proposed to improve the throughput of the smartphone applications [15, 17]. We use this technique to minimize the energy consumption associated with IO activities. Fig. 10 illustrates the structure of undo-log page in the SQLite journal file with Metadata Embedding. In SQLite, we can configure the size of each B-tree node and can reserve a certain amount of space at the end of the node. The size of a B-tree node is set to 4 KByte. Different from the existing SQLite journal file, we reserve 40 Byte in the B-tree node. We record the journal header (28 Byte), file change counter (4 Byte), page number (4 Byte), and checksum (4 Byte) at the reserved space. In the original SQLite database design, file change counter is stored in the database header page. We will explain the meaning of storing file change counter at the reserved space of journal file in Section 5.3. The actual space of a B-tree node decreases from 4096 Byte to 4054 Byte. The available space decreases by 1%. We believe that 1% space decrease is not unreasonable given the benefit of Metadata Embedding. In the first block of the journal file, the reserved space harbors the journal header, file change counter, page number, and checksum. In the rest of the blocks of the journal file, the reserved space carries the associated page number and checksum.

Flash storage guarantees 4 KByte atomic write. This is because FTL services the write operation in out-of-place manner and the content of a given logical page is updated via changing the physical page number of the associated logical page to a new location. With Metadata Embedding technique, we embed the journal header at the reserved space of the first undo-log page. We also embed the page number and checksum into the undo-log page for aligning the size of B-tree node to 4 KB. Through this, we can eliminate 2 page writes with 1 `fdatasync()` in logging phase. Despite its simplicity, the implication of Metadata Embedding is substantial. For example, when you log one database page, we can reduce the total write operations associated with undo-logging from three to one.

5.2 Selective Directory Sync

The second optimization is *Selective Directory Sync*. The directory synchronization is not needed in all transactions. It is required only when the journal file is created for the first time. SQLite removes the rollback journal file when it closes the database connection and it creates the rollback journal file when it opens the database connection. The application keeps the connection to the database file when it is active. The journal file is rarely created. The benefit of eliminating the directory synchronization is significant. It saves the IOs associated with committing a filesystem journal transaction. A filesystem journal transaction consists of at least three disk blocks and it is written with two flush operations.

5.3 Relocating File Change Counter

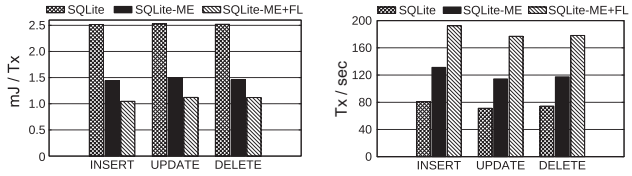
SQLite maintains the database version number (i.e. file change counter) at the database header page. SQLite increases the file change counter in the database update phase. After the SQLite updates the database node, it increases the file change counter by one. The database header with the updated file change counter is written to the database file at the end of the database update phase. SQLite calls `fdatasync()`to make the result of the database updates durable. File change counter is used to determine if the database pages cached in the application's local address space are up-to-date. When an application starts an SQLite transaction, it first compares the database version number of the cached the database with the database version number of the associated database file in the disk. If they are the same, SQLite updates the cached database pages. Otherwise, SQLite reads the database file from the disk again.

In this work, we migrate the file change counter from the database header to the journal header as shown in Fig. 10. In rollback journaling, SQLite reads the journal header when SQLite starts a database transaction. At this time, SQLite can extract the database version number by reading the file change counter in the journal header. When the transaction finishes, SQLite updates the database version number and stores the updated database version number at the journal header. The journal file persists with the associated database file. We can safely manage the database version number with the journal file. Via relocating the file change counter to the journal file, we can save an IO associated with updating the database header in database update phase.

5.4 Flushless Durability Guarantee

Any reasonably capable mobile devices have PMIC to protect the system against the power crash and to avoid the data loss. Most modern filesystems take overly pessimistic approach in committing a journal transaction [5]; the filesystem uses a flush command to control the order in which the results of the writes reach the disk surface. This pessimistic approach is grounded upon the assumption that the storage controller can flush the writeback cache contents in out of order manner and the contents in the writeback cache can be lost at any time due to the power crash [30].

In the wearable device, we believe that this assumption is overly stringent. Wearable device has non-removable battery and PMIC module. Due to the hardware assistance, the writeback cache contents can survive the warm crash such as OS failures [3] or kernel panic [9]. Buggy software can drain the battery very rapidly [22], but PMIC module shuts down the OS ahead enough giving sufficient time slack for flash storage to persist the contents in the writeback cache. `fsync()`and `fdatasync()`invoke the flush command to ensure that the data blocks associated with the preceding writes become durable before the host issues the following writes. On the same token, if the results of preceding writes are guaranteed to be durable, the host does not have to issue the flush command. If the results of preceding writes are guaranteed to be durable, the



(a) Energy Consumption (b) Throughput

Figure 11: Energy consumption and SQLite performance

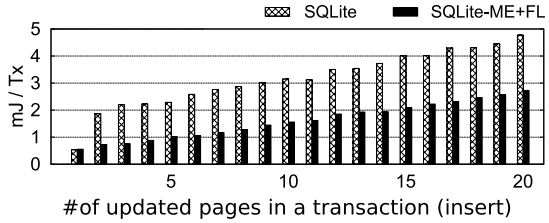


Figure 12: Energy consumption of insert transaction for varying transaction size: stock IO stack vs. energy efficient IO stack

host can issue the following writes without waiting for the results of the preceding writes to become durable. We exploit the hardware characteristics of wearable device and propose to exclude flush command in `fsync()` and `fdatasync()`. When we mount the filesystem with `nobarrier` option, EXT4 filesystem does not issue flush command in `fsync()`and `fdatasync()`.

Persisting the results of writes is expensive in terms of energy consumption. The flash controller has to program one or more flash pages every time when it receives a flush command. As a means to improve the energy consumption of wearable device, we propose to exclude the flush command in `fsync()`and `fdatasync()`and yet guarantee the durability of transaction.

6 EXPERIMENT

6.1 Experimental Setup

We implement Metadata Embedding and Selective Directory Sync in commercially available Samsung Gear S3 smartwatch model. Gear S3 has Dual-Core 1.0 GHz Exynos 7270, 512 MByte RAM, 4 GByte eMMC, and 380 mAh battery capacity. Gear S3 is operated with Tizen 2.3 platform that runs with Linux kernel 3.18. In order to evaluate the performance and energy consumption of SQLite transactions, we use Mobibench [12] as well as real IO trace. We use Monsoon power monitor to obtain the energy consumption of IO stack from Gear S3. To capture the energy consumption of IO stack, we first measure the energy consumption when the device is in idle state. Then, we measure the energy consumption of Gear S3 when the device is running various different types of IO workloads. We take the difference between the two in order to obtain the energy consumption associated with IO. We use in-house board to measure the voltage level of battery.

6.2 SQLite Benchmark

We measure the energy consumption and SQLite performance of Gear S3 smartwatch. We use Mobibench to generate the SQLite transactions. We use PERSIST journal mode and FULLSYNC option that are the default SQLite configurations in Gear S3. We examine

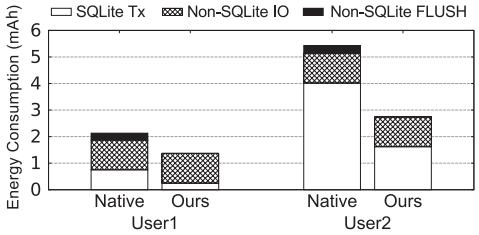


Figure 13: Daily energy consumption of IO stack for Gear S3

three different versions of the SQLite: (i) stock SQLite, (ii) SQLite with Metadata Embedding and Selective Directory Sync (SQLite-ME), and (iii) SQLite with Metadata Embedding, Selective Directory Sync, and `nobarrier` filesystem mount (SQLite-ME+FL). As described in Section 3.3, the average size of SQLite transaction is about 80 KByte in Gear S3. This corresponds to the transaction which has six database operations including insert, update, and delete. We set the Mobibench to perform six database operations in a transaction and each of operation works on the different tables. We run each of SQLite transaction 10,000 times. All experiments were repeated 5 times and then averaged.

Fig. 11(a) illustrates the energy consumption of an SQLite transaction. An insert transaction of the stock SQLite consumes 2.5 mJ. With Metadata Embedding and Selective Directory Sync (SQLite-ME), insert transaction of the modified SQLite consumes 1.4 mJ. We reduce the energy consumption of insert transaction by 43%. Next, we eliminate the flush command in `fsync()`and `fdatasync()`with `nobarrier` filesystem option. Applying all proposed techniques, we reduce the energy consumption of SQLite transaction by up to 58%. The proposed techniques not only reduce the energy consumption but also improve the SQLite performance. Fig. 11(b) illustrates the throughput of SQLite transactions. SQLite-ME improves the throughput of SQLite by 62% in insert transactions. With `nobarrier` option, SQLite-ME+FL further improves the throughput of SQLite transactions by up to 2.5×.

6.3 Effect of Transaction Size

We measure the energy consumption of an SQLite transaction. We vary the number of operations in a transaction from one to twenty. We repeat the transaction 10,000 times and take the average. As shown in Fig. 12, the optimized IO stack reduces the energy consumption per transaction from 60% to 40%. The energy saving is more significant when the number of operations in a transaction is smaller. On the average, an SQLite transaction updates three to four database pages [6]. In this case, the energy saving per SQLite transaction corresponds to 60%.

6.4 Real World Energy Saving

The vital concern is how significant the proposed techniques are from the total battery capacity's point of view. The battery capacity of Gear S3 is 380 mAh. The vendor claims that the battery lasts for three to four days. We assume that Gear S3 consumes 95 mAh of energy per day. We apply the energy consumption model to the real IO traces collected from user 1 and user 2. Fig. 13 illustrates the daily energy consumption of IO stack for Gear S3 smartwatch. In case of user 1, the energy consumption of IO activities accounts

for 2.1 mAh per day. SQLite transactions account for 35% of the energy associated with the IO activities. In case of user 2, the energy consumption of IO activities accounts for 5.4 mAh per day. User 2 creates 2.5× as many IO energy as user 1. In user 2, SQLite transactions account for 74% of the energy associated with the IO activities. According to our estimation, the energy consumption associated with the IO activities accounts for 2.2% and 5.7% of the daily energy in user 1 and user 2, respectively.

Applying our optimization techniques, we save 0.8 mAh per day in user 1. In user 2, we reduce the energy consumption associated with the IO activities by 2.7 mAh. We reduce the energy consumption associated with the IO activities by up to 50%. It is equivalent to 3% of the daily energy consumption of the smartwatch. In commercial wearable device, every individual hardware and software components are highly optimized to minimize the energy consumption. There is barely any room for further reduction in the energy consumption. We believe that saving 3% of total energy consumption is significant since it is achieved solely via software optimization and since it does not require any hardware assistance nor does it accompany any performance overhead.

7 RELATED WORK

Energy consumption of smartphones has been extensively studied since smartphones have emerged [2, 4, 10]. Compared to the other hardware components such as CPU, GPU, network, and display, the energy consumption of storage has not been received much attention. Li *et al.* [18] showed that storage software may consume as much as 200 times more energy than storage hardware. Mohan *et al.* [23] proposed a model to obtain the energy consumption of IO operation via taking the difference between the energy consumption associated with the different workloads. Recently, a few works have focused on the energy consumption of wearable devices. Huang *et al.* [11] proposed a fast storage system based on battery-backed RAM to increase the performance and battery life of wearables. Liu *et al.* [21] analyzed the usage patterns, energy consumption, and network traffic of smartwatch. Liu *et al.* [20] conducted an in-depth analysis of wearable OS in terms of four key aspects including CPU usage, idle episodes, thread-level parallelism, and micro-architectural behaviors.

Android IO stack has been under intense research for the past few years [6, 16]. Jeong *et al.* [12] found the cause for anomalous IO behavior of the Android IO stack, Journaling of Journal. A fair amount of works has been dedicated to resolve the Journaling of Journal anomaly, e.g. Delta Journaling [13], Write-Ahead Logging with direct IO [17], multi-version B-tree [15], and NVRAM Write-Ahead Logging [14].

8 CONCLUSION

Battery life is one of the most crucial aspects of battery-powered mobile devices, especially for wearables. In this work, we found that storage IO activities account for non-trivial amount of energy consumption in wearables. To extend the battery life of wearable device, we developed a set of techniques including Metadata Embedding, Selective Directory Sync, and Flushless Durability Guarantee. We demonstrated that the proposed IO stack improves the battery life of wearables by 3% solely via software optimization.

REFERENCES

[1] Apple. 2018. Apple watch3 specification. https://support.apple.com/kb/SP766.
[2] Aaron Carroll and Gernot Heiser. 2010. An Analysis of Power Consumption in a Smartphone. In *Proc. of USENIX ATC*.
[3] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. 1996. The Rio File Cache: Surviving Operating System Crashes. In *Proc. of ASPLOS*.
[4] Xiaomeng Chen, Ning Ding, Abhilash Jindal, Y Charlie Hu, Maruti Gupta, and Rath Vannithamby. 2015. Smartphone energy drain in the wild: Analysis and implications. *ACM SIGMETRICS Performance Evaluation Review* 43, 1 (2015), 151–164.
[5] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic Crash Consistency. In *Proc. of ACM SOSP*.
[6] Tuan Quang Dam, Seungyong Cheon, and Youjip Won. 2016. On the io characteristics of the sqlite transactions. In *Proc. of IEEE/ACM MOBILESoft*.
[7] Stuart Dredge. 2014. Apple Watch: battery life a challenge for a round-the-clock health tracker. The Guardian. https://www.theguardian.com/technology/2014/sep/10/apple-watch-battery-life-health-tracker.
[8] Samsung Electronics. 2016. Samsung Gear S3 Highlight. http://www.samsung.com/global/galaxy/gear-s3
[9] Weining Gu, Zbigniew Kalbarczyk, K Iyer, Zhenyu Yang, et al. 2003. Characterization of linux kernel behavior under errors. In *Proc. of IEEE DSN*.
[10] Yao Guo, Chengke Wang, and Xiangqun Chen. 2017. Understanding Application-Battery Interactions on Smartphones: A Large-Scale Empirical Study. *IEEE Access* 5 (2017), 13387–13400.
[11] Jian Huang, Anirudh Badam, Ranveer Chandra, and Edmund B Nightingale. 2015. WearDrive: Fast and Energy-Efficient Storage for Wearables.. In *Proc. of USENIX ATC*.
[12] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O Stack Optimization for Smartphones. In *Proc. of USENIX ATC*.
[13] Junghoon Kim, Changwoo Min, and Young Eom. 2014. Reducing excessive journaling overhead with small-sized NVRAM for mobile devices. *IEEE Transactions on Consumer Electronics* 60, 2 (2014), 217–224.
[14] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. *ACM SIGOPS Operating System Review* 50, 2 (2016), 385–398.
[15] Wook-Hee Kim, Beomseo Nam, Dongil Park, and Youji Won. 2014. Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split. In *Proc. of USENIX FAST*.
[16] Kisung Lee and Youjip Won. 2012. Smart layers and dumb result: IO characterization of an android-based smartphone. In *Proc. of ACM EMSOFT*.
[17] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. 2015. WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly. In *Proc. of USENIX ATC*.
[18] Jing Li, Anirudh Badam, Ranveer Chandra, Steven Swanson, Bruce Worthington, and Qi Zhang. 2014. On the Energy Overhead of Mobile Storage Systems. In *Proc. of USENIX FAST*.
[19] Eunryoung Lim, Seongjin Lee, and Youjip Won. 2015. Androtrace: Framework for Tracing and Analyzing IOs on Android. In *Proc. of ACM INFLOW*.
[20] Renju Liu and Felix Xiaozhu Lin. 2016. Understanding the characteristics of android wear os. In *Proc. of ACM MobiSys*.
[21] Xing Liu, Tianyu Chen, Feng Qian, Zhixiu Guo, Felix Xiaozhu Lin, Xiaofeng Wang, and Kai Chen. 2017. Characterizing Smartwatch Usage in the Wild. In *Proc. of ACM MobiSys*.
[22] Hao Luo, Lei Tian, and Hong Jiang. 2014. qNVRAM: quasi non-volatile RAM for low overhead persistency enforcement in smartphones. In *Proc. of USENIX HotStorage*.
[23] Jayashree Mohan, Dhathri Purohith, Matthew Halpern, Vijay Chidambaram, and Vijay Janapa Reddi. 2017. Storage on Your SmartPhone Uses More Energy Than You Think. In *Proc. of USENIX HotStorage*.
[24] Seth Proges. 2015. These 4 Challenges Could Keep Smartwatches From Succeeding. https://www.forbes.com/sites/sethporges/2015/02/25/these-are-the-4-challenges-keeping-smartwatches-from-succeeding
[25] Reza Rawassizadeh, Blaine A. Price, and Marian Petre. 2015. Wearables: Has the Age of Smartwatches Finally Arrived? *Commun. ACM* 58, 1 (2015), 45–47.
[26] Monsoon Solutions. 2018. Monsoon High Voltage Power Monitor. https://www.msoon.com/online-store
[27] Sqlite.org. 2012. Power Safe Overwrite. https://sqlite.org/psow.html
[28] Statista. 2018. Smartwatch unit sales worldwide from 2014 to 2018 (in millions). https://www.statista.com/statistics/538237/global-smartwatch-unit-sales
[29] Wired. 2018. The best smartwatches for Android and iPhone. https://www.wired.co.uk/article/best-smartwatch-for-android-iphone-android-wear
[30] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. 2018. Barrier-enabled IO stack for flash storage. In *Proc. of USENIX FAST*.