Differential Write-Conscious Software Design on Phase-Change Memory: An SQLite Case Study

SUNGKWANG LEE, POSTECH
TAEMIN LEE, Seoul National University
HYUNSUN PARK, POSTECH
JUNWHAN AHN and SUNGJOO YOO, Seoul National University
YOUJIP WON, Hanyang University
SUNGGU LEE, POSTECH

Phase-change memory (PCM) has several benefits including low cost, non-volatility, byte-addressability, etc., and limitations such as write endurance. There have been several hardware approaches to exploit the benefits while minimizing the negative impact of limitations. Software approaches could give further improvements, when used together with hardware approaches, by taking advantage of write behavior present in the program, e.g., write behavior on dynamically allocated data, which is hardly captured by hardware approaches. This work proposes a software design methodology to reduce costly PCM writes. First, on top of existing hardware approach such as Flip-N-Write, we advocate exploiting the capability of PCM bit-level differential write in the software by judiciously reusing previously allocated memory resource. In order to avoid wear-out incurred by the reuse, we present software-based wear-leveling methods that distribute writes across PCM cells. In order to further reduce PCM writes, we propose identifying data, the loss of which does not affect the functionality of the underlying software, and then diverting write traffic for those data items to volatile memory. To evaluate the effectiveness of these methods, as a case study, we applied the proposed methods to the design of journaling in SQLite, which is an important database application commonly used in smartphones. For the experiments, we used an in-house PCM-based prototype board. Our experiments with four representative mobile applications show that the proposed design methods, which is applied on top of the hardware approach, Flip-N-Write, result in 75.2% further reduction in total bit updates in PCM, on average, without aggravating wear-out compared with the baseline of PCM-based journaling, which is based only on the hardware approach. Also, the proposed design methods result in 49.4% reduction in energy consumption and 52.3% reduction in runtime compared to a typical FIFO management of free resources.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-based Systems**]: Real-Time and Embedded Systems; B.3.1 [**Semiconductor Memories**]: Phase Change Memory

---

This work is supported by Research Resettlement Fund for the new faculty of Seoul National University, the IT R&D program of MKE/KEIT (No. 10041608, Embedded System Software for New Memory based Smart Devices) and SK Hynix.
Authors' addresses: S. Lee, Department of Electrical Engineering, POSTECH, 77 Cheongam-ro, Nam-gu, Pohang-si, Gyeongsangbuk-do 37673, Korea; email: gwangyi@postech.ac.kr; T. Lee, Department of Electrical and Computer Engineering, Seoul National University, 1 Kwanak-ro, Kwanak-gu, Seoul 08826, Korea; email: taemin.lee@snu.ac.kr; H. Park, Department of Electrical Engineering, POSTECH, 77 Cheongam-ro, Nam-gu, Pohang-si, Gyeongsangbuk-do 37673, Korea; email: lena0911@postech.ac.kr; J. Ahn, Department of Electrical and Computer Engineering, Seoul National University, 1 Kwanak-ro, Kwanak-gu, Seoul 08826, Korea; email: junwhan@snu.ac.kr; S. Yoo (corresponding author), Department of Computer Science and Engineering, Seoul National University, 1 Kwanak-ro, Kwanak-gu, Seoul 08826, Korea; email: yeonbin@snu.ac.kr; Y. Won, Division of Computer Science and Engineering, Hanyang University, 222 Wangsimni-ro, Seongdong-gu, Seoul 04763; email: yjwon@hanyang.ac.kr.

DOI: http://dx.doi.org/10.1145/2842613

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Phase change memory, differential write, wear leveling, SQLite

ACM Reference Format:
Sungkwang Lee, Taemin Lee, Hyunsun Park, Junwhan Ahn, Sungjoo Yoo, Youjip Won, and Sunggu Lee. 2016. Differential write-conscious software design on phase-change memory: An SQLite case study. ACM Trans. Des. Autom. Electron. Syst. 21, 3, Article 47 (April 2016), 25 pages.
DOI: http://dx.doi.org/10.1145/2842613

1. INTRODUCTION

Phase-change memory (PCM) provides important benefits such as scalability (better than dynamic random access memory, DRAM), non-volatility, short latency, and byte-addressability (unlike the page-level access granularity of Flash memory). Among several possibilities of PCM application, e.g., storage, main memory, and cache, its application to main memory is considered to be the most promising one due to the lower area cost (more than 2X lower cost than DRAM) and standby power (due to non-volatility). Lee et al. [2009] presented a PCM-only main memory architecture to replace conventional main memory based on DRAM. Qureshi et al. [2009a] proposed a hybrid DRAM and PCM main memory structure and showed the benefit of PCM usage on the main memory, i.e., higher program performance due to reduced disk accesses and lower system energy consumption due to reduced runtime and low standby power than in the system with conventional DRAM-only main memory under a similar memory cost. Zhou et al. [2009] presented a tiered DRAM and PCM main memory to reduce PCM writes.

PCM subsystems on main memory buses are promising due to their broad applicability, covering both main memory (e.g., for use as background main memory [Qureshi et al. 2009a] or tiered main memory [Zhou et al. 2009]) and a portion of storage (e.g., for use with journaling or memory page swapping [Fang et al. 2011]). However, in such systems, write-related problems (especially write endurance) become critical issues since such systems are expected to experience more frequent writes than in PCM storage-only applications [Akel et al. 2011; Kim et al. 2014].

There are several hardware-based approaches that address write endurance to reduce bit updates by differential write [Yang et al. 2007] and data coding (including Flip-N-Write [Cho and Lee 2009]) and maximum bit updates (which usually determine PCM lifetime) by evenly distributing writes across PCM cells in wear leveling [Qureshi et al. 2009b; Seong et al. 2010; Yoon et al. 2012; Liu et al. 2013; Zhao et al. 2014]. Such hardware approaches are effective in mitigating the write endurance problem. However, they have a significant limitation in that the program behavior is not fully exploited to address the write endurance problem. It is critical to make the best use of the data write behavior in the software program running on the PCM. For instance, in the case of wear leveling, when writes are concentrated on a small amount of hot data, hardware approaches have a clear limitation in evenly distributing writes since they do not utilize exact write behavior, which could be gathered by the software program. Instead, hardware approaches mostly rely on past history, e.g., write counts, in order to predict future write behavior [Yoon et al. 2012]; such a prediction often fails to fully capture the write behavior of program. This problem becomes critical especially when frequent dynamic memory allocations and de-allocations are performed in the software program where the physical address-based information of past write history, which hardware approaches rely on, is likely to be useless.

In this article, we propose a software-based approach that, when used together with hardware approaches, can give further reductions in the write overhead of PCM subsystems on main memory than when only hardware approaches are applied. For this purpose, we first investigated the characteristics of real Gb-capacity PCM chips based on a LPDDR2 interface, which is compatible with a typical main memory bus.

Based on this study, we developed software design methods that carefully consider the underlying hardware features to minimize costly PCM writes. First, we propose to design software that is cognizant of a well-known hardware approach, namely, bit-level differential write,[1] which is unexploited and under-estimated in previous software-based approaches. Differential write is a hardware mechanism that writes only the updated portion of data to the memory. The granularity can vary from a dirty page, cache block, and word to a single bit. For instance, the real PCM chips, used in our experiments, support bit-level differential write, whereby a single bit update in a large write, e.g., a page write, requires only one PCM cell to be programmed [Chung et al. 2011]). To the best of the authors' knowledge, none of the previous *software*-based approaches are aware of this feature in addressing write-related problems. Second, we present a software-level approach that identifies data that do not need to be persistent and stores such data in volatile DRAM thereby reducing PCM writes. Our experiments show that our proposed approach can guarantee the same level of durability with regards to system failure while further reducing the total number of PCM writes.

In this article, we make the following contributions:

—We propose bit-level differential write-conscious software design methods, which utilize write behavior of software program and reuses previously allocated resources to reduce bit updates in the case of overwrites.
—Since too much resource reuse might increase wear-out, we also propose wear-leveling-conscious data structures to avoid aggravating wear-out even with the proposed resource reuse optimization.
—We propose to identify a minimal set of data that is required to be persistent for system consistency and store only those data in PCM to minimize PCM writes.
—We performed a case study of the proposed approach using SQLite [Hipp and Kennedy 2010], which is widely utilized in smartphones, and evaluated its effectiveness on a real PCM-based prototype by comparing both hardware-only approaches (differential writes and Flip-N-Write) and our software/hardware approach

The rest of this article is organized as follows. Section 2 gives background. Section 3 explains our proposed software-based approach. Section 4 introduces our PCM-based prototype. Section 5 describes the SQLite design for PCM. Section 6 shows experimental results. Section 7 reviews related work. Section 8 concludes this article.

2. BACKGROUND: PHASE CHANGE MEMORY

PCM stores information by changing the physical state of memory cells. Figure 1 illustrates a PCM cell. It is composed of GST (germanium-antimony-tellurium) material, a heater, and two electrodes. In order to store a bit in the cell, a current is applied to the cell. This makes the heater generate heat due to Joule heating, thereby melting the GST material at a high temperature (up to 600°C). The amount or the chemical state of the melted area (the shaded hemisphere in the figure) determines the cell resistance. The resistance of a cell represents the stored data, which can be either logical '0' (high resistance) or logical '1' (low resistance).

The states (i.e., high- and low-resistance states) can be controlled using several mechanisms including current pulse width, time, slope, etc. Typically, the reset (set) operation requires a short (long) and high (low) amplitude current pulse. Since the resistance can be controlled by the programming method, more than two levels of states (resulting in a multi-level cell (MLC) PCM) can be achieved by subdividing the resistance ranges of a cell and employing sophisticated programming and read schemes. Due to the mechanism used with PCM writes, referred to as Joule heating

[1]Note that a hardware approach called Flip-N-Write [Cho and Lee 2009], which is adopted in the real PCM chips used in our experiments, is also based on differential writes.

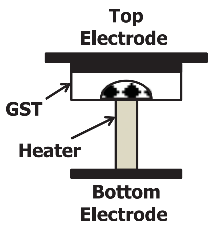

Fig. 1. A PCM cell.

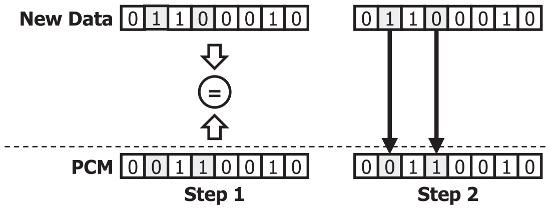

Fig. 2. Bit-level differential write.

(i.e., high temperature-based cell programming), each PCM cell can undergo only a limited number of writes before it is permanently damaged, i.e., worn out.

One way of reducing PCM writes is to use differential write, as illustrated in Figure 2. In this mechanism, each write operation is preceded by a read operation to the existing data and subsequent comparison of the old (existing) data and the new data in a bit-by-bit manner (step 1). Then, only modified bits (shaded rectangles in the figure) are written to the PCM cells (step 2). This approach improves both wear out and power consumption of PCM, since a large amount of power is consumed during cell programming operations. Due to these benefits, bit-level differential write has already been adopted in real PCM chips [Chung et al. 2011].

In terms of performance, the read latency of PCM is comparable (3–5x slower) to that of DRAM. However, its write latency is much longer than that of DRAM, which is due to the following two factors. First, cell program latency itself is long, typically on the order of 100–200ns (up to 10x longer than DRAM). Second, a write operation for the given data (e.g., 512-bit data) is usually split into multiple writes to small-sized chunks (e.g., 32 16-bit chunks) since the high power consumption of PCM cell programming and the given peak power constraint of PCM chip limit the maximum number of bits to be programmed at a time. Consequently, the total write latency becomes proportional to the amount of data to be written. This makes small writes (writes that modify a small number of bits) more desirable in PCMs [Akel et al. 2011] unlike the situation with Flash memory, where writes are performed at a page level.

3. DIFFERENTIAL WRITE-CONSCIOUS SOFTWARE DESIGN METHODOLOGY

As discussed in the previous section, PCM exhibits high write overhead in terms of wear out, power, and performance. Although PCM devices provide several features to mitigate this overhead, software design for such devices has not taken these features into account yet. In this section, we propose two strategies of software design to minimize writes to PCM on a memory bus by utilizing the characteristics and features of PCM.

3.1. Strategy 1: Reuse Previously Allocated Resources to Exploit Bit-Level Differential Write

We propose to reuse previously allocated resources, e.g., page frames, in order to reduce bit updates. For instance, let us assume a case where a database page is frequently written to append small new data to the existing page. In such a case, if a new page frame is allocated every time for the data update, as is the case in real database [Hipp and Kennedy 2010], the contents of entire data page is written to the new page frame, thereby incurring a large number of bit updates. Note that hardware- only approaches hardly utilize such a write behavior related with dynamic memory allocation, which is popular in the software program. This is because when a new page frame is allocated every time, the new page frame has a new physical address that is different from that of the previously allocated page frame. Unfortunately, this makes the information of write count, gathered by the hardware approach for the previously allocated page frame, useless. Our idea is to make dynamic memory allocation aware of differential writes and reuse the previously allocated page frames. Such a differential write-aware memory allocation enables reducing bit updates on the allocated page frames since they still contain previous data and the new data can be written to them through bit-level differential write.

Reuse can provide significant reductions in bit updates as will be shown in our experiments. However, too much reuse may cause some PCM cells to wear out earlier than the others, thereby shortening the lifetime of PCM devices, mainly because of the following two reasons. First, if some page frames receive a higher amount of writes than the others, reusing resources concentrates PCM writes to the cells that store such hot page frames. Second, this situation is aggravated if the bit-change distribution of a frequently written page is highly skewed. In order to address these two issues, we propose to limit the maximum number of reuse per resource to control the impact of reusing resources for hot page frames on the wear-out behavior. In addition, we design and apply wear-leveling-conscious data structures to mitigate wear-out problems caused by frequently updated 1-bit flags and data words (e.g., frequently written state flags, counters, checksums, etc.). In Section 5, we will discuss how to design frequently updated data structures to be amenable to wear leveling.

3.2. Strategy 2: Minimize the Amount of Non-Volatile Data to Reduce PCM Writes

In order to minimize the maximum bit updates in the PCM, we aim at reducing the amount of non-volatile data. To do that, we rethink the definition of non-volatile data structures. More specifically, we propose to differentiate the data recoverable from the non-volatile data, which we call *quasi-non-volatile data*, from ordinary non-volatile data. Examples of quasi-non-volatile data include the number of entries in a non-volatile linked list and the presence bits for the allocation of non-volatile blocks. For instance, in the former case, even if the size information is lost due to system crash, it can be re-obtained by scanning the non-volatile linked list during recovery. In order to optimize writes to such data, we store them in volatile memory (i.e., DRAM) in the hybrid DRAM/PCM main memory instead of writing them to PCM. Although this makes such data volatile, our approach does not harm the durability of the data at all since, in the case of system crash, they can be recovered during the recovery process by re-calculating them from the non-volatile data. Moreover, the cost of such re-computation is negligible in practice since system crashes occur very rarely.

In addition to the identification of quasi-non-volatile data, we also propose to store non-functional data into volatile memory. Non-functional data are defined as data that do not require recovery from a system crash since they do not affect the functionality in case of loss while possibly degrading performance or wear out. As wil be shown in Section 5, a representative example is a block use counter that is used to limit the reuse of resources for wear-leveling purposes. Considering that system crashes are extremely rare events, keeping non-functional data in volatile memory can further reduce write traffic to PCM without a noticeable impact on overall performance and wear-out.

Our proposed strategies will be effective in software programs in which small overwrites to dynamically allocated data are frequent. Database programs and their journaling behavior are especially characterized by such behavior, i.e., frequent small overwrites. Thus, the first target domain of applying our method will be database applications such as MySQL InnoDB [Tuuri and Sun 2009], PostgreSQL [PostgreSQL 2015], MongoDB [Chodorow 2012], Berkeley DB [Olson et al. 1999], Cassandra [DataStax 2015], HBase [Lehene 2010], MariaDB [MariaDB Corporation 2015] and SQLite which is utilized in our case study. In addition to database programs, journaling is another important target of our method. Journaling is popular in current file systems,

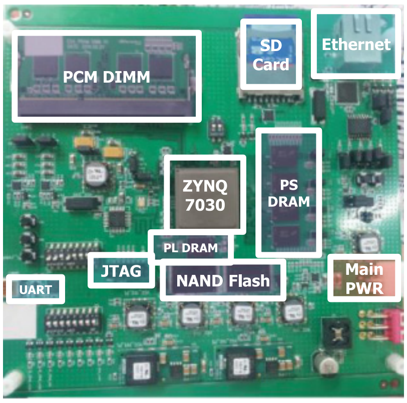

Fig. 3. PCM-based prototype board.

Table I.. Summary of Board Specifications

Core	ARM Cortex-A9 dual core, 667MHz
Cache	L1 I/D = 32KB/32KB, L2 = 512KB
PS-DRAM	256MB×4, DDR3, 1066MHz, 8b×4, tRCD = 15ns, tCL = 15ns, tRP = 15ns
PL-PCM	128MB×4, LPDDR2-N, 150MHz, 16b×4, tRCD = 80ns, tCL = 40ns, tRP = 13ns, 2KB program buffer
SD card	Class 10 Transcend SD card

e.g., ext4 [The Linux Kernel Organization 2015], NTFS [Microsoft TechNet 2015] and HFS+ [Apple 2015] as well as in the database programs. Thus, our methods will be effective when software programs, having frequent small overwrites on files, run on such journaling file systems.

Section 5 presents a case study where our strategies are applied to an SQLite design implemented on a PCM-based prototype (to be explained in Section 4).

4. PCM-BASED PROTOTYPE SYSTEM

Figure 3 shows our PCM-based prototype system [Lee et al. 2014]. It consists of a motherboard (with a Xilinx Zynq XC7Z030 chip, on-board DRAM chips, SD card, and peripherals) and a PCM dual in-line memory module (DIMM). The Xilinx Zynq field-programmable gate array (FPGA) chip consists of a processing system (PS) that uses an ARM Cortex A9 processor (with a dual-core CPU and L2 cache) and programmable logic (PL). The DRAM chips are connected to the PS while the PCM DIMM is connected to the PL. Both PS and PL are connected via the main AXI crossbar. To be exact, the PCM controller on the PL is connected to the crossbar as a slave and the ARM processor on the PS is connected to the crossbar as a master. Thus, the software program running on the ARM processor can access the PCM via the PCM memory controller. There are two ways of communication between the DRAM and the PCM, direct memory access (DMA) and processor intervention (load and store instructions). A version of Linux (version 3.14.0-xilinx) runs on the ARM processor. Table I summarizes the specification of our prototype system.

The PCM DIMM (x64) consists of four x16 PCM chips of 1Gb each [Chung et al. 2011]. We developed an in-house PCM controller conforming to an industry-standard memory interface protocol, referred to as *LPDDR2-N* [JEDEC Standard 2011]. A key feature of this protocol is that it uses separate read and write paths. The read path behaves like a DRAM interface. Thus, given a read address from the CPU (or the L2 cache), the PCM

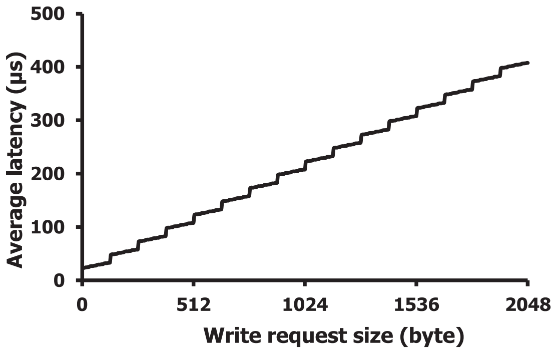

Fig. 4. Write latency vs. write data size.

controller first issues row pre-activation (PREACT) and activation (ACT) commands to the PCM. After a latency of a few clock cycles (defined in the LPDDR2-N protocol as *tRP* for PREACT and *tRCD* for ACT), it initiates a read operation by issuing a read command to the PCM. Then, after an additional read latency *tCL*, the data can be read via the memory chip's data I/O pins. On the other hand, the write path behaves like Flash memory. First, the PCM controller writes the data to a 512-byte program buffer in the PCM chip. Then, it writes a write command to a specific physical address (in an address range called the overlay window). Upon receiving this command, the PCM starts cell programming. As mentioned in Section 2, the peak power limit forces programming to be done in a chunk-by-chunk manner, thereby making write latency proportional to the total amount of data written.

In our PCM-based prototype, software applications can access the PCM in two ways, cacheable and non-cacheable. In the case of cacheable accesses, the PCM data blocks are cached in the L1 and L2 cache. Cacheable access has the benefit of reduced read latency on cache hits. However, the miss penalty becomes difficult to estimate. In the worst case, a read miss in the L2 cache can be delayed by dirty block eviction to the PCM, which can take tens of microseconds. Non-cacheable accesses are implemented with *mmap()*, which allows us to access the physical address region of PCM. Non-cacheable accesses are slow compared with cacheable ones as every access has to access the PCM. Thus, it is often necessary to copy data from the PCM to the DRAM (cacheable region) to access them with low latency. In such a case, the dirty data in the DRAM needs to be written back to the PCM later by software. The choice of cacheable or non-cacheable PCM accesses may depend on system requirements – e.g., real-time constraints, fast checkpoints, etc. In our case study, we used non-cacheable PCM access since volatile caches complicate consistency and endurance management.

Figure 4 shows the latency of sequential write to the PCM chips measured on the prototype system. In order to avoid interference from Linux background processes, we developed a hardware module that generates write requests to the PCM controller. As the figure shows, the write latency is proportional to the amount of data to be written. In other words, if the PCM device supports differential write (like the one used in our evaluations), writes with small differences (i.e., the number of bits changed from the original data) take a shorter time to complete. In addition, small writes (e.g., 128B writes) result in lower latency than with typical Flash memory (with a latency of ~1ms per page write). Note that the latency increases in a stepwise manner (with every 128B of data) since each PCM chip provides constant latency for write data size smaller than 32B (chunk size) and the DIMM is equipped with four PCM chips.

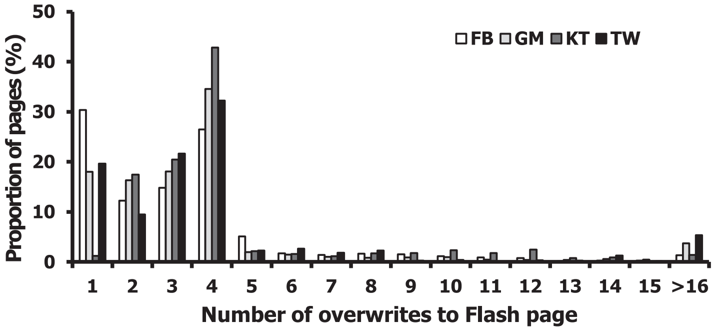

Fig. 5. Overwrite characteristics.

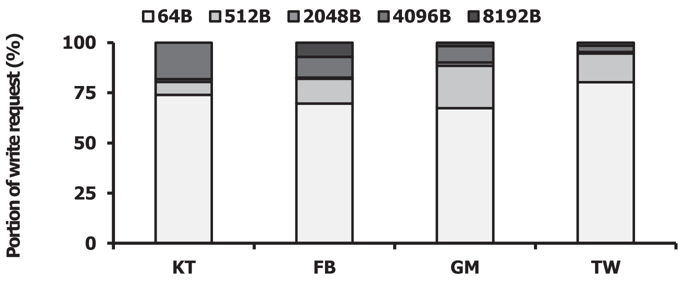

Fig. 6. Page-level data difference statistics.

5. SQLITE JOURNAL DESIGN ON PCM

This section describes the case study used to evaluate the effectiveness of the proposed software design methodology when applied to our PCM-based prototype board. We chose SQLite journal since SQLite is widely used software upon which numerous mobile applications are based and its journaling operation exhibits representative behavior, namely, frequent overwrites and small writes, suitable for us to show the effectiveness of the strategies.

5.1. SQLite Write Behavior Analysis

Figure 5 shows the characteristics of overwrites on logical pages in SQLite. We obtained this from four mobile applications: Facebook (FB), Gmail (GM), Kakaotalk (KT), and Twitter (TW) on a real smartphone using a rollback journal in delete mode (see Section 6 for the detailed experimental setup). The figure shows that most of the logical pages receive multiple writes (i.e., average 5.2 overwrites per page).

Figure 6 shows the amount of data difference between two consecutive writes to a logical page. As shown in the figure, on average, 72.8% of page writes update only 64 bytes. It shows that small writes dominate SQLite writes. This is mainly because (1) mobile applications are user-interactive, and thus, a transaction in SQLite tends to insert or update a small amount of data, e.g., short message in Twitter, to the database page in the storage and (2) journaling can duplicate such data writes to maintain consistency. In summary, Figures 5 and 6 show that a significant portion of SQLite writes are overwrites with small differences.

5.2. Modified SQLite Journal Design for PCM

In our case study, we applied the two strategies explained in Section 3 to the SQLite journaling function, which incurs frequent and small overwrites. In order to apply these strategies, we modified the public source code of SQLite [Hipp and Kennedy 2010].

Terminology. In the following, we use these terms.

—*Page frame* is a unit of PCM resource allocation and corresponds to a memory region of 64KB in our experiments.
—*Portion* is a unit of resource allocation for a DB page. Since the size of a DB page (1KB in our experiments) is smaller than 64KB, a page frame can contain multiple portions.
—*Three states of frames and portions*. When a page frame (or a portion) is in use, it is in *active* state. When it is no longer used, but can be re-used for a future allocation of the same DB page, it is in *reuse* state. When it cannot be re-used for a future allocation of the same DB page, it is in *invalid* state. Detailed definitions of three states for frames/portions will be given below.

Overview. Figure 7 compares the original and our modified SQLite designs. As Figure 7(a) shows, in the original design, both the database (DB) and journal files are stored in the Flash memory storage. As Figure 7(b) shows, our modified design implements the journal on the PCM. Contrary to the original SQLite that utilizes a journal file, our PCM-based journal design does not utilize a file for journaling. Instead, it manages its own data structure consisting of *data frames*, *header frames*, and a *manager frame*. Each frame has the same size (64KB[2]) for easy management. A data frame contains the copies of DB pages (typically, 512B–64KB DB pages). In our experiments, we used DB pages of 1KB. Thus, a 64KB data frame can contain up to 64 DB pages, i.e., 64 portions.

A header frame and associated data frames correspond to a journal file in the original SQLite design. The header frame has the same metadata as in the original journal file, e.g., a hot journal flag (Hot in the figure) and a journal header structure (Header). The header frame also has an array of *data frame descriptors* which contain the same information as the original SQLite design, i.e., page number (PgNo) and checksum (CkSum) and new information of our own design, i.e., valid flag (V) and a pointer to a portion (PTR) in the data frame. Data structures such as data frame descriptors are required in our PCM-based SQLite design because, in the original design, the file system manages the storage resource for the data (called *data blocks* in the ext4 file system) in the journal file while our design is based on the PCM attached to the main memory bus and needs to manage storage resource for reuse. A data frame descriptor is linked with a portion in the data frame that has the contents of a DB page (e.g., Page #J_m in Figure 7(b)), as the arrows show.

A portion can have one of three states, active, reuse and invalid. The combination of the valid flag, V, and the pointer to a portion, PTR, in the data frame descriptor constitutes a state of the associated portion in the data frame. If the valid bit is true, then the portion is in use. If the valid bit is false and the pointer is not zero, then the state is labeled "reuse" since the associated portion can be reused (see the next paragraph). If the valid bit is false and the pointer is zero, the descriptor is invalid.

The manager frame keeps per-'page frame' presence and reusable bits to ease checking the status of resource allocation for a new page frame. The states of header and

[2]The original SQLite supports the DB page size from 512B to 64KB. In order to support the largest DB page, we use the page frame of 64KB.

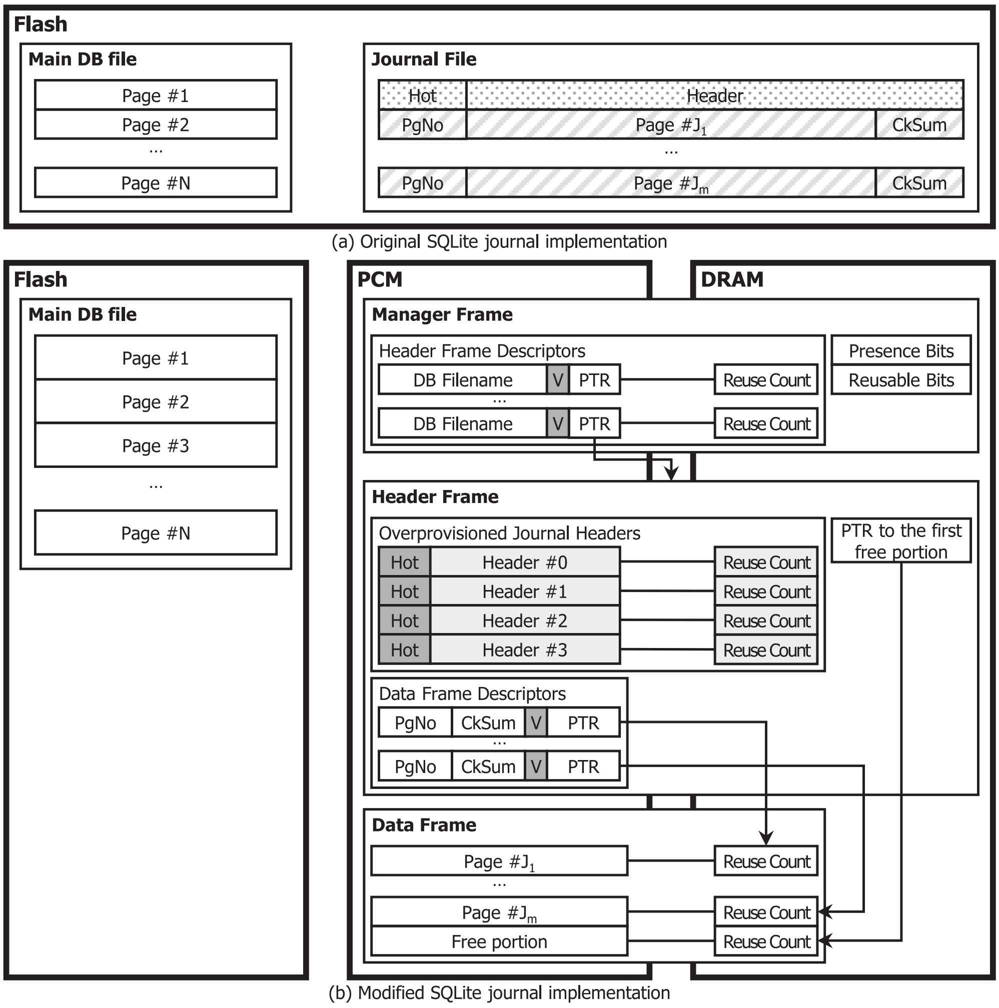

Fig. 7. Modification of SQLite journal implementation.

data frames are determined by the combination of presence and reusable bits. If both are set, then the corresponding header or data frame is in active state. If the presence bit is reset and the reusable bit is set, then it is in reuse state where the associated page frame can be allocated only for the DB file for which it was allocated. If both presence and reusable bits are reset, then it is in the invalid state and the associated page frame can be allocated for any DB file. If a header frame is in one of reuse and invalid states, then the journal for which the header frame was allocated is considered to be deleted. The manager frame also has an array of header frame descriptors, each of which points to a header frame, as shown in the figure.

Our modified design of SQLite incurs a storage overhead of two page frames for manager and header frames. Note that the storage overhead can be reduced in larger systems in that we chose the minimum size of journal to support the mobile applications used in the experiments and, for systems requiring larger journal, we still require only two additional page frames.

Page Frame Reuse. When starting a new journal to update a database file, for instance, we need to allocate a free page frame for the data frame in order to store new data (in the case of a write-ahead logging journal) or old data (rollback journal) in the journal.[3] First, we try to find the page frame that was utilized for the target DB page(s) (which we call the *associated DB page*) which needs to be updated. If such a page frame is found, we reuse a portion of that page frame utilized for the target DB page(s) (which we call the *associated portion*) in order to exploit differential write (Strategy 1).

In order to find the associated portion, we first search for a header frame descriptor (in the manager frame) that has the same DB file name. If such a header frame descriptor exists, i.e., if there is a journal that was utilized for the DB file, then the associated header frame is visited and its data frame descriptors are searched in order to find a data frame that contains a portion which was previously utilized for the associated DB page and is now in 'reuse' state. Note that the associated portion in 'reuse' state keeps the old contents of the associated DB page. If such an associated portion is found, then we reuse it to minimize the number of bit updates. If there is no such data frame containing an associated portion, one of the free page frames is allocated. If there is no available free page frame, then garbage collection is performed, where all page frames in 'reuse' state are freed by changing the state of page frame to 'invalid'. Note that garbage collection changes the state of page frame (from reuse to invalid) without deleting the contents of data frame. After that, we try again to allocate a data frame from newly freed ones. Header frames are reused in a similar way. Note that the manager frame is not reused.

As mentioned in Section 3, always reusing previously allocated resources possibly aggravates wear-out problems. In order to avoid this, our SQLite design reuses page frames as long as their reuse count does not exceed a given threshold, called the reuse limit. This alleviates the issues of increasing wear out for hot pages caused by too much reuse of page frames. Note that, upon the reuse limit, the contents of page frame is copied to a new page frame, which incurs additional writes.

Metadata in header frames tend to be more frequently updated than normal data in data frames. In our SQLite design, such hot metadata can incur wear-out problems, which in turn limits the reuse of header frames. In order to address this issue, (1) we apply wear-leveling-conscious data structures to hot non-volatile metadata, and (2) we differentiate quasi-non-volatile and non-functional metadata from non-volatile metadata and allocate them on volatile memory.

Wear-Leveling-Conscious Data Structure for Hot Non-Volatile Metadata. Among hot non-volatile metadata, 1-bit flags (e.g., valid bit in header frame descriptors, etc.) can especially incur a severe wear-out problem with frame reuse since only a single fixed-bit position (i.e., a specific PCM cell) in a word is frequently written. In order to address this issue, we propose a wear-leveling-conscious data structure for 1-bit flags called an XOR flag, shown in Figure 8. An XOR flag uses an 8-bit word to represent a binary state (0 or 1), which is calculated by XORing all the bits in the 8-bit word. Figure 8 exemplifies the reset and set operations of an XOR flag. For instance, in Figure 8(a), the binary state of the word on top is '1' since the word has three 1's. In order to reset the binary state to '0', we shift the bits toward the left and insert the inverted form of the previous most significant bit value into the least significant bit position. Note that the XOR flag still updates only one bit (shaded in dark gray) as with the original 1-bit flags. The benefit of XOR flags is that they are inherently wear-leveling-friendly since all the bits in a word are evenly updated. We utilize the XOR flag

[3]Note that the original SQLite design utilizes a journal file to write journal data.

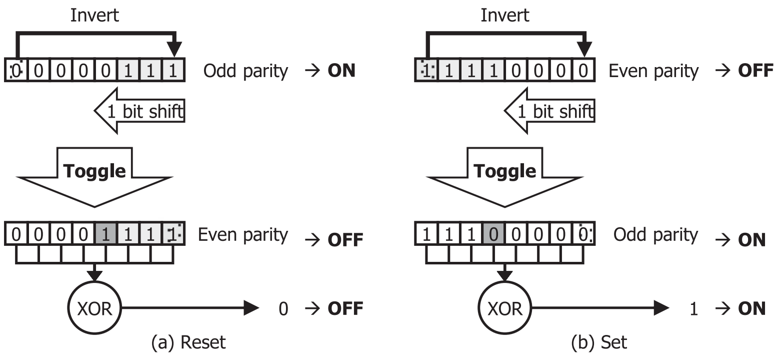

Fig. 8. XOR flags.

for hot journal flags in header frame descriptors and valid bits in data framedescriptors (dark grey rectangles in Figure 7(b)).

For the frequently written non-binary metadata (e.g., data frame descriptors), we supply overprovision resources for such data for wear-leveling purposes. Given N-1 spare resources, we allocate N times larger resources only for such frequently written metadata and write new data in a rotational manner such that each PCM cell in the overprovisioned resources receives only one write for every N writes to the metadata. Figure 7(b) shows that the resource of journal header (light grey rectangles) is overprovisioned (with three additional ones).

Minimizing Non-Volatile Data. In order to address the issue of hot metadata, we apply Strategy 2 to differentiate quasi-non-volatile and non-functional metadata from the other metadata in header and manager frames, and then divert write traffic for such data (in PCM) to volatile memory (DRAM in our case) in the hybrid DRAM/PCM main memory.

An example of quasi-non-volatile data in our SQLite journal design is the presence bits of manager frames, which represent the status of page frame allocation. The presence bits are useful in accelerating the procedure for allocating free page frames. The problem with presence bits is that, whenever a new frame is allocated and, later, de-allocated, the corresponding presence bit needs to be set and reset, i.e., written twice per allocation cycle. Thus, they receive more writes than the other metadata. In addition, the pointer to the first free portion in the data frame, which is utilized to accelerate the search operation for free portions, is considered to be quasi-non-volatile. They can be classified as quasi-non-volatile data since their information can be constructed from other non-volatile metadata during system recovery after a sudden system failure by traversing non-volatile data, i.e., header frame descriptors, and the associated data frame descriptors. Therefore, as Figure 7(b) shows, our design stores them in volatile memory (i.e., DRAM) in order to reduce PCM writes. Note that, during system recovery, the reconstruction of quasi-non-volatile data can be done in O(n) time, where n is the number of active journals at the moment of system crash. Typically, n takes a small value. In our experiments, the maximum value of n is 3, and we measured the recovery time of quasi-non-volatile data in the worst case of three active journals and found it takes a very small amount of time, 7.8ms. The recovery cost can be considered negligible since, as mentioned previously, a system crash rarely occurs and the recovery time itself is very small, as measured in our experiments.

In addition, we also identify hot non-functional metadata and store them in volatile memory for further reductions in PCM writes. Most notably, reuse counters and reusable bits, which track the frequency and possibility of resource reuse for a portion

Table II. Summary of Applying our Proposed Methods to Hot Metadata

	Manager frame	Header frame	Data frame
XOR flag	Valid bit (header frame descriptor)	Hot journal flag (journal header) Valid bit (data frame descriptor)	
Overprovisioned structure		Journal header	
Quasi-non-volatile metadata	Presence bits	Pointer to the first free portion	
Non-functional metadata	Reusable bits Reuse count (header frame descriptor)	Reuse count (journal header)	Reuse count

and a data frame, respectively, can be written frequently, but are not necessary for the functionality of SQLite since such counters and bits are used only for controlling wear-out and reuse behavior. Figure 7(b) shows that quasi-non-volatile and non-functional metadata, e.g., presence bits, reusable bits, reuse counter, and pointer to the first free portion, are mapped on the volatile memory, DRAM. Table II summarizes how our proposed methods are applied to hot metadata.

In our case study, the total amount of quasi-non-volatile and non-functional data is only 20KB. Thus, their allocation on the DRAM is not a concern. However, in more general applications, in order to reduce the utilization of DRAM resource, hot quasi-non-volatile data need to be identified and allocated on the DRAM, which needs further studies on profiling techniques to find hot write data and analysis methods to identify quasi-non-volatile and/or non-functional data, which is left for future work.

Area Overhead. In our case study of PCM-based SQLite journal design, the area overhead is mostly incurred by the overprovision resource of wear leveling-conscious data structure. As will be explained in the next section, the area overhead is small (less than 1.6% in the best configuration) in our case study.

6. EVALUATION

6.1. Experimental Setup

We used two environmental setups to evaluate the modified SQLite design: our PCM-based prototype and a PC. The PCM-based prototype was used to measure/estimate the performance/energy consumption of both our PCM-based and the original Flash-memory-based SQLite journal designs by counting execution cycles with our hardware performance monitor. We used the PC for bit update calculations since this requires a large storage area and long running time. To be specific, each bit, i.e., a PCM cell, needs a counter-variable to store the number of bit changes. On every write to the PCM, the corresponding counters are updated. Note that our PCM chip is equipped with Flip-N-Write. Thus, we assume the PCM chip uses 64-bit Flip-N-Write when calculating the bit updates of PCM cells. The same SQLite code runs on each of the PC and the PCM-based prototype. On the PC, we emulated the PCM region by allocating an address region of 2MB, which is the minimum journal size to support all the four applications used.

We estimated the energy consumption of PCM and Flash memory based on the measured execution cycles per memory power mode since the PCM-based prototype does not have current measurement capability. For the energy estimation of PCM, we used the Micron memory power calculator with the datasheet of the PCM chip [Chung et al. 2011].[4] For Flash memory, we used the energy consumption data in Mohan et al.

[4]Note that the datasheet information cannot be made public for the moment. Thus, we refer to the related paper describing the PCM chip used on our PCM-based prototype system.

Table III. Mobile Applications

Category	Application	Scenarios
Social Network Service	Facebook [FB]	1. Execute Facebook and view new messages. 2. Write a message, link, and picture. 3. Repeat Step 2 with eight different patterns.
	Twitter [TW]	1. Execute Twitter. 2. View new tweets and write a status. 3. Repeat Step 2 with eight different patterns.
	Kakaotalk [KT]	1. Execute Kakaotalk. 2. Perform 1:1 chat with emoticons, pictures, game invitations, video, and group chats. 3. Repeat Step 2 with eight different patterns.
Email	Gmail [GM]	1. Execute Gmail. 2. Write and send a message. 3. Receive a message. 4. Repeat Steps 2–3 with eight different patterns.

[2013] and write amplification factor (WAF) in Desnoyers [2012]. We used ext4 as the file system (ordered mode for journaling).[5]

Mobile applications are interactive. Thus, in order to obtain reproducible experiments, we generated and used SQLite command-level traces with real application usage scenarios on a smartphone. Table III shows the applications and their usage scenarios used to obtain the traces. For each application, we generated four traces (2,000–38,000 page writes per trace when both journal and database files are stored in the Flash memory). For the trace generation, we inserted trace generation code into the original SQLite code. Each trace consists of two kinds of files: an initial database file and an SQLite command trace file. In order to replay the traces on the PC and the PCM-based prototype, we employed our in-house trace replay program, which takes the trace files as input and processes them. The trace replay program can utilize either the original or modified SQLite program.

We make two types of comparisons. First, we evaluate our proposed methods, strategies 1 and 2, with our PCM-based SQLite journal implementations. Then, we select the best configuration of our proposed PCM-based SQLite journal implementations and compare it with the original SQLite design with journaling on the Flash memory.

Table IV shows the configurations of PCM-based SQLite journal implementation that we evaluated in our experiments. Note that our proposed method is complimentary to the hardware-based approaches of bit update reduction, e.g., Flip-N-Write. Thus, we used Flip-N-Write for all the configurations in Table IV. In the configuration called RR, resources for data and header frames are allocated in a rotational, i.e., wear-leveling-friendly manner while resources for manager frames are not dynamically re-allocated, but statically allocated. We use, as our baseline, RR to represent a typical FIFO (first-in-first-out) management of free resources, i.e., page frames under the hardware-based solution, Flip-N-Write.

6.2. Experimental Results

Figure 9 compares the total bit update reduction normalized to the baseline configuration RR. We observed that applying Strategy 1 (RU) achieves significant reductions in

[5]The original 1Gb PCM chip shows write performance in proportion to the amount of data difference [Kwon et al. 2012]. Our prototype board utilizes a newer version of the 1Gb PCM chip fabricated at more advanced technology which, however, does not show data difference-proportional write performance even though bit-level differential write is still applied. Thus, in order to show the performance impact, we implemented a software write buffer of 64-byte size on the DRAM. When a PCM write is performed, first, the associated 64-byte block is allocated in the write buffer. Then, the corresponding 64-byte data are fetched from the PCM

Table IV. Configurations of PCM-based SQLite Journal Implementation

Configuration	Description
RR	Round robin. For this, we run our modified SQLite design by setting the reuse limit to 1.
RR-S2	RR with Strategy 2 (allocation of quasi-non-volatile data and non-functional data to DRAM)
RR-S2X	RR with Strategy 2 and XOR flags
RU	Reuse for both data and header frames (our modified SQLite design on PCM under Strategy 1 without wear leveling-conscious data structures)
RU-S2	RU with Strategy 2
RU-S2X	RU with Strategy 2 and XOR flags
RUn-S2X	RU-S2X with a reuse limit of n times
RUn-S2X-WHm	RUn-S2X with wear leveling header (WH) where each header frame utilizes wear leveling-conscious fields, as shown in Table II, with m spare blocks

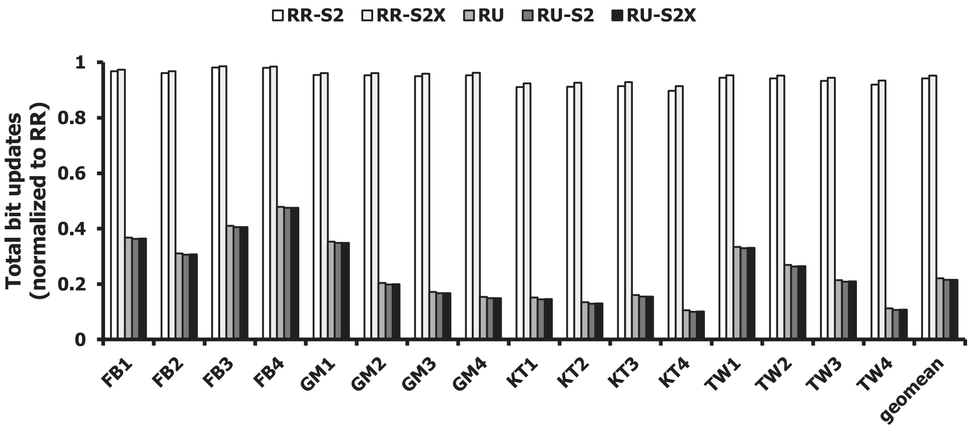

Fig. 9. Reductions in total bit updates normalized to RR.

total bit updates. Note that RU does not have a reuse limit. On top of this, Strategy 2 and XOR flags (RU-S2 and RU-S2X) provide slight additional reductions. These results show that Strategy 1 (reuse of previously allocated resources) is effective in reducing bit updates in PCM. Note that RR-S2 shows less total bit update than RR-S2X. It is due to the initialization of XOR flags. When a frame is reallocated for a header frame, XOR flags are initialized to zero, which incurs additional bit updates.

Figure 10 shows the relationship between total bit updates and the reuse limit on header and data frames. As the figure shows, a larger reuse limit enables more reduction in bit updates since each data frame can be reused up to n times in RUn. For each application, we ran four traces and numbered them in increasing order of trace sizes (e.g., FB4 represents the largest trace for the Facebook application). As the figure shows, except for FB, longer traces tend to show more reduction in PCM bit updates than shorter traces due to the increased opportunities for reuse. Among all traces, FB traces give the smallest reduction in bit updates because the Facebook application uses many SQLite DB files (13–16 files, while others use 3–6) but its trace length is not much longer than the other applications. In other words, its trace length per DB

and compared with the new write data. Later, when a new 64-byte block needs to be written, the one in the write buffer is written to the PCM chip if its contents are different from that on the PCM.

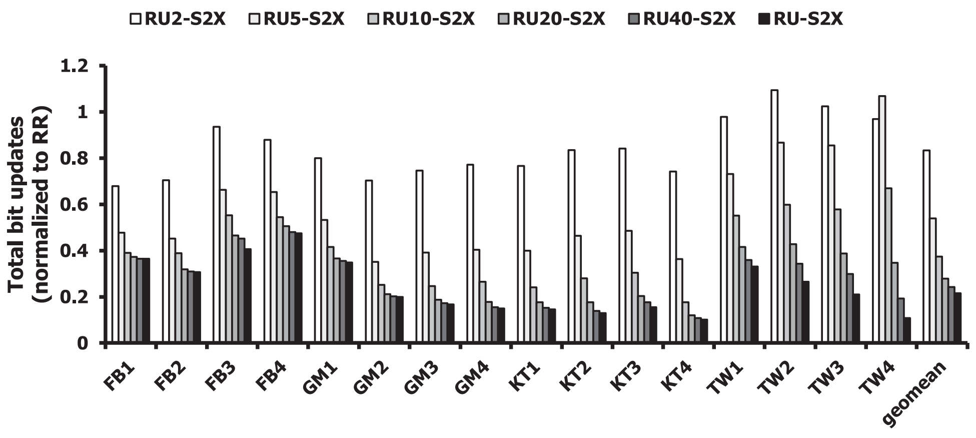

Fig. 10. Total bit updates vs. reuse limit.

file is shorter than others, which implies less reuse opportunity. Apart from FB, GM1 shows the least reduction in bit updates because, in this scenario, the user is logged into two user accounts, which requires downloading two totally different mail lists, thereby incurring resource conflicts in the journal area.

In Figure 10, in the case of TW4, RU2-S2X gives different behavior from the other traces, i.e., it gives less total bit updates than RU5-S2X. It is because the trace TW4 has a repetitive write pattern where the allocations of journal header and data frames alternate and, coincidently, in the case of RU2-S2X, the page frames previously utilized for journal header (data) are repetitively allocated for the same type, i.e., journal header (data). Thus, RU2-S2X reuses page frames more heavily thereby giving less total bit updates than RU5-S2X. By modifying our design, such a coincidence can be avoided. For instance, by adopting the incremental garbage collection (to be explained later) or slightly increasing the size of PCM resource allocated for SQLite journal, we found that RU2-S2X does not show the coincidence behavior for TW4, which means RU2-S2X gives more bit updates than RU5-S2X in the modified designs.

Figures 9 and 10 demonstrate the potential of differential write-aware dynamic memory allocation for programs having strong behavior of small overwrites. Given such programs, the underlying software (like our modified SQLite) and hardware architecture need to be able to make best use of such behavior possibly by adjusting their behavior to the program behavior in the management of free memory resource, e.g., by allocating a dedicated list of free resource to each of distinct write streams towards different journal files, which is left for future work.

Figure 11 shows the relationship between total bit updates and the reuse limit on header frames. Note that in the case that the wear leveling-conscious data structure is applied, the header frames can be reused more than the data frames. In this figure, RU20-S2X-WH4, for instance, represents the case when a data frame can be reused up to 20 times while a header frame can be reused up to 4×20 times since the header frame has three more spare resources (in addition to the original storage) for each of its hot fields. The figure shows that the wear-leveling-conscious data structure is effective in reducing total bit updates by allowing more reuse in header frames (e.g., RU20-S2X vs. RU20-S2X-WH64).

In Figure 11, only TW gives noticeable benefits of header frame reuse while the other applications show small improvements. There are two reasons for this. First, as Figure 12 shows, TW exhibits strong overwrite behavior on journal header frames in the original SQLite. Since our SQLite design is obtained by modifying the original

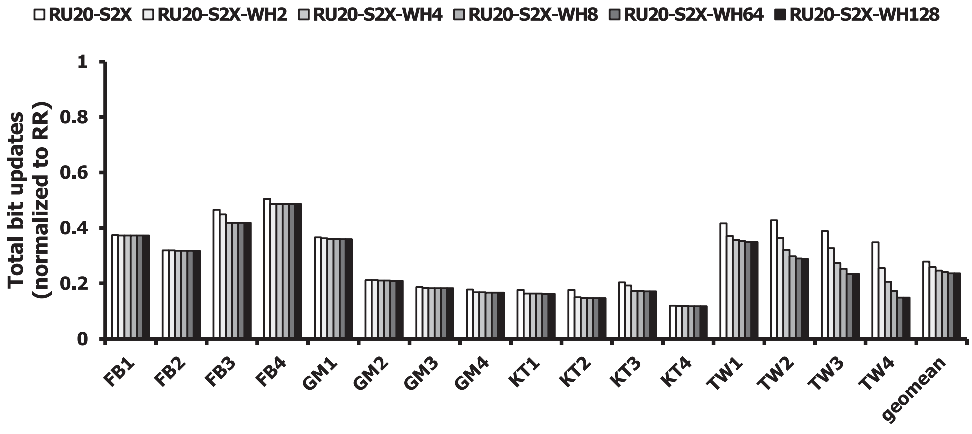

Fig. 11. Total bit updates vs. header frame reuse.

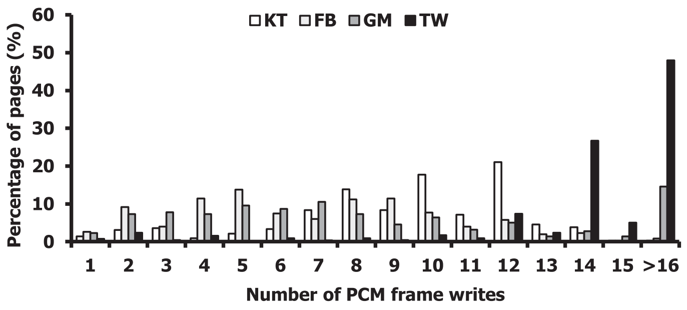

Fig. 12. Overwrites statistics for journal header pages in the original SQLite.

SQLite, it inherits the same behavior from the original one. Thus, the header frames of TW receive much more overwrites than those of the other applications, which makes TW benefit most from the header frame reuse. The second reason is that the traces of TW are relatively longer than the others, again confirming our observation that longer traces (i.e., longer usage) result in more bit update reductions (refer to Figure 10).

Figure 13 compares the maximum bit updates, which is defined as the number of bit updates for a PCM cell that receives the largest number of writes, under the various reuse methods. Since the larger maximum bit updates can indicate the shorter lifetime of PCM device, its purpose is to evaluate how much lifetime can be affected by reuse methods since reuse can result in more concentrated writes on some PCM cells. Note that the maximum bit updates in Figure 13 are normalized to those of RR-S2X.[6] As shown in Figure 13, the SQLite program adopting RR results in large maximum bit updates, which is contrary to the expectation that wear leveling, i.e., the FIFO management of free resources can resolve the problem of wear-out. This is mainly because wear-leveling is applied to data and header frames, not to the manager frame, as mentioned previously. To be specific, the presence bits in the manager frame wear-out earlier than the other fields due to frequent updates. Note that RR allocates all the journal data on the PCM. By applying Strategy 2 to RR (in RR-S2), i.e., by allocating the presence bits in volatile memory, we can significantly reduce the maximum bit updates

[6]We think RR-S2X can be considered as a 'better' baseline than RR where frequently written metadata incur severe wear-out problems as shown in Figure 2.

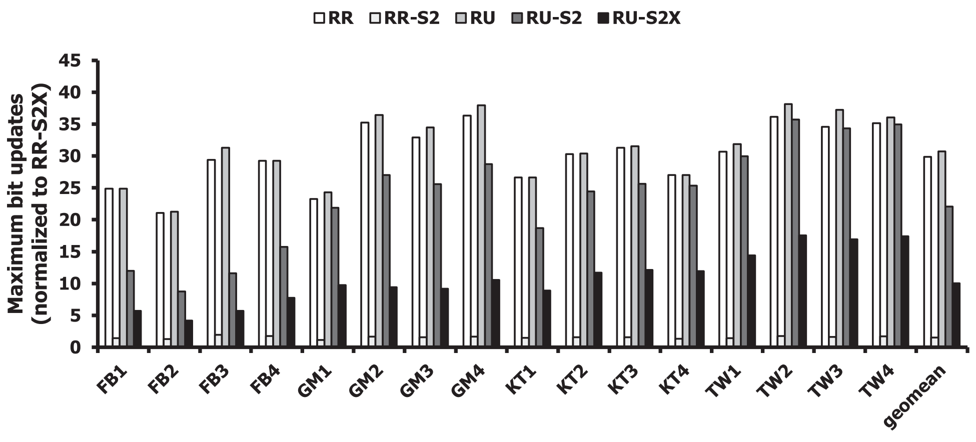

Fig. 13. Comparison of maximum bit updates.

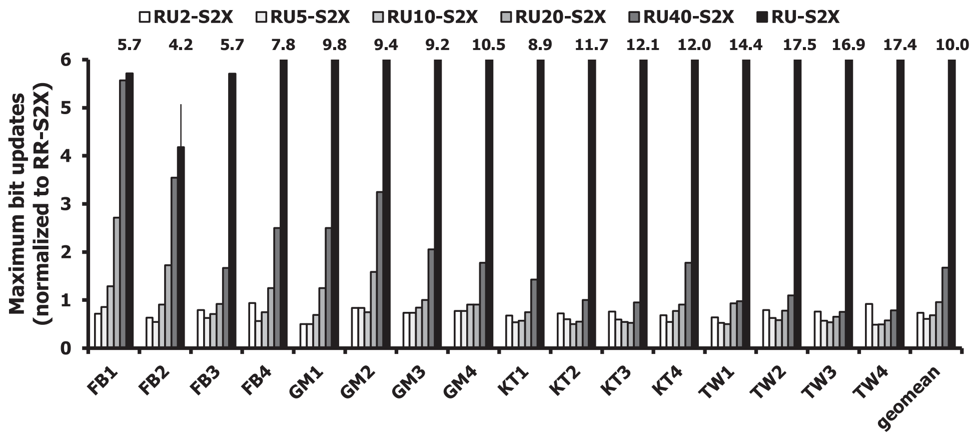

Fig. 14. Maximum bit updates vs. reuse limit.

in PCM. Figure 13 also shows that both Strategy 2 and XOR flags are very effective in mitigating the increase in maximum bit updates, which is a negative side-effect (affecting the lifetime of the PCM device) of reusing page frames (RU).

Figure 14 shows the effects of the reuse limit on maximum bit updates (less is better). As the reuse limit increases, the maximum bit updates tend to increase. However, we also observed that smaller reuse limits can result in increases in the maximum bit update. This is because a small reuse limit results in frequent copies which migrate the contents of page frames (which reached the reuse limit) to new page frames, thereby increasing the maximum bit updates.

Figure 15 shows the relationship between maximum bit updates and header frame reuse. We use the cases of RU2 since they are representative of the effects of header frame reuse. As the figure shows, the wear-leveling-conscious data structure is effective in reducing maximum bit updates for most applications. In the cases of FB1 and FB2, the effects of wear-leveling-conscious data structure is not gradual. It is because the two traces are not long enough to incur significantly more writes to hot non-volatile metadata in header frames than the other metadata. The increase in maximum bit updates for GM2 and GM3 in WH2 can be explained in the same way as in the case of

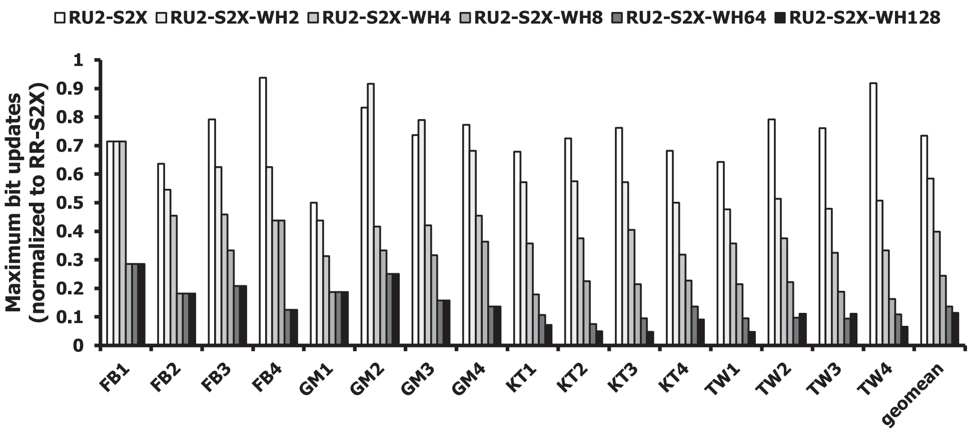

Fig. 15. Maximum bit updates vs. header frame reuse.

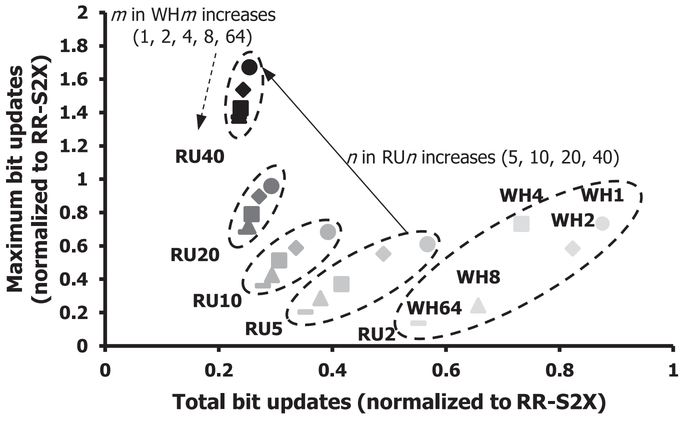

Fig. 16. Total vs. Maximum bit Updates: RUn-S2X-WHm Cases.

small reuse limit in Figure 14, i.e., due to frequent metadata migrations. In the cases of TW2 and TW3, WH128 gives larger maximum bit updates than WH64. This is because the overprovisioning, i.e., the wear-leveling-conscious data structure, is applied only to the journal header of header frame. Thus, as the header frame is reused more, the other metadata in the header frame, i.e., the valid bits (V) in the data frame descriptor of header frame, suffer from more bit updates, which changes the major contributor of maximum bit updates from the journal header to the valid bits when WH128 is applied.

Figures 13 and 15 exemplify the effectiveness of wear-leveling-conscious data structure in reducing maximum bit updates. When applying our proposed solution to software programs, in order to apply wear-leveling-conscious data structure and the notion of quasi-non-volatile data, there will be required new methods of profiling (for identifying hot write data), code analysis (to identify quasi-non-volatile data), code generation (to convert the data type of hot write data to the wear-leveling conscious structure) and data allocation (to allocate quasi-non-volatile data to volatile memory). We expect those methods can be automated, which is left for future work.

To summarize, Figure 16 shows the overall relationship between total bit updates and maximum bit updates in the cases of RUn-S2X-WHm. Each point in the figure represents the geometric mean over all the traces of the four applications. The figure

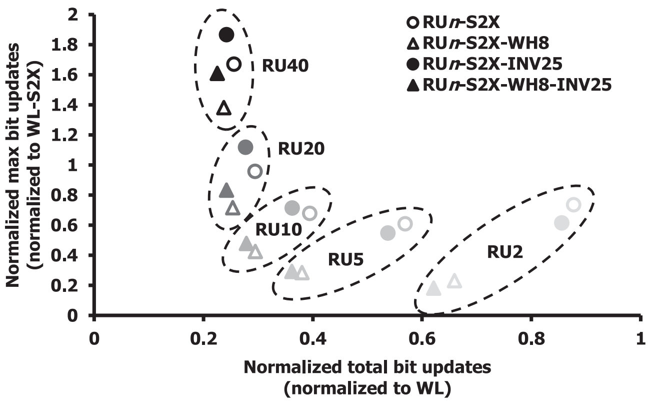

Fig. 17. Baseline vs. Incremental Garbage Collection.

shows two general trends. First, more reuse (larger n in RUn) results in fewer total bit updates and more maximum bit updates (towards upper left corner of the figure, see the solid arrow). However, when header frames are not fully reused (e.g., small m cases in RU2-S2X-WHm), frequent migrations increase the maximum bit update.

In Figure 16, the second trend is that more spare resources in the wear leveling-conscious data structures of header frames (larger m in WHm) results in more reductions in both the maximum and total bit updates (see the dashed arrow in the figure). This is because increasing m not only adds spare resources, but also elevates the reuse limit for header frames (note that, under RUn-S2X-WHm, each header frame is reused up to $n \times m$ times, as explained in Figure 11).

Based on the relationships shown in Figure 16, one can determine a suitable configuration under a given system requirement (e.g., maximum bit update limit as a function of PCM lifetime). For example, when we limit the maximum bit update to that of RR-S2X (=1), i.e., a FIFO management of free resources, our methods achieve a 75.2% reduction in total bit updates on average (by adopting RU20-S2X-WH64).

As mentioned in Section 5, the overprovision resource of wear-leveling-conscious data structure can incur area overhead. The area overhead of the best configuration, RU20-S2X-WH64 is calculated as follows. The size of journal header is 32 bytes and 63 spare blocks of 32 bytes each are utilized for the overprovisioning giving an area overhead of 2,016 bytes for a header frame. One header frame is required for one DB file and mobile applications used in our experiments typically utilize only a few DB files (up to 16 in Facebook application). Thus, the total area overhead is less than 32KB (~16*2,016 bytes), which occupies only 1.6% (=32KB/2MB) of journal resource. Considering that we used a minimum amount of journal resource, in reality, the area overhead can be much smaller.

The total and maximum bit updates can be improved by further optimizations. One possibility is to apply incremental garbage collection (GC). Figure 17 compares the relationship between the baseline and incremental GC. In the incremental GC, we manage the page frames in reuse state in groups and, upon a GC request, we free a group (RUn-S2X-WHm-INV25, which amounts to 25% of total page frames in reuse state in our experiments) of page frames. As the figure shows, the incremental GC can give two opportunities. First, in the cases that the reuse limit is small, i.e., RU2 cases, both total and maximum bit updates, are improved by adopting the incremental GC. There are two reasons, one for each improvement. First, the incremental GC delays invalidations, which gives more opportunities of resource reuse thereby reducing total

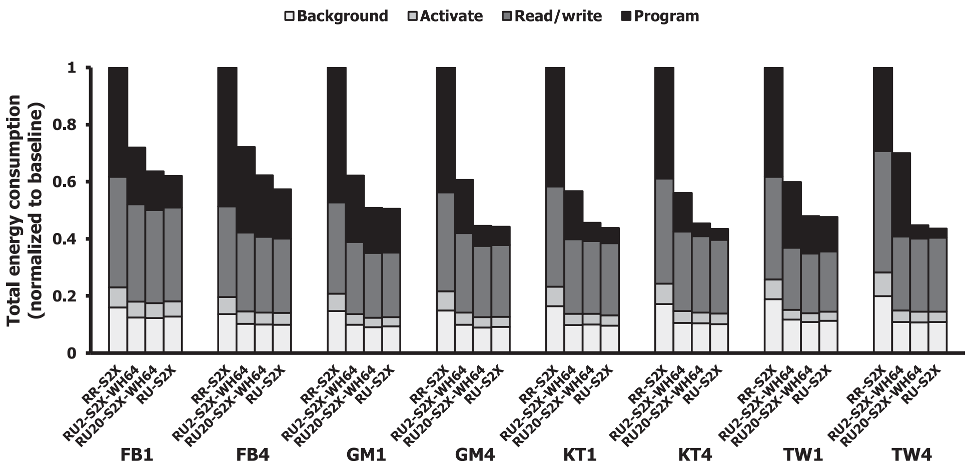

Fig. 18. Comparison of energy consumption.

bit updates. Second, in the original GC, the cases of small reuse limit suffer from the effect of random writes since frequent invalidations tend to write uncorrelated data to PCM cells. In such a case, the incremental GC tends to lessen the effect of random writes thereby reducing the maximum bit updates. However, such an effect is only visible when the reuse limit is small. In the case of large reuse limit, even under the incremental GC, the relationship shown in Figure 16 reappears. Thus, as the reuse limit increases, the total bit updates decrease while the maximum bit updates increase. Compared with the baseline GC where, upon a GC request, all the page frames in reuse state are freed, the incremental GC makes the effects stronger, i.e., more decrease in total bit updates and more increase in maximum bit updates, due to more opportunities of resource reuse. We think a joint optimization of reuse limit and incremental GC will give further improvements in bit updates, which is left for future work.

Figure 18 compares the energy consumption (normalized to RR-S2X) of the PCM-based SQLite journaling methods. The baseline, RR-S2X consumes the largest energy due to two reasons. First, it has much larger total bit updates than the others, as shown in Figure 9, which gives larger energy consumption in PCM program. Second, as will be shown later, it gives (average 2X) longer runtime of SQLite commands than the other methods (e.g., RU20-S2X-WH64), which gives large background energy consumption. On the contrary, RU20-S2X-WH64 reduces energy consumption by average 49.4% (for all the 16 traces including those not shown in this figure), compared to the baseline, through a careful consideration of differential write support for PCM devices.

Figure 19 compares the average processing time of SQLite command. The baseline, RR-S2X gives average 2.67 times longer processing time than the case of no journaling. It is mainly due to the frequent writes to the PCM. By reducing PCM writes, compared with the baseline, our proposed method, RU20-S2X-WH64 reduces by 52.3% average processing time. The reduction in average processing time results mostly from the reduction (by 75.2%) in total bit updates as shown in Figure 16.

7. RELATED WORK

Bit updates in the PCM can be reduced by applying data encoding. Cho and Lee [2009] propose a hardware solution called Flip-N-Write that applies invert coding. Flip-N-Write is implemented in the real PCM chip [Chung et al. 2011] used in our experiments. Sun et al. [2010] propose frequent value-based data encoding to reduce

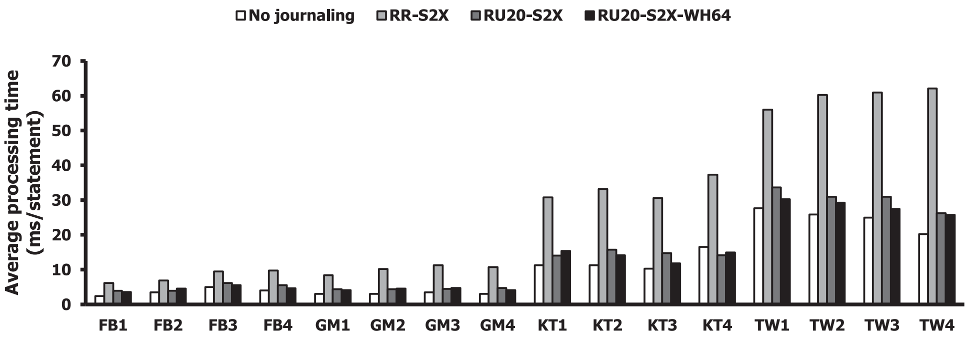

Fig. 19. Comparison of average processing time.

bit updates. Wear leveling is applied to reduce maximum bit updates. Qureshi et al. [2009b] propose start-gap wear leveling, which first applies address randomization to distribute spatially correlated hot data and periodically shifts their positions. Seong et al. [2010] present a wear-leveling method called *security refresh,* which implements address randomization by a single random key generation and XOR operation. Liu et al. [2013] propose Curling-PCM, which identifies hot data and periodically move them over PCM cells thereby distributing writes. Zhao et al. [2014] present a wear-leveling method called *intra-line flipping (ILF),* which periodically swaps bit positions for most and least significant parts of the data line in order to distribute bit updates mostly concentrated on the least significant part.

Kim et al. [2010] present storage class memory (SCM)-based journaling, where journal data are stored in the SCM (storage class memory), thereby resulting in fast recovery as well as better journaling performance. Fang et al. [2011] present an SCM-based database logging method, which covers SCM log space management, hole detection and recovery methods. Lee et al. [2013] propose a method of unioning buffer cache and journal called *UBJ* which reduces journaling writes to the fast NVM-based main memory. The UBJ and our case study of SQLite journal design are complementary to each other since the UBJ can reduce the frequency of writes for journaling and ours can be applied to each journaling write in the UBJ to further reduce total/maximum bit updates. Condit et al. [2009] present a file system for byte-addressable persistent memory, where a new technique referred to as *short-circuit shadow paging* provides fine-grained atomic updates by utilizing byte-addressability. Chi et al. [2014] propose a modified B+ tree, which reduces writes to the PCM by delaying and batching parent-node updates in node splits. This method is orthogonal to ours in that this method reduces PCM writes during node splits (incurred by writes to the database) in the B+ tree while ours can be applied to reduce bit updates in writes to the journal.

In order to reduce writes to non-volatile main memory, Hu et al. [2013] propose three software-based techniques, scheduling, migration and data recomputation to reduce dirty data eviction. Wang et al. [2012] propose write-activity-aware page table management scheme (WAPTM), where the bit updates of initializing a page table entry (PTE) are reduced by utilizing an initialization flag bit. In addition, bit updates incurred by PTE allocation are reduced by finding a page frame the physical address of which matches (i.e., has a minimum Hamming distance) with the contents of the new PTE. Compared with the existing methods, our proposed methods are effective for software programs in which small overwrites to dynamically allocated data are frequent.

Recently, there have been studies on hardware architectures for PCM-based main memory and storage. Qureshi et al. [2009a] propose a hybrid PCM/DRAM main memory and show that, given the same cost constraint, the hybrid memory outperforms the DRAM-only main memory by reducing disk traffic. Zhou et al. [2009] propose dynamic data migration between DRAM and PCM in a tiered hybrid memory where the migration from PCM to DRAM is triggered based on hot write data detection. Lee et al. [2009] show that a PCM-only main memory can provide comparable performance to the DRAM-only main memory by decomposing a row buffer into several smaller ones. Kim et al. [2008] present a PCM-based metadata filter in SSD where frequent metadata writes are absorbed by the PCM buffer, thereby reducing write pressure on the Flash memory. Sun et al. [2010] propose to utilize PCM as a write buffer in SSD in order to absorb hot data writes and reduce writes to Flash memory.

Recently, there have been real PCM storage system prototypes. For example, Moneta is a PCIe-attached storage array with PCM emulation capability [Caulfield et al. 2010]. Onyx is a real PCM-based storage prototype that provides better performance for reads and small writes than a solid state disk (SSD) [Akel et al. 2011]. Vucinic et al. [2014] presented DC Express for low read latency in PCIe-based PCM storage. Kim et al. [2014] presented an evaluation study using a PCM-based SSD and show the utility of PCM in both caching and tiering solutions.

Compared with the above-mentioned existing works, our key contributions are (1) cross-layer software optimization that is conscious of hardware capability of bit-level differential write, (2) wear-leveling-conscious data structures that minimize the impact of frequently reused resource on the wear-out behavior, and (3) a data management scheme that classifies quasi-non-volatile and non-functional data and stores them in volatile memory in order to minimize the total amount of non-volatile data.

8. CONCLUSIONS

In order to exploit bit-level differential write for total bit update reductions in real PCM chips, we propose a software design methodology that reuses previously allocated PCM resources in the case of frequent and small-sized overwrites. This in turn reduces data differences at the bit level and thereby reduces PCM writes. Since too much reuse might worsen wear-out problems, we design wear-leveling-conscious data structures that utilize XOR flags and overprovision spare resources to address the localized wear-out incurred by reuse. In addition, we reduce the total amount of PCM traffic by identifying quasi-non-volatile and non-functional data from existing non-volatile data and allocating such data in volatile memory.

We performed a case study of software design for PCM using SQLite journaling on our PCM-based prototype board. Experiments with four representative mobile applications showed that our proposed methods achieved a 52.3% reduction in processing time and a 49.4% reduction in energy consumption compared to a typical FIFO management of free resources. Compared with a typical FIFO management of free resources on the PCM, on top of the existing hardware approach, Flip-N-Write, our reuse method gave average 75.2% reduction in total bit updates without increasing maximum bit updates. In our future work, we aim to show the benefits of differential write-conscious software design with general software design frameworks, e.g., profilers and compilers.

REFERENCES

A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. 2011. Onyx: A protoype phase change memory storage array. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'11)*. 2–2.

Apple. 2015. Mac OS X: About file system journaling. retrieved Aug. 16, 2015 from https://support.apple.com/en-us/HT204435.

A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. 2010. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. 385–395.

P. Chi, W. Lee, and Y. Xie. 2014. Making B^+-tree efficient in PCM-based main memory. In *Proceedings of the 2014 International Symposium on Low Power Electronics and Design (ISLPED'14)*. 69–74.

S. Cho and H. Lee. 2009. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *Proceedings of the 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. 347–357.

K. Chodorow. 2012. How mongodb's journaling works. Retrieved Aug. 16, 2015 from http://www.kchodorow.com/blog/2012/10/04/how-mongodbs-journaling-works/.

H. Chung, B. H. Jeong, B. Min, Y. Choi, B. Cho, J. Shin, J. Kim, J. Sunwoo, J. Park, and Q. Wang. 2011. A 58nm 1.8 V 1GB PRAM with 6.4 MB/s program bw. In *Proceedings of the 2011 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC'11)*. 500–502.

J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. 133–146.

DataStax. 2015. Understanding the architecture. Retrieved Aug. 16, 2015 from http://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureIntro_c.html.

P. Desnoyers. 2012. Analytic modeling of SSD write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR'12)*. 12.

R. Fang, H. Hsiao, B. He, C. Mohan, and Y. Wang. 2011. High performance database logging using storage class memory. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE'11)*. 1221–1231.

D. R. Hipp and D. Kennedy. 2010. SQLite (Jan. 2010). Retrieved Sep. 20, 2014 from http://sqlite.org/.

J. Hu, C. J. Xue, Q. Zhuge, W. Tseng, and E. H. Sha. 2013. Write activity reduction on non-volatile main memories for embedded chip multiprocessors. *ACM Transactions on Embedded Computing Systems (TECS)*, 12, 3, 77.

JEDEC Standard. 2011. Low power double data rate 2 (LPDDR2). *JESD209-2E* (April 2011).

H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. 2014. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. 33–45.

J. Kim, H. Lee, S. Choi, and K. Bahng. 2008. A PRAM and NAND flash hybrid architecture for high-performance embedded storage subsystems. In *Proceedings of the 8th ACM International Conference on Embedded Software (EMSOFT'08)*. 31–40.

Y. Kim, I. H. Doh, E. Kim, J. Choi, D. Lee, and S. H. Noh. 2010. Design and implementation of transactional write buffer cache with storage class memory. *Journal of Korean Institute of Information Scientists and Engineers: Computing Practices and Letters,* 16, 2 (Feb. 2010), 247–251.

S. Kwon, D. Kim, Y. Kim, S. Yoo, and S. Lee. 2012. A case study on the application of real phase-change RAM to main memory subsystem. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'12)*. 264–267.

B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. 2009. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. 2–13.

E. Lee, H. Bahn, and S. H. Noh. 2013. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. 73–80.

T. Lee, D. Kim, H. Park, S. Yoo, and S. Lee. 2014. FPGA-based prototyping systems for emerging memory technologies. In *Proceedings of the 25th IEEE International Symposium on Rapid System Prototyping (RSP'14)*. 115–120.

C. Lehene. 2010. Why we're using HBase: Part 2. Retrieved Aug. 16, 2015 from http://hstack.org/why-were-using-hbase-part-2/.

The Linux Kernel Organization. 2015. Ext4 Filesystem. Retrieved Aug. 16, 2015 from https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

D. Liu, T. Wang, Y. Wang, Z. Shao, Q. Zhuge, and E. Sha. 2013. Curling-PCM: Application-specific wear leveling for phase change memory-based embedded systems. In *Proceedings of 18th Asia South and Pacific Design Automation Conference (ASP-DAC)*. 279–284.

MariaDB Corporation. 2015. MariaDB Documentation. Retrieved Aug. 16, 2015 from https://mariadb.com/kb/en/mariadb/undo-log/.

Microsoft TechNet. 2015. Recovering Data with NTFS. Retrieved Aug. 16, 2015 from https://technet.microsoft.com/en-us/library/cc976815.aspx.

V. Mohan, T. Bunker, L. Grupp, S. Gurumurthi, M. Stan, and S. Swanson. 2013. Modeling power consumption of NAND Flash memories using FlashPower. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 32, 7, 1031–1044.

M. A. Olson, K. Bostic, and M. Seltzer. 1999. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 1999.

PostgreSQL Global Development Group. 2015. PostgreSQL: Documentation. Retrieved Aug. 16, 2015 from http://www.postgresql.org/docs/9.4/static/wal-intro.html.

M. K. Qureshi, V. Srinivasan, and J. A. Rivers. 2009a. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. 24–33.

M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. 2009b. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. 14–23.

N. H. Seong, D. H. Woo, and H. H. S. Lee. 2010. Security refresh: Prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. 383–394.

G. Sun, Y. Joo, Y. Chen, Y. Chen, and Y. Xie. 2010. A Hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of the 2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*. 1–12.

H. Tuuri and C. Sun. 2009. InnoDB Internals: InnoDB File Formats and Source Code Structure. In *MySQL Conference*, April 2009.

D. Vucinic, Q. Wang, C. Guyot, R. Mateescu, F. Blagojevi, L. Franca-Neto, D. Le Moal, T. Bunker, J. Xu, S. Swanson, and Z. Bandic. 2014. DC express: shortest latency protocol for reading phase change memory over PCI express. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. 309–315.

T. Wang, D. Liu, Z. Shao, and C. Yang. 2012. In *Proceedings of 17th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 317–322.

B. D. Yang, J. E. Lee, J. S. Kim, J. Cho, S. Y. Lee, and B. G. Yu. 2007. A low power phase-change Random Access Memory using a data-comparison write scheme. In *Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS)*. 3014–3017.

J. Yoon, S. Lee, and S. Yoo. 2012. Bloom filter-based dynamic wear leveling for phase change RAM. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'12)*. 1513–1518.

M. Zhao, L. Shi, C. Yang, and C. J. Xue. 2014. Leveling to the last mile: Near-zero-cost bit level wear leveling for PCM-based main memory. In *Proceedings of 32nd IEEE International Conference on Computer Design (ICCD)*. 16–21.

P. Zhou, B. Zhao, J. Yang, and Y. Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. 14–23.

Received May 2015; revised August 2015; accepted November 2015