PCMLogging: Optimizing Transaction Logging and Recovery Performance with PCM

Shen Gao, Jianliang Xu, Theo Härder, Bingsheng He, Byron Choi, Haibo Hu

Abstract—Phase-change memory (PCM), as one of the most promising next-generation memory technologies, offers various attractive properties such as non-volatility, byte addressability, bit alterability, and low idle energy consumption. Recently, PCM has drawn much attention from the database community for optimizing query and transaction performance. As a complement to existing work, we present PCMLogging, a novel logging scheme that exploits PCM for both data caching and transaction logging to minimize I/O accesses in disk-based databases. Specifically, PCMLogging caches dirty pages/records in PCM and further maintains an implicit log in the cached updates to support database recovery. By integrating log and cached updates, PCMLogging enables simplified recovery and prolongs PCM lifetime. Furthermore, using PCMLogging, we develop a wear-leveling algorithm, that evenly distributes the write traffic across the PCM storage space, and a cost-based destaging algorithm that adaptively migrates cached data from PCM to external storage. Compared to classical write-ahead logging (WAL), our trace-driven simulation results reveal up to 1∼20X improvement in system throughput.

Index Terms—Phase-change memory, database recovery, caching, performance

1 INTRODUCTION

Write-ahead logging (WAL) has been extensively adopted by database systems as state-of-the-art mechanism to ensure transaction atomicity and durability [8], [21], [28]. Using the WAL scheme, transactions buffer dirty data and redo/undo log records in a volatile dynamic random access memory (DRAM). Upon transaction commit/abort or dirty-page replacement, WAL spends I/Os in propagating the buffered log data to an external stable storage such as hard-disk drives (HDD). Due to the large speed gap between DRAM and external storage, it is cost-effective in most cases to flush log records instead of dirty data, which would comprise all pages a transaction has updated. Yet, this approach has a few deficiencies. First, log I/Os cause additional cost besides asynchronous dirty-page write I/Os. Second, the log I/O latency aggravates lock contention and increases context switching overhead [14]. Third, the required recovery mechanism such as ARIES [21] is fairly complex. Although the use of checkpoints will help accelerate the recovery process, it makes its implementation more complicated. Having said that, using slow HDDs as external storage, maintaining a separate log has been proven a good trade-off that achieves high system throughput without compromising database consistency [27].

Recently, the emergence of non-volatile random access memory (NVRAM) fills the speed gap by taking the best of DRAM and HDD: read/write speed close to DRAM and non-volatility similar to HDD. Within the variety of such memories, phase-change memory (PCM) is one of the most promising ones [15]. Compared with DRAM, PCM is not only a persistent random-access memory but also provides higher chip density. Compared to HDDs and flash-memory-based solid-state drives (SSDs), PCM's access speed exceeds that of them by two to four orders of magnitude. Moreover, it is a byte-addressable memory that allows direct access of data (similar to DRAM), instead of having to go through a block-based I/O interface. Unlike SSD having an erase-before-write constraint, PCM is bit alterable without requiring a time-consuming erasure operation. These advantages already brought manufacturers to start the mass production of PCM at a reasonable price. For example, Numonyx (now Micron) released their first PCM products in 2010 [34]. Samsung has recently announced a 8Gb PCM package and the deployment of PCM in their mobile handsets [5], [37]. Hence, PCM may be envisioned in the near future to be integrated into the memory/storage hierarchy of computer systems [6], [15], [25].

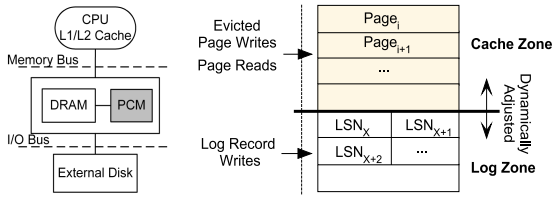


Fig. 1. Memory/storage hierarchy and the WAL design

Research work already started to explore the opportunities of optimizing database performance by using PCM [4], [8], [20]. As a complement to existing work mainly focusing on query processing and storage management, we explore in this paper how to leverage PCM for improving transaction performance through novel logging and recovery methods. We advocate the hybrid memory/storage hierarchy suggested by [6], [25], where both DRAM and PCM are placed on top of

- *Shen Gao, Jianliang Xu, Byron Choi and Haibo Hu are with Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Hong Kong.*
- *Theo Härder is with Department of Computer Science, University of Kaiserslautern, Germany.*
- *Bingsheng He is with School of Computer Engineering, Nanyang Technological University, Singapore.*

the I/O interface (see Figure 1(a)). The non-volatile nature of PCM makes it an ideal place for saving transaction log records. Moreover, its fast access speed enables caching of the dirty pages evicted from DRAM to minimize disk I/Os. Hence, a unique opportunity is emerging to support both data caching and transaction logging at the same time in PCM.

A basic method of adopting traditional WAL is to divide the PCM space into two zones: 1) a cache zone to cache dirty pages and 2) a log zone to keep transaction log records (see Figure 1(b)).[1] When a page is evicted from DRAM, it will be moved to the PCM cache zone. In case the PCM cache zone is full, a replacement algorithm such as LRU has to make room first. When a transaction commits, the log records buffered in DRAM will be flushed to the PCM log zone. When the log zone is full, we may trigger a checkpoint process or migrate some of the log records to an external disk to reclaim log space. Clearly, this WAL design not only reduces the data and log I/Os to external disks, but also accelerates commit processing of a transaction. Nevertheless, this method has several critical drawbacks:

- **Data Redundancy**. The information maintained in the PCM cache zone and log zone might be redundant. For example, a transaction update kept in a log record might also be cached in a dirty page. Such data redundancy not only wastes the precious storage space but also incurs more writes to PCM, which shortens its lifetime.
- **Complicated Space Management.** Space management of PCM becomes a challenge as it is shared by the cache zone and the log zone. Although a dynamic scheme may dynamically adjust these two zones, an ill-advised setting may significantly deteriorate the overall performance.
- **Excessive Write Traffic.** Most updates of an OLTP workload are small writes [3]. Thus, caching full pages in PCM may not be very cost effective, because caching of the page's clean portion is not needed to achieve durability. Even worse, writing full pages involves a large amount of write traffic, thus reducing PCM's lifetime.
- **Expensive Recovery.** Crash recovery cost is still expensive. Although, guided by the ARIES algorithm, many dirty pages have been captured in the persistent PCM cache, we still need to go through the analysis, redo, and undo phases to recover the database to a consistent state.

Consequently, this method may not fully exploit the performance advantages provided by PCM. To address its drawbacks, we propose a new record-level logging scheme named PCMLogging. Different from the above method where log zone and cache zone are independent, we integrate cached updates and log records into implicit log records that are kept in PCM. By taking advantage of the implicit log, our scheme saves log I/Os and simplifies space management of PCM. Moreover, our new scheme makes checkpoints unnecessary and enables a simpler and cheaper recovery algorithm (to be detailed in Section 3). Our main contributions are as follows:

- To the best of our knowledge, PCMLogging is the first alternative to WAL that exploits the PCM hardware features to optimize transaction logging and recovery.
- To address the PCM endurance issue, we propose a wear-leveling algorithm for PCMLogging that discovers and redistributes skewed write traffic by considering the residence time of PCM data. Our algorithm does not incur any space overhead and has a low time complexity.
- As a staging area between DRAM and external disk, PCM needs to migrate cached data to disks when space is in short supply (known as *destaging*). Because PCM overflow would block transaction execution and increase lock contention, we develop a cost model based on the Markovian birth-death process and adaptively adjust the speed of data migration so as to balance the costs of data migration and PCM overflow.
- We develop a trace-driven simulator to evaluate the performance of PCMLogging based on the TPC-C benchmark. Compared to WAL, the simulation results reveal that PCMLogging achieves substantial performance improvements of up to 1X, 3X, and 20X for transaction throughput, when employing HDD, SSD, and SD card-based flash memory as external storage, respectively. PCM lifetime and transaction response time are also significantly improved by our proposed wear-leveling and destaging algorithms.

Organization. Section 2 prepares the background of our research, including PCM hardware features and memory architecture alternatives. Section 3 presents the PCMLogging scheme and discusses its various operations in detail. In Section 4, a wear-leveling algorithm enhancing PCM's endurance is described. A cost-based adaptive destaging technique is presented in Section 5. In Section 6, we extensively evaluate the PCMLogging performance. Section 7 surveys related work and, finally, Section 8 concludes this paper and discusses future directions.

2 Background

In this section, we give some background information concerning NVRAM and PCM technologies and review the alternatives of integrating PCM into the memory architecture.

2.1 NVRAM and PCM

NVRAM has long been considered as dream-class storage, providing superb fast access speed like DRAM and non-volatility like HDD. Over decades, substantial efforts aimed at the development of practical solutions for NVRAM. Among many other alternatives such as ferroelectric RAM (FeRAM) and magnetic RAM (MRAM), PCM appears as today's most promising NVRAM technology due to a recent breakthrough in materials technology [5], [15], [34].

Table 1 summarizes the hardware performance of several current storage technologies including DRAM, flash memory, PCM, and HDD, where density, read/write latency, and endurance are compared [4]. Read latency of PCM is close to that of DRAM and two orders of magnitude shorter than that of flash memory. Write latency of PCM is in between those of DRAM and flash memory. Without erase-before-write

1. Yet, a similar idea was proposed in [7] by using battery-backed-up DRAM.

TABLE 1
Comparison of Storage Technologies [4]

Parameter	DRAM	Flash	HDD	PCM
Density	1X	4X	N/A	2-4X
Read latency (granularity)	20-50ns (64B)	$\sim$25μs (4KB)	$\sim$5ms (512B)	$\sim$50ns (64B)
Write latency (granularity)	20-50ns (64B)	$\sim$50μs (4KB)	$\sim$5ms (512B)	$\sim$1μs (64B)
Endurance (write cycles)	N/A	10^4-10^5	∞	10^6-10^8

constraint, its random-write performance is much better than that of flash memory. Moreover, write latency of PCM is three orders of magnitude shorter than that of HDD.

In addition, PCM has the following important hardware features [6], [15], [34]:

- *Fine-grained access:* Compared to other non-volatile memory technologies such as flash memory, erase-before-write and page-based access do not restrain PCM. It is byte addressable (or word addressable) and bit alterable, which enable PCM to support small in-place updates.
- *Asymmetric read/write latency:* As shown in Table 1, the write speed of PCM is about 20 times slower than its read speed. This is similar to flash memory that has such an asymmetry as well.
- *Endurance limitation:* Similar to flash memory, PCM endures a limited number of writes, about 10^6 to 10^8 writes for each cell, which is however much higher than that of flash memory.
- *Low idle energy consumption:* While PCM uses for data access similar energy as DRAM (i.e., 1-6 J/GB), it consumes much lower idle energy compared to DRAM (i.e., 1 v.s. 100 mW/GB).

This paper focuses on improving transaction logging and recovery performance by PCM integration. For this purpose, we mainly exploit PCM's low access latency and fine-grained access granularity and address its endurance limitation.

2.2 PCM in the Memory Hierarchy

So far, two representative architectures for the use of PCM in a memory hierarchy are proposed [6], [25]: 1) PCM co-existing with DRAM to serve as main memory (as shown in Figure 1(a)); 2) main memory only composed of PCM chips thereby fully replacing DRAM. Considering the hardware features of PCM, the co-existence architecture might be more practical. The first reason is that PCM has endurance limitation, which prevents a complete replacement of DRAM. Secondly, write latency of PCM is still 20-50 times larger than that of DRAM. Thirdly, PCM capacity is expected to still remain relatively small in the near future, in comparison with DRAM. Thus in this study, we focus on the memory architecture using PCM as an auxiliary memory, being a staging area between DRAM and external disks.

3 PCMLogging

We consider the memory architecture as shown in Figure 1(a). Without largely modifying the buffer manager residing in DRAM memory, we present a new logging scheme, called PCMLogging, where the cached updates and transaction log records are combined and kept in PCM. The wear-leveling and data destaging issues of PCMLogging will be discussed in Sections 4 and 5.

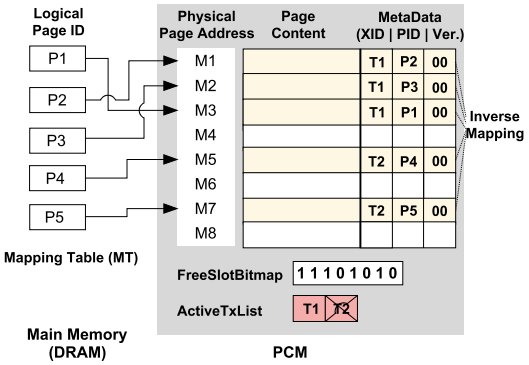


Fig. 2. Page format and Mapping Table

3.1 Overview

The basic idea of PCMLogging is to integrate the transaction log into the updates cached in PCM, by exploiting the persistence property of PCM storage. For ease of exposition, we assume in this section that PCM caching granularity is a page and concurrency control is also at a page level. That is, a page can be updated by at most one transaction at a time. To further improve transaction processing performance, we will extend the design to record-level caching and record-based concurrency control in Section 3.3.

Overview of the PCMLogging scheme: To support data caching in PCM, we maintain the following data structures in main memory (DRAM) / PCM (see Figure 2):

- *Mapping Table.* This table maps logical page IDs to physical PCM addresses. It is maintained in DRAM rather than in PCM, because the mapping entries are frequently updated and the write speed of PCM is 20-50 times slower than that of DRAM.
- *Inverse Mapping.* The inverse mapping is embedded in each PCM page as metadata (i.e., PID). It is used to construct the initial Mapping Table at boot time.
- *FreeSlotBitmap.* This bitmap is used to keep track of the free page slots in PCM. Note for PCMLogging, only the dirty pages evicted from main memory are cached in PCM to minimize disk write I/Os.

Inspired by shadow paging [10], we adopt an out-of-place update scheme in PCM. When a transaction is to commit, all its dirty pages are flushed to PCM to ensure durability. Also, when a dirty page is evicted from main memory, it will be cached in PCM. For each dirty page, if there already exists a previously committed version in PCM, the committed version will not be overwritten. Instead, the previous version is retained, while the dirty page as new version is written to a free PCM slot. After that, the logical page address in the Mapping Table is adjusted to the new version. The need of retaining the previously committed version is to support undo operations in case of transaction rollback or system crash.

To support transaction recovery, an ActiveTxList is maintained in PCM to record the in-progress transactions that have dirty pages cached in PCM. Each cached page records the XID of the last transaction that caused the page to be dirty. Before the first dirty page of a transaction is written to PCM, its corresponding XID should be recorded in the ActiveTxList to guarantee atomicity. The XID is not removed until the transaction is to commit and all its dirty pages are flushed to PCM. Thus, during recovery, if the XID of a transaction is found in the ActiveTxList, it implies that the transaction was not yet committed before the crash; otherwise, the transaction was already committed. Consequently, each PCM page can be recovered according to the status of the corresponding transaction. For example, if PCM appears as shown in the right part of Figure 2, we can infer that $T1$ is not yet committed, whereas $T2$ is committed. Thus, the pages updated by $T1$ (i.e., those stored in $M1$-$M3$) are discarded,[2] whereas the pages updated by $T2$ (i.e., those stored in $M5$ and $M7$) need to be restored. Accordingly, the FreeSlotBitmap will be updated to "**000**01010." We note that, to avoid hot-spots in PCM, wear-leveling techniques should be adopted to evenly distribute writes across the PCM space, which will be discussed in more detail in Section 4.

As a brief summary, PCMLogging eliminates the explicit transaction log by integrating it into the dirty pages cached in PCM. This integrated design has several advantages. First, the data redundancy between the log and cached updates is minimized. Second, it avoids the challenging space management issue, which is a must if they are separated. Third, recovery can be done without checkpoints, because we do not maintain an explicit log. In addition, the recovery process becomes extraordinarily simple and efficient. In the following, we describe the PCMLogging scheme in detail.

3.2 PCMLogging Operations

Durability is achieved by forcing the affected dirty pages to PCM when a transaction is to commit. On the other hand, a *steal* buffer policy allows a dirty page to be flushed to PCM before the transaction commits. To ensure atomicity, undo operations will be needed if the transaction is finally aborted. To efficiently support such undo operations, we maintain two additional data structures in main memory:

- *Transaction Table (TT).* This table records all in-progress transactions. For each of them, it keeps track of all its dirty pages stored in main memory and PCM. The purpose is to quickly identify relevant pages when the transaction is to commit or abort.
- *Dirty Page Table (DPT).* This table keeps track of the previously committed version of each PCM page "overwritten" by an in-progress transaction. Recall that we employ out-of-place updates in PCM. This is necessary for restoring the previously committed version in the event of a rollback. A dirty page entry will be removed from the table, once the in-progress transaction is committed or aborted.

2. They have not left the PCM, because our destaging algorithm (Section 5) only flushes committed pages to external storage.

PCMLogging needs to handle the following key events:

Flushing Dirty Pages to PCM. When main memory becomes full or a transaction is to commit, some dirty pages may need to be flushed to PCM. For each dirty page, we first check the Transaction Table. If it is the first dirty page of the transaction to be flushed to PCM, we add the related XID to the ActiveTxList in PCM before flushing. If there exists a previously committed version M in PCM, we do not overwrite it in place. To support undo, we create instead an out-of-place copy M' with a larger version number. Then, M is added to the Dirty Page Table and the page is mapped to M' in the Mapping Table. Finally, the Transaction Table is updated.

Commit. Upon receiving a commit request, all dirty pages of the transaction being still buffered in main memory are forced to PCM, by consulting the Transaction Table. After that, we remove its XID from the ActiveTxList to indicate the transaction is committed. Next, if any of its pages is contained in the Dirty Page Table, the previous versions are discarded by resetting their corresponding bits in the FreeSlotBitmap. Finally, we clear the relevant entries in the Transaction Table and Dirty Page Table.

Abort. When a transaction is aborted, all its dirty pages are discarded from PCM, by consulting the Transaction Table. If any of its pages is contained in the Dirty Page Table, the current version should be invalidated and the mapping should be re-mapped (restored) to the previous version in the Mapping Table. Finally, we clear its XID in the ActiveTxList and the relevant entries in the Transaction Table and Dirty Page Table.

An Example: *Consider the example shown in Figure 3, where T1 is in progress and T2 is committed. Suppose now a new transaction T3 updates page P5. Before this dirty page is flushed, T3 points to page P5 kept in main memory (see Figure 3(a)). When it is flushed to PCM slot M8, T3 is added to the ActiveTxList in PCM (see Figure 3(b)). After that, P5 is mapped to M8, T3 points to M8, and the previous version M7 is kept in the Dirty Page Table. Finally, if T3 is to commit, it is removed from the ActiveTxList; the previous version is discarded (the corresponding bit becomes 0 in the FreeSlotBitmap); and the corresponding entries are removed from the Transaction Table and Dirty Page Table (see Figure 3(c)). Otherwise, if T3 is finally aborted, the current version is discarded (the corresponding bit becomes 0 in the FreeSlotBitmap) and the previous version is restored in the Mapping Table; and the corresponding entries are also removed from the ActiveTxList, Transaction Table, and Dirty Page Table (see Figure 3(d)).*

Recovery. A recovery process is invoked when the system restarts after a failure. It identifies the last committed version for each PCM page and re-constructs the Mapping Table. To do so, the recovery algorithm reads all *valid* pages, whose corresponding bits are 1's in the FreeSlotBitmap. As a valid page can be the latest version of the page updated by an in-progress transaction or a previously committed version that needs to be restored, we discard the uncommitted pages that belong to an in-progress transaction, which can be identified from accessing the ActiveTxList. Note that this process does not involve any disk I/Os.

Discussion. A subtle issue is hidden in the example of Figure 3(c). What happens in case of crash, after $T3$ is re-

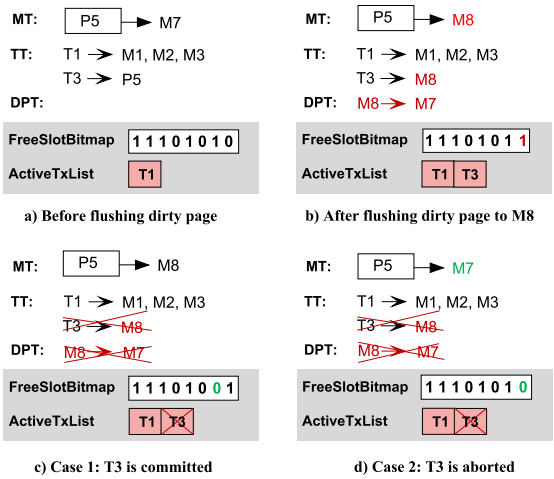


Fig. 3. An example of PCMLogging (MT: Mapping Table; TT: Transaction Table; DPT: Dirty Page Table)

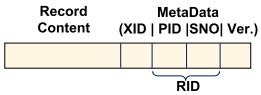


Fig. 4. Record format for PCMLogging

moved from the ActiveTxList and before the previous version $M7$ is discarded, i.e., both $M7$ and $M8$ are valid at system restart. Because both of their corresponding transactions $T2$ and $T3$ are committed, we are not able to determine the latest version. Therefore, a version number is kept for each page cached in PCM. For example, a 2-bit version number would suffice under lock-based concurrency control, because at most two versions co-exist (i.e., current version and previously committed version). Given two consecutive numbers in the modulation domain, the version with the number most recently assigned would be considered the latest.

3.3 Record-Level PCMLogging

After illustrating our basic idea, we now extend PCMLogging to record-level logging for practical use. Recall that PCM supports byte addressability and bit alterability. Thus, in response to small writes in the OLTP workload and record-based concurrency control, we propose to cache dirty records, instead of dirty pages, in PCM. The advantages are three-fold: 1) because the dirty information is only a small portion of a page in the OLTP workload, caching only dirty records saves PCM space; 2) caching dirty records reduces the write traffic to PCM, which decreases the chance of potential contention and prolongs PCM lifetime; 3) this naturally supports record-based concurrency control, which is widely adopted by modern database systems for a high degree of concurrency.

To support record-based caching and logging, we make the following modifications to the page-level PCMLogging scheme presented in the last section:

- *Record-level data management:* The cache slots in PCM should now be managed in units of records, rather than pages. The PCM record format is similar to the PCM page format shown in Figure 2 except that now the payload is a record and we have an additional metadata field, i.e., slot no. (SNO) (see Figure 4).[3] PID and SNO constitute a record identifier (RID). To manage the free PCM space, a similar slot-based bitmap can be adopted if the records have fixed size. If they are variable, we may employ a standard method such as slotted directory [27].
- *Table structures:* In the Mapping Table, we still organize the entries by pages, but now each entry (related to a dirty page) keeps track of the mappings of all dirty records in the related page (the records may be indexed by a binary tree based on their RIDs). Similarly, in the Dirty Page Table, the entries are also organized by pages, and each entry keeps track of the previous versions of all PCM records in the related page. We also maintain, for each page, a list of dirty records stored in main memory. When a transaction is to commit or a dirty page is evicted from main memory, the related dirty records are identified through this table and flushed to PCM. In the Transaction Table, each transaction maintains a list of its dirty records.
- *Serving read/write requests:* By consulting the Mapping Table, a record can be directly accessed from PCM, if available. Otherwise, if an entire page is requested, the page is loaded from external storage and merged with the latest contents of the relevant records cached in PCM. Note, loading page contents from external disk and PCM is a parallel process and access latency of PCM is negligible.
- *Destaging:* When the PCM is close to full (or the disk system is idle), we may select some committed records and write them back to the external disk. In this case, we first load the corresponding page from the external disk, and then merge it with the committed record(s) before writing back. This destaging process has some impact on system performance and a cost-based adaptive destaging algorithm will be presented in Section 5.

Algorithms 1, 2, and 3 summarize the detailed operations of the record-level PCMLogging scheme.

Next, we discuss the conceivable performance impact of the record-level scheme in two aspects: data structure overhead and cost of destaging. First, as we shrink the access granularity from page to record, the size of auxiliary data structures in DRAM, such as Mapping Table and Dirty Page Table, grows. This will burden the buffering capacity of main memory.

Algorithm 1 Record Flushing Logic in `PCMLogging`

Procedure: Flushing (Record t)
Let T be the transaction that made the last update to t
if t is T's first dirty record to be flushed **then**
Append T to the *ActiveTxList* in PCM
Write t to a free space of PCM
if there is a previously committed version of t in PCM **then**
Add the previous version to the *Dirty Page Table*
else if there is a copy of t (updated by T) in PCM **then**
Invalidate the uncommitted copy
Update the *Mapping Table* and *Transaction Table*

3. In case of deletion, the record content in PCM can be void.

Algorithm 2 Commit/Abort Logic in `PCMLogging`

Procedure: Commit (Transaction T)
  Access the *Transaction Table*
  for each dirty record t of T in DRAM **do**
    Flushing(t)
  Remove T from the *ActiveTxList* in PCM
  Discard the previous versions of T's dirty records
  Update the *Transaction Table* and *Dirty Page Table*

Procedure: Abort (Transaction T)
  Discard all T's dirty records, if any, in both PCM and DRAM
  if $T \in$ *ActiveTxList* **then**
    Restore the previous versions of T's dirty records in PCM and update the *Mapping Table*
    Remove T from the *ActiveTxList*
  Update the *Transaction Table* and *Dirty Page Table*

Algorithm 3 Recovery Logic in `PCMLogging`

Procedure: Recovery
  Scan all valid records in PCM
  for each valid record t **do**
    if t's XID $\in$ *ActiveTxList* or t has a larger version **then**
      Discard t
    else
      Add an entry of t to the Mapping Table in DRAM

Nevertheless, as shown by the experiments (Section 6), this cost fraction is small and can be compensated by the increased read and write hits to cached records in PCM. Second, regarding the destaging cost, because we need to read a page from disk and merge it with the committed record(s) in PCM before writing back, this cost will run up by an additional read I/O. But much more recovery-specific data can be stored in PCM, because we cache dirty records only. Thus, a page has probably collected more dirty records when it is destaged to disk, thereby reducing the overall I/O cost. Moreover, because less data is written to PCM during transaction commit, the commit processing time can be reduced.

4 Wear Leveling

Wear leveling is a critical mechanism for improving PCM lifetime. In this section, we propose a probabilistic record-swapping algorithm to evenly distribute the write traffic across the PCM space and therefore enhance its lifetime.

4.1 Design Requirements of Wear Leveling in PCM-Logging

PCM has a limited write endurance with about 10^6-10^8 writes per cell. Some previous work already focused on wear leveling at the device level [15], [17]. For example, [17] implements a read-before-write loop at the bit level to improve reliability and extend lifetime. Assume that the PCM space is divided into slots, and one record occupies one or more units. While these techniques are helpful to even out the write distribution within pre-defined units (i.e., intra-record space), the writes among different units could still be very skewed, because of the skewed writes in transactional workloads [3]. Due to this fact, a system-level approach is necessary to discover skewed write patterns and evenly distribute the writes to different units.

Using PCMLogging, two kinds of data have an impact on the write traffic of PCM. First, some objects such as ActiveTxList and FreeSlotBitmap are high-traffic data structures and frequently updated. Inspired by [18], we can periodically re-locate them to prevent them from becoming hot spots. Second, recall that for dirty records cached in PCM, we apply out-of-place updates. Upon each write request, we allocate free space for the write. Thus in the next section, we propose a new wear-leveling algorithm that works with such a space allocation mechanism. The proposed algorithm is lightweight in the sense that it does not incur additional space overhead.

4.2 Probabilistic Record-Swapping Algorithm

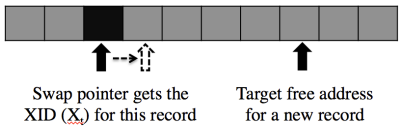


Fig. 5. Wear-leveling in PCMLogging

For the PCM space allocated to cached records, the overall objective is to avoid hot and cold spots and evenly distribute the incoming update traffic to physical addresses. Because we employ out-of-place writes to ensure durability, updates to a (hot) record may be distributed to different physical addresses. However, the write frequency of the (physical) slots held by cold records is relatively little, thereby increasing the skewness of write distributions across the whole PCM space. To address this problem, we propose to re-locate cold records during space allocation for new writes according to their residence time.

To obtain the residence time of a record, the most accurate way is to maintain a timestamp. However, such metadata incurs additional space overhead. Thus, to minimize the overhead, we decided to use as an indicator the transaction ID (XID), which is already maintained for the PCMLogging scheme (Section 3). As XID is a monotonically increasing number, the difference between a record's XID and the current XID of the system can very often indicate the time since the record's last update. Compared to the timestamp-based approach, it can also avoid a miscounting due to system idles.

After having chosen the XID difference as an indicator, an intuitive method is to select the coldest record for swapping when a new write arrives. However, to identify the coldest record, this would require either scanning all the records at runtime or dynamically maintaining an index with some additional data structure. To deal with this issue, we propose *probabilistic record swapping*, which attempts to effectively discover the cold spots with limited overhead. The idea is as follows. We maintain a *swap pointer* in DRAM, which points to the next record to be considered for swapping. Upon the arrival of a new write request, we check the record in PCM pointed to by the current *swap pointer*. Let δ be the

Algorithm 4 Wear-Leveling Logic in `PCMLogging`

Procedure: Wear-Leveling (New Update Record r, Swapping Threshold δ)

```
X_t ← XID of the record t pointed by swap pointer
X_c ← XID of the system so far
Compute Pr(Swapping) based on X_t, X_c and δ
Generate a random number θ between 0 and 1
if Pr(Swapping) > θ then
    Move record t to a free address of PCM
    if t is a committed record then
        Assign t with a new pseudo XID
    Write new record r to the address of t
else
    Write new record r to a free address of PCM
Update the Mapping Table
Advance swap pointer to the next record in PCM
```

system-specified swapping threshold and t be the record being checked. We compute the difference between t's XID, X_t, and the system's current XID, X_c. If $X_c - X_t \geq \delta$, t must be swapped with the new write. However, a single swapping threshold could potentially incur a long delay in identifying a cold record if they are clustered. To ease this problem, when $X_c - X_t < \delta$, we use a probability of $(X_c - X_t)/\delta$ to decide whether to swap the two records:

$$Pr(Swapping) = \begin{cases} 1 & X_c - X_t \geq \delta; \\ \frac{X_c - X_t}{\delta} & X_c - X_t < \delta. \end{cases}$$

The *swap pointer* will be advanced to the next record in PCM after checking. The detailed algorithm of wear leveling is formally described in Algorithm 4 and illustrated in Figure 5. Note that if the swapped record t has been committed, it will be assigned a new pseudo XID (i.e., the largest XID that has been used but not in the ActiveTxList) after being moved to the new address. This facilitates the wear-leveling algorithm to compute its residence time at the new address while keeping the committed status. On the other hand, we expect that the chance of a swapped record belonging to an in-progress transaction is very small, according to our probabilistic swapping model.

Finally, we give a simple cost and benefit analysis for the proposed algorithm. Regarding the cost, there is no overhead on space and only a negligible computational cost to compute the swapping probability. Record swapping introduces some additional write traffic. Nevertheless, as the threshold δ is tunable, the system can choose an appropriate setting to strike a good balance between the total traffic and the wear leveling of writes (e.g., a higher δ setting may lead to a less even write distribution with a lower traffic overhead). Regarding the benefit, we avoid the case that a cold record occupies an address for an excessively long time, thereby evening out the write distribution. Moreover, we distribute the incoming traffic which potentially contains hot records to cold addresses. As will be shown in the experiments (Section 6), our algorithm improves the PCM lifetime by 5X, even with about 34% additional write traffic.

5 Destaging Algorithm

In our PCMLogging scheme, we need to migrate cached records to external storage when PCM runs out of space. In this section, we explore this destaging problem for PCMLogging. We first discuss the design trade-offs of the destaging process. To fully utilize the disk bandwidth, we then develop a cost model to adaptively determine the destaging speed based on the system workload.

5.1 Algorithm Overview and Design Trade-offs

The destaging process selects and migrates some of the committed records cached in PCM back to the external disk, based on specific selection criteria such as LRU. To increase the efficiency of this process, we perform destaging only when the PCM is close to full or the disk system has available bandwidth. When destaging starts, we need to decide "how much bandwidth to spend on destaging." A naive approach is that when the PCM reaches a certain occupancy rate, we devote 100% of the bandwidth to destaging. After the PCM occupancy rate becomes lower than the threshold, the destaging process stops. However, we observe that an ill-advised setting may greatly deteriorate the system performance. If the destaging is executed too lazily, the PCM may overflow and hence the new writes might be blocked to wait for free space allocation. On the other hand, if the destaging is executed too aggressively, it may reduce the hit rate of PCM and increase the I/O contention for normal disk reads. In either case, the system throughput and transaction response time could be significantly degraded.

Thus, the design goal of our destaging algorithm is to optimize the overall system performance by adaptively allocating I/O bandwidth to destaging. Furthermore, the destaging process in our PCMLogging scheme poses a unique requirement. Instead of simply writing back a committed record to the external disk, we need to reload the target page of the record and perform a merge operation before writing it back. In the literature, the destaging problem has been studied for disk arrays more than a decade ago, where updating of data or parity is destaged from the write cache to the disk array. However, the existing solutions such as *high-low watermark* [22] and *linear threshold* algorithms [29] are all based on heuristics. For example, in the *high-low watermark* algorithm, destaging starts when an upper threshold is reached and stops at a lower threshold; in the *linear threshold* algorithm, a set of linearly increased destaging rates is arranged according to the occupancy rate of the staging area. In the next section, we develop a cost model for the destaging process of PCMLogging and dynamically determine its optimal rate based on the current workload.

5.2 Cost Model

To find the optimal destaging rate, we develop a cost model based on Markovian birth-death queueing system. We assume that the PCM space is divided into *segments* (each segment may accommodate a number of records). The destaging process starts when there are $M - 1$ segments of free space left.

We model the system states when there are $M, M-1, \cdots, 1, 0$ segment(s) of free space, denoted by states $0, 1, 2, \cdots, M$, respectively. We also model the system states, $M+i$ ($i = 1, 2, \cdots$), when the amount of pending writes exceeds the PCM capacity by i segments of space. Formally, we define the following notations:

- P_k: probability of the PCM being in state k.
- λ_k: birth rate of incoming traffic from state k to $k+1$ (i.e., the speed of new record writes from DRAM).
- μ_k: death rate of destaging traffic from state $k+1$ to k (i.e., the speed of destaging records from PCM to disk).

We assume that arrivals of both birth events and death events follow a Poisson distribution. Each birth rate of λ_k can be directly measured at runtime. It is determined by the arrival rate of the incoming traffic and the PCM write hit rate. For the death rates of μ_k's, the system can specify any pattern of destaging rates, such as linearly-increased rates. Without loss of generality, we use $f(k)$ and $g(k)$ to denote the relationships between the rates of state k ($k \leq M$) and those of state 0. Note that $f(k)$ can be measured and determined during system warm-up, and $g(k)$ is a configurable parameter. After state M, we assume that the birth rate remains at $f(M)\lambda_0$ (as the PCM hit rate remains constant) and the death rate is set at the highest possible I/O rate μ_{max} (to recover from the overflow state as soon as possible). More specifically, we have the following relationships:

$$\lambda_k = \begin{cases} f(k)\lambda_0 & 0 \leq k < M; \\ f(M)\lambda_0 & k \geq M. \end{cases}$$

$$\mu_k = \begin{cases} g(k)\mu_0 & 0 \leq k < M; \\ \mu_{max} & k \geq M. \end{cases}$$

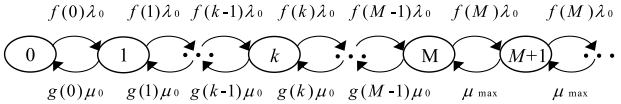


Fig. 6. State transition diagram

The state transition diagram for the birth-death queue of our system is illustrated in Figure 6. For simplicity, we only consider the I/O cost for each state. Because the I/O cost for destaging is proportional to the death rate, we can formulate the overall cost of destaging as the summation of the death rates weighted by the probability of each state:

$$Cost_{destaging} = \sum_{k=1}^{M} \mu_{k-1} P_k + \sum_{k=M+1}^{\infty} \mu_{max} P_k. \quad (1)$$

Next, we show, given a birth rate λ_0, $f(k)$'s and $g(k)$'s, how to determine the optimal μ_0 that minimizes the above overall cost. When the system is in an equilibrium state, the following two conditions hold:

$$\sum_{k=0}^{M} P_k + \sum_{k=M+1}^{\infty} P_k = 1. \quad (2)$$

$$P_k = \frac{\lambda_{k-1}}{\mu_{k-1}} P_{k-1}. \quad (3)$$

To simplify the equations, we denote $\rho = \frac{\lambda_0}{\mu_0}$ and $a = \frac{\lambda_M}{\mu_{max}}$. We can rewrite Eq. (3) as follows:

$$\begin{aligned} P_k &= P_0 \prod_{i=0}^{k-1} \frac{f(i)}{g(i)} (\frac{\lambda_0}{\mu_0})^k \\ &= P_0 \rho^k \left(\prod_{i=0}^{k-1} \frac{f(i)}{g(i)} \right), \qquad 1 \leq k \leq M. \quad (4) \\ P_k &= \left(\frac{\lambda_M}{\mu_{max}} \right)^{k-M} P_M \\ &= P_0 a^{k-M} \rho^M \left(\prod_{i=0}^{M-1} \frac{f(i)}{g(i)} \right), \quad k \geq M+1. \quad (5) \end{aligned}$$

Substituting P_k's in Eq. (2), we have:

$$P_0 + P_0 \sum_{k=1}^{M} \left(\rho^k \prod_{i=0}^{k-1} \frac{f(i)}{g(i)} \right) + P_0 \frac{a\rho^M}{1-a} \left(\prod_{i=0}^{M-1} \frac{f(i)}{g(i)} \right) = 1. \quad (6)$$

We can observe that Eq. (6) contains only two variables ρ and P_0. As we will show below, the objective function (1) can also be expressed by these two variables. Note that $\rho < 1$. Thus, by resolving Eq. (6), we can pre-compute a set of P_0's for all possible ρ values, e.g., all values between 0 and 1 with a small interval of 0.001. Then, we iterate all pairs of ρ and P_0 to find the minimal cost given by the objective function (1):

$$\sum_{k=1}^{M} \mu_{k-1} P_k + \sum_{k=M+1}^{\infty} \mu_{max} P_k =$$

$$P_0 \sum_{k=1}^{M} \left(\lambda_0 g(k-1) \rho^{k-1} \prod_{i=0}^{k-1} \frac{f(i)}{g(i)} \right) + P_0 \frac{\lambda_M \rho^M}{1-a} \left(\prod_{i=0}^{M-1} \frac{f(i)}{g(i)} \right)$$

After determining ρ, we can obtain the optimal $\mu_0 = \lambda_0/\rho$ based on the λ_0 value currently measured. Thus, the destaging rate for each state can be decided accordingly by $\mu_k = g(k)\mu_o$.

6 EXPERIMENTS

In this section, we evaluate the performance of our proposed PCMLogging scheme. A description of the experimental setup is followed by a thorough comparison of PCMLogging against existing logging schemes. Finally, as flash memory is becoming a competitive external storage to HDDs, we explore the performance of PCMLogging on flash memory devices.

6.1 Experiment Setup

We have developed a trace-driven simulator based on DiskSim [33]. We implemented a transaction processing model and a PCM model on top of simulated disks. In the transaction processing model, we employed the strict two-phase locking protocol for concurrency control at the record level. Deadlocks were prevented by the "wait-die" protocol: each transaction is given a timestamp when it starts. The older transaction is allowed to wait for a lock being held by a younger transaction. A younger transaction, on the other hand, is forced to abort, when requesting a lock being held by an older transaction [27], and will be restarted after the older transaction is completed.

In the PCM model, as with [6][8], we simulated 4 PCM chips, which could serve at most 4 concurrent requests. Recall that a variable-size record could occupy one or more PCM units. We set the data access unit for PCM to 128B, because the average record length in our trace is 117B. We set its write latency to $1\mu s$ and its read latency to the same quantity as that of DRAM access (i.e., $50ns$).

For HDD-based simulation experiments, we configured the read/write latency the same as the IBM DNES-309170 hard disk without write cache. For SSD-based simulation experiments, we adopted the device configuration specified by [23], [24], i.e., 32GB SSD with 8 fully connected 4GB flash packages.

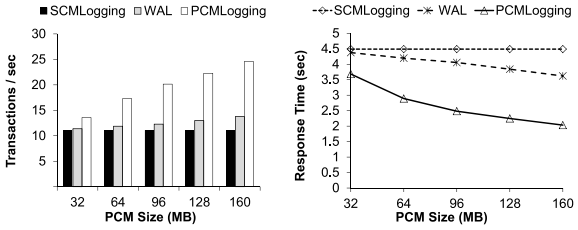
Our evaluation is based on the TPC-C benchmark [38], which represents an on-line transaction processing workload. To obtain the workload trace, we ran the DBT2 [35] toolkit to generate the TPC-C transaction SQL commands, which were then fed to PostgreSQL 8.4 [36]. In the generator, by default, we set the number of clients to 50 and the number of data warehouses to 20. We recorded both the transaction semantics (BEGIN/COMMIT/ABORT) and I/O requests at the record level in PostgreSQL. For simplicity, we assume that a database record has a constant size and fits into one or more PCM units by including its metadata.

We compared three logging schemes: our record-level PCM-Logging scheme (denoted as PCMLogging), the WAL design of using PCM for both data caching and transaction logging (detailed in the Introduction section, denoted as WAL), and a recent proposal of using storage-class memory such as PCM for logging only (denoted as SCMLogging) [8]. Note that the page-level PCMLogging scheme was not evaluated as it does not support record-level concurrency control. For a fair comparison using PCMLogging, the main memory area included the space needed for holding the data structures of PCM management (such as Mapping Table and Dirty Page Table). For the destaging process of PCMLogging, we assumed the rate relationship function $g(k) = 1$ for simplicity ($f(k)$ was measured during warm-up). For WAL, we used for simplicity the whole PCM space for data caching and reserved an additional page for archiving log records, though this might give performance advantages to the WAL scheme. For SCMLogging, the log records were created and maintained in PCM directly. In general, the WAL and SCMLogging schemes represent two typical usages of PCM, i.e., maximizing its caching or logging capabilities. In both of these schemes, the log records archived in PCM can be flushed out to external disks asynchronously.

We conducted the simulation experiments on a desktop computer running Windows 7 Enterprise with an Intel i7-2600 3.4GHz CPU. We focus on disk-based databases, and as in the previous work [3], [13], [19], we set the buffer size to be 5%-15% of the database size. By default, as we simulated a database of 2.4GB, the default sizes of DRAM and PCM were both set at 128MB (i.e., total 10% of the database size). For all schemes, the results were collected after the system reaches the stable state. The system performance was measured for the same number of transactions (i.e., 100,000 transactions). Table 2 summarizes the settings of our simulation experiments.

TABLE 2
Default Parameter Settings

Parameter	Default Setting
HDD read/write latency	$8.05ms$/$8.20ms$
SSD read/write latency	$25\mu s$/$50\mu s$
SD card read/write latency	$1.47ms$/$200.1ms$
PCM write latency	$1\mu s$
Logical page size	$8KB$
PCM unit size	$128B$
TPC-C database size	$2.4GB$
TPC-C client number/warehouse number	50/20
Main memory (DRAM) size	$128MB$
PCM size	$128MB$



(a) Transaction throughput (b) Transaction response time

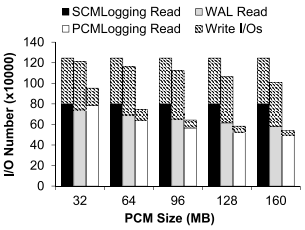
Fig. 7. Overall performance results

6.2 Overall Performance Comparison

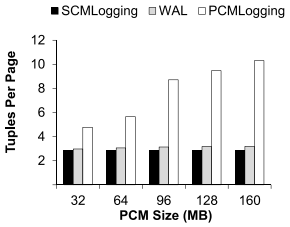
In this section, we report the overall comparison of PCMLogging with WAL and SCMLogging. We plot the transaction throughput and response time of the three schemes in Figures 7(a) and 7(b), respectively, by varying the size of PCM from 32MB to 160MB. We make the following observations from the results. First, WAL has a better performance than SCMLogging in all cases tested. This is mainly because applying WAL, PCM is not only used for logging but also for data caching, which makes its I/O cost less than that of SCMLogging (see Figure 8(a)). Second, PCMLogging has the best performance among all the three schemes. In Figure 7(a), the throughput improvement of PCMLogging over WAL increases from 19.2% to 78.3%, as the PCM size grows. A similar trend is observed for the response time in Figure 7(b). This confirms our argument that PCMLogging can better exploit the PCM hardware features for superior performance improvements. Next, we reveal more details of the PCMLogging performance from various perspectives.

I/O Breakdown

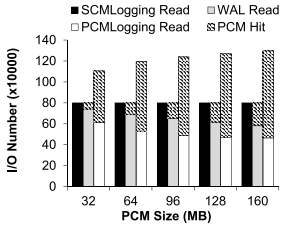
We decompose the total I/O number into reads and writes and plot the I/O breakdown in Figure 8(a). Obviously, the total I/O number of SCMLogging remains the same under different PCM sizes, because it writes only log records to PCM. In contrast, due to data caching in PCM, the total I/Os of WAL and PCMLogging decrease as the size of PCM grows. In particular, compared with WAL, PCMLogging saves 21.5%$\sim$46.6% of total I/Os under different PCM sizes. The saving is mainly due to reduced write I/Os (the upper part of each bar). For example, when the PCM size is 128MB, the number of write I/Os of PCMLogging is only 12.7% of that of WAL.



(a) I/O breakdown



(b) Dirty records per page



(c) Read I/Os and PCM hits

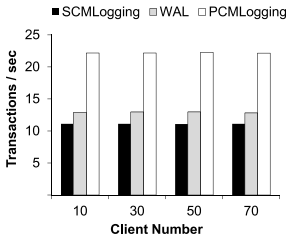
Fig. 8. I/O breakdown comparison

To gain more insights, we further measure the average number of dirty records per page for each write I/O operation and plot the results in Figure 8(b). This number indicates the efficiency of flushing dirty information from DRAM/PCM to external disk. The higher this number, the less write I/Os are required for handling the same workload. As illustrated by caching dirty pages in PCM, WAL has up to 11.1% more dirty records per page than SCMLogging. As for PCMLogging, by caching dirty records in PCM, it further improves this number to 60.4%$\sim$223% compared to WAL. This result reveals that PCMLogging can collect a relatively larger number of dirty records for each write I/O, thereby reducing overall write I/Os.

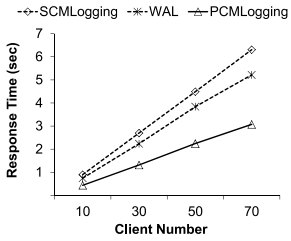
On the other hand, as shown in Figure 8(a), the improvement of PCMLogging in read I/Os is not as much as that in write I/Os. This is partly because PCMLogging incurs extra page read I/Os (to merge dirty records with original page contents) when destaging records from PCM to external disk (recall Section 5). Nevertheless, when the PCM size is larger than 64MB, this overhead is outweighed by the benefit due to data caching in PCM. To observe this effect in Figure 8(c), we plot the number of actual read I/Os (the bottom part of each bar, excluding the extra I/Os due to destaging) and the number of PCM hits (the upper part of each bar). Note, each bar represents the total number of DRAM misses under a particular PCM size setting. For SCMLogging, the read I/Os correspond to the DRAM misses, because it has no PCM cache. For WAL, the larger PCM, the more PCM hits; up to 27.3% of the DRAM misses are satisfied by the cached pages in PCM. Although buffer management combined with PCMLogging experiences the highest number of DRAM misses, it causes the least number of read I/Os (to disk), because most of its DRAM misses are satisfied by the cached records in PCM.

Impact of Concurrent Execution

We now investigate how concurrency of transaction execution would influence system performance. Figures 9(a) and 9(b)



(a) Transaction throughput



(b) Transaction response time

Fig. 9. Impact of concurrent execution

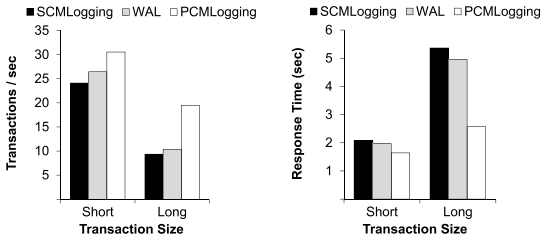
show transaction throughput and response time of the three schemes, respectively, by varying the number of clients in the trace generator. While the average response time increases with the client number for all three schemes, PCMLogging has the least degradation in response time (Figure 9(b)). The reason is as follows. With a larger number of clients, the lock contention becomes higher. As PCMLogging eliminates some time-consuming operations such as dirty-page replacement, each transaction holds its locks for a shorter time than that of WAL and SCMLogging. Consequently, PCMLogging has a better scalability to support more concurrent transactions with less penalty in response time. On the other hand, as shown in Figure 9(a), the system throughput remains almost unchanged for different client numbers, because the system is bounded by the I/O performance.

Impact of Transaction Size

Next, we reveal the impact of transaction size (i.e., the number of pages updated by a transaction) on the system performance. To do so, we split the original trace of transactions into two sub-traces according to the transaction size (denoted by m): short transactions ($m \leq 10$) and long transactions ($m > 10$). From Figures 10(a) and 10(b), we can observe that the performance improvement of PCMLogging over WAL and SCMLogging is higher for long transactions. For example, PCMLogging outperforms WAL by 15.5% in terms of throughput for short transactions, and this improvement increases to 87.7% for long transactions. This can be explained as follows. For a long transaction, it is more likely to have a larger amount of dirty page replacements than a short transaction has. As discussed earlier, PCMLogging alleviates lock contention and saves write I/Os by caching and logging dirty records in PCM. Consequently, PCMLogging favors long transactions by a higher performance improvement than short transactions.

Recovery Performance

For the WAL scheme, we used PostgreSQL to simulate its recovery time. We set the checkpoint interval to be the default setting (i.e., 5 minutes). We randomly injected 10 failures during the experiment. The average recovery time is 3.2 seconds. In contrast, using our PCMLogging, the average recovery time is 19 ms only. This is because the recovery process of WAL must involve I/O operations, whereas in PCMLogging we only need to scan the PCM and discard the uncommitted records without incurring any I/O operations.



(a) Transaction throughput (b) Transaction response time

Fig. 10. Impact of transaction size

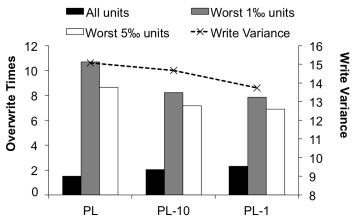


Fig. 11. Wear-leveling performance

6.3 Wear-Leveling Performance

In this section, we examine the PCM write performance and evaluate our proposed probabilistic record-swapping wear-leveling algorithm. We compare PCMLogging without wear-leveling mechanism (denoted as PL) and with different swapping thresholds (denoted as PL-δ, where $\delta \times 10,000$ is the swapping threshold). In Figure 11, we show the average number of overwrite times for all PCM units, the worst 1‰ and 5‰ PCM units, as well as the variance of the write distribution over all PCM units. Two observations are made. First, by applying our weal-leveling algorithm, although PL-10 and PL-1 increase the total write traffic by 34.2%∼51.5%, the average write traffic for the worst 1‰ and 5‰ PCM units is reduced by up to 26.4% and 23.0%, respectively. As the worst PCM units decide the PCM lifetime, we believe this is worthwhile even at the cost of increased total traffic. Also, the time of PCM writing (including the swapping overhead incurred for wear-leveling) is negligible, accounting for less than 0.2% of the total response time. Second, comparing PL-10 and PL-1, a smaller swapping threshold has 12.9% more traffic overhead. However, the variance of the write distribution is decreased by 8%, and the overwrite traffic of the worst 1‰ and 5‰ PCM units is decreased by 4.6% and 3.6%, respectively. As discussed in Section 5, a smaller swapping threshold results in more swap operations, which leads to more evenly distributed traffic.

6.4 Destaging Performance

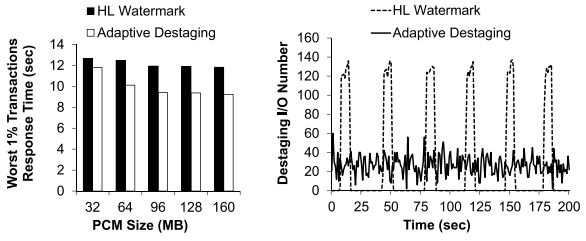
To evaluate the performance of our cost-based adaptive destaging algorithm, we compare it with the *high-low watermark* destaging algorithm [22]. In the high-low watermark algorithm, the destaging process is triggered by an upper threshold of the PCM occupancy rate and terminated by a lower threshold. In the experiments, we set the upper threshold to be 99% of the PCM size to reserve some free space for transaction commit. The lower threshold is set at 97.5%. Our adaptive destaging algorithm starts the destaging process when there is about 2.5% of free PCM space left, and the segment size was set to 640KB by default. With these parameter settings, caching performance would not be too much affected by the destaging process.

Because destaging operations may cause sudden overhead on some transactions, we investigate the performance of the worst 1% transactions in terms of response time. As shown in Figure 12(a), the adaptive algorithm shortens the worst response time of the high-low watermark algorithm by 6.8%∼21.9%. To explore the cause of performance improvement, we recorded the destaging I/O numbers of the two algorithms for a time period under the default system settings and plot the results in Figure 12(b). We can observe that the destaging I/Os of the high-low watermark algorithm are quite wavy, which may delay the transaction execution from time to time. However, in our adaptive algorithm, the destaging I/O pattern is more stable. As the PCM size increases, the transactions in the high-low watermark algorithm are still largely affected by the skewed I/O pattern. In contrast, our cost-based destaging algorithm adaptively distributes the destaging workload and reduces the worst transaction response time by up to 21.9%.

6.5 Additional Experiments on Other Datasets

As a complement to the experiment results so far, this section presents some additional results obtained from other datasets and system settings.

In Figure 13, we present the results of experimenting logging schemes on a widely adopted telecom workload benchmark, TM1 [39]. The trace generation and the experimental parameters were set the same as those used in the TPC-C experiments. As we can observe from the figure, PCMLogging outperforms SCMLogging and WAL by up to 110% and 67%,



(a) Response time (b) Destaging I/O pattern

Fig. 12. Destaging performance

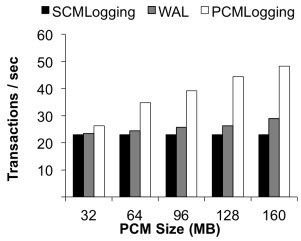


Fig. 13. Performance results on the TM1 dataset

respectively. PCMLogging retains its advantages over the other logging schemes for this benchmark.

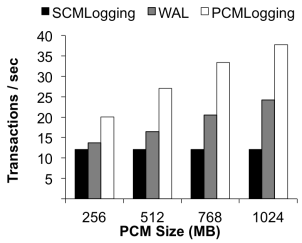


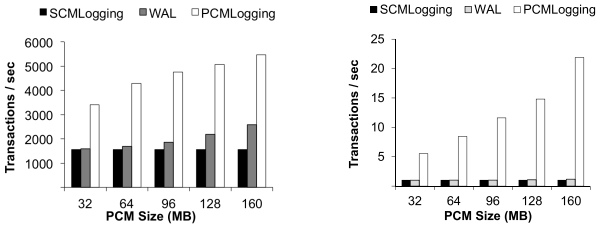
Fig. 14. Performance results on a larger TPCC dataset

We have also conducted experiments under the setting of a more powerful system. We collected a TPC-C trace from 200 warehouses with 50 clients, where the lock contention has less impact on transaction execution. The size of DRAM was set as 5% of the size of the dataset; i.e., they were set at 1GB and 20GB, respectively. The evaluation results with varying PCM sizes are shown in Figure 14. For WAL, we allocated 1MB PCM to be the log buffer (among which 512KB as the transfer size) and the rest PCM space as the data cache. Benefiting from the larger log cache, the performance of WAL slightly improves. However, the overall trend remains similar to that of the default case (Figure 7(a)), and PCMLogging still outperforms WAL by up to 56%.

6.6 PCMLogging Performance on Flash Memories

As the flash memory technology matures, various flash-memory-based devices have been available in the market. For examples, SSD devices have been deployed in desktop and server computers as external storage and secure digital memory (SD) cards have been widely adopted by smartphones to store application data. This section studies the performance of PCMLogging in SSD- and SD card-based database systems.

In Figure 15(a), we report the transaction throughput of the three schemes under evaluation in an SSD-based system. Figure 15(b) plots the experimental results of using a simulated Kingston SD card as external database storage (see Table 2 for detailed read/write latency settings). The performance improvement of PCMLogging against the other schemes is up to 2.1X and 20X in SSD- and SD card-based systems, respectively, which is much larger than the improvement observed in HDD-based systems (Figure 7). This can be explained as follows. First, compared to the write I/O cost, the destaging cost on SDD and SD card is relatively lower, because the merge process benefits from the fast random read of flash memory. Second, recall that PCMLogging greatly reduces the number of write I/Os by caching and logging dirty records. On flash memory, due to the asymmetric read/write performance, the savings on write I/Os lead to a larger gain in overall performance. This reason further explains why the performance improvement on SD card is even larger than that on SSD, because SD card has a much higher asymmetry in read/write latency.



(a) SSD (b) SD Card

Fig. 15. Transaction throughput on flash memories

7 Related Work

A long stream of research exists on transaction-oriented database logging and recovery [11] for which the existing algorithms can be classified into two categories, namely WAL [21] and shadow paging [10]. Using WAL, in-place update can be performed, i.e., a data page can be modified only after its old state has been logged, which enables the undo of the update during transaction rollback or system failures. In contrast, shadow paging handles data updates by out-of-place schemes. After an update, a page is written to a free block, leaving its old state as a shadow page on disk. Although both WAL and shadow paging have been implemented in real systems, shadow paging is not as popular as WAL for disk-based databases due to performance reasons [23], [27]. Over the past few decades, considerable research efforts have been devoted to the optimization of WAL performance, e.g., group commit [12], finer-grained logging [28], multi-core techniques [14], to name but a few.

Transaction processing schemes for NVRAM have been studied a long time ago. Agrawal and Jagadish [31] presented an idea of using NVRAM to support transaction logging based on shadow paging. However, there are several major differences between this previous work and our PCMLogging. First, the previous work assumed an NVRAM-only setting, where the entire main memory is non-volatile. In contrast, in our PCMLogging we consider PCM as a supplement to DRAM. Therefore, the logging protocol needs to be re-designed to work in such a hybrid memory setting. Second, in addition to the logging protocol, we address the wear-leveling and destaging problems of using PCMLogging, which are critical to system performance but were not discussed in the previous work. Third, we conduct extensive simulation experiments to validate the proposed scheme and reveal insightful findings, whereas the previous work did not report any performance evaluation result.

Recently, flash-memory-based SSDs have emerged as a competitive alternative of external storage. Lee and Moon [16] proposed a new In-Page Logging (IPL) scheme for reducing the database update cost on flash memory. IPL avoids direct updates to a page by logging the data changes in a reserved area of the flash block, and then performs erasure and merge operations until the log area becomes full. Chen [2] proposed *Flashlogging* to flush transactional log records to flash devices by exploiting their fast sequential write performance. In a previous contribution [23], we proposed a *FlagCommit* protocol for flash-based databases, where the partial-page-programming

feature of flash memory is exploited to optimize the transaction recovery performance.

More recently, as one of the most promising next-generation memory technologies, PCM has drawn attention from various research aspects. The hardware-level optimizations have been focused on improving the write performance and the lifetime of PCM. Several wear-leveling techniques based on the idea of randomizing address-to-frame mappings have been proposed [15], [17]. Software techniques have also been developed to take the best of both PCM and DRAM. Condit *et al.* proposed a file system called BPFS [6] based on the characteristics of byte-addressable and non-volatile memory (BPRAM). PCM can be viewed as one specific type of BPRAM, and a hybrid memory system of BPRAM and DRAM is adopted in [6].

Chen *et al.* presented a pioneer study on how database algorithms should be adapted to PCM technology in [4]. They improved two fundamental database algorithms (i.e., B+-tree and hash join), by reducing the write operations in PCM. Kim *et al.* [20] extended IPL to IPL-P by storing the logged changes in PCM. Regarding transaction processing, Fang *et al.* [8] used PCM as an asynchronous WAL pool. They discussed hardware features and OS interface support for PCM and addressed several issues resulting from the opportunity to directly write log records to PCM.

We remark that many of the techniques developed in this paper have been inspired by research related to shadow paging [10] and finer-grained logging [28]. However, to the best of our knowledge, this is the first complete work that is dedicated to developing transaction logging and recovery schemes for PCM-assisted database systems.

8 Conclusions and Future Work

In this paper, we have presented a study on leveraging PCM to support efficient transaction logging and recovery. We have developed an effective yet simple PCMLogging scheme that integrates the log and cached updates, by taking advantage of the PCM hardware features. To address the PCM endurance issue, we have proposed a probabilistic wear-leveling algorithm that proactively migrates the cold records in PCM. Furthermore, we have developed a cost model to adaptively adjust the speed of data destaging from PCM to external disks. The experiments based on TPC-C benchmark have demonstrated a significant performance improvement of our PCMLogging scheme compared to WAL and SCMLogging. It not only outperforms WAL by up to 1$\sim$20X in terms of transaction throughput and response time, but also prolongs the PCM lifetime due to reduced traffic and wear leveling of write operations. The performance improvement is observed to be even higher when we employ flash memories such as SSD and SD card as external database storage. As for future work, we plan to implement the PCMLogging scheme in an open-source database management system. It is of particular interest to incorporate this scheme into SQLite, a database widely used on smartphones, because PCM has already been deployed on mobile handsets. We are also going to further improve the scheme in various directions, such as advanced PCM replacement policies, support of index structures, and integration with multi-version concurrency control.

Acknowledgment

A preliminary version of this paper was presented at the 20th ACM International Conference on Information and Knowledge Management (CIKM) [9]. This work was supported by the German-Hong Kong Joint Research Scheme (Grant G_HK018/11) and the Research Grants Council of Hong Kong SAR, China (Grants 211510 and HKBU211512).

References

[1] S. Byun, "Transaction management for flash media databases in portable computing environments", *J. Intell. Inf. Syst.*, vol. 30, pp. 137–151, 2008.

[2] S. Chen, "Flashlogging: exploiting flash devices for synchronous logging performance", in *Proc. SIGMOD*, pp. 73–86, 2009.

[3] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica, "TPC-E vs. TPC-C: characterizing the new TPC-E benchmark via an I/O comparison study", *SIGMOD Record*, pp. 5–10, 2010.

[4] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking database algorithms for phase-change memory", in *Proc. CIDR*, pp. 21–31, 2011.

[5] Y. Choi et al., "A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth", in *IEEE Int'l Solid-State Circuits Conf.*, 2012.

[6] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory", in *Proc. SOSP*, pp. 133–146, 2009.

[7] G. P. Copeland, T. W. Keller, R. Krishnamurthy, and M. G. Smith, "The case for safe RAM", in *Proc. VLDB*, pp. 327–335, 1989.

[8] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang, "High-performance database logging using storage-class memory", in *Proc. ICDE*, pp. 1221–1231, 2011.

[9] S. Gao, J. Xu, B. He, B. Choi, and H. Hu, "PCMLogging: Reducing Transaction Logging Overhead with PCM," in *Proc. ACM CIKM*, pp. 2401–2404, 2011.

[10] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, "The recovery manager of the system R database manager", *ACM Comput. Surv.*, vol. 13, pp. 223–242, 1981.

[11] T. Härder and A. Reuter, "Principles of transaction-oriented database recovery", *ACM Comput. Surv.*, 15(4): 287–317, 1983.

[12] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter, "Group commit timers and high-volume transaction systems", in *Proc. Int'l Workshop on High-Performance Trans. Systems*, pp. 301–329, 1987.

[13] W. W. Hsu, A. J. Smith, and H. C. Young. "I/O Reference Behavior of Production Database Workloads and the TPC Benchmarks - An Analysis at the Logical Level." in *UC Berkeley Report UCB/CSD-99-1071*, 1999.

[14] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki, "Aether: A scalable approach to logging", in *Proc. VLDB*, 2010.

[15] B. C. Lee et al., "Phase-change technology and the future of main memory", *IEEE Micro*, pp. 131–141, 2010.

[16] S. W. Lee and B. Moon, "Design of flash-based DBMS: An in-page logging approach", in *Proc. SIGMOD*, pp. 55–66, 2007.

[17] K. J. Lee et al., "A 90nm 1.8V 512Mb diade-switch PRAM with 226 MB/s read throughput", *IEEE J. Solid-State Circ.*, 43(1):150–162, 2008.

[18] D. Liu, T. Wang, Y. Wang, Z. Qin, and Z. Shao, "PCM-FTL: A write-activity-aware NAND flash memory management scheme for PCM-based embedded systems", in *Proc. RTSS*, 2011.

[19] M. Rosenblum and J. K. Ousterhout. "The Design and Implementation of a Log-Structured File System."in*ACM Trans. on Computer Systems* 1992.

[20] K. Kim, S.-W. Lee, B. Moon, C. Park, and J.-Y. Hwang, "IPL-P:In-page logging with PCRAM", in *VLDB*, 2011 (desmo).

[21] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging", *ACM Trans. Database Syst.*, vol. 17, pp. 94–162, 1992.

[22] Y. J. Nam and C. Park, "An adaptive high-low water mark destage algorithm for cached RAID5", in *Proc. PRDC*, 2002.

[23] S. T. On, J. Xu, B. Choi, H. Hu, and B. He, "Flag commit: Supporting efficient transaction recovery on flash-based DBMSs", *IEEE Trans. on Knowledge and Data Engineering*, 24(9): 1624-1639, 2012.
[24] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash", in *Proc. OSDI*, pp. 147–160, 2008.
[25] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high-performance main-memory system using phase-change memory technology", in *Proc. ISCA*, 2009.
[26] E. Rahm, "Performance evaluation of extended storage architectures for transaction processing", in *Proc. SIGMOD*, pp. 308–317, 1992.
[27] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. McGraw-Hill, 2003.
[28] R. Sears and E. Brewer, "Segment-based recovery: write-ahead logging revisited", in *PVLDB*, pp. 490–501, 2009.
[29] A. Varma and Q. Jacobson, "Destage algorithms for disk arrays with non-volatile caches", *IEEE Trans. on Computers*, pp. 83–95, 1995.
[30] C.-H. Wu, T.-W. Kuo, and L.-P. Chang, "Efficient initialization and crash recovery for log-based file systems over flash memory", in *Proc. SAC*, pp. 896–900, 2006.
[31] R. Agrawal and H. V. Jagadish, "Recovery algorithms for database machines with nonvolatile main memory", in *Proc. the Sixth International Workshop on Database Machines*, 1989.
[32] S. Akyurek and K. Salem, "Management of Partially-Safe Buffers", in *Proc. of the Fifth International Symposium on High Performance Computer Architecture*, 1993.
[33] Disksim, http://www.pdl.cmu.edu/DiskSim/.
[34] Micron, "Phase change memory", http://www.micron.com/products/phase-change-memory.
[35] OSDL Database Test 2. http://osdldbt.sourceforge.net.
[36] PostgreSQL, http://www.postgresql.org/.
[37] Samsung, http://www.samsung.com/us/business/semiconductor/news View.do?news_id=1149, 2010.
[38] TPC Benchmark C, Standard Specification. http://www.tpc.org/tpcc/spec/tpcc-current.pdf.
[39] Nokia Network Database. http://hstore.cs.brown.edu/wordpress/wp-content/uploads/2011/05/Nokia_TM1_Description.pdf.

Shen Gao is an MPhil student in the Department of Computer Science at Hong Kong Baptist University. He received his BSc degree in computing studies (information systems) from the same university. His research interest is on data management for next-generation storage devices.

Jianliang Xu is a Professor in the Department of Computer Science, Hong Kong Baptist University. He received his BEng degree in computer science and engineering from Zhejiang University, Hangzhou, China, in 1998 and his PhD degree in computer science from Hong Kong University of Science and Technology in 2002. He held visiting positions at Pennsylvania State University and Fudan University. His research interests include data management, mobile/pervasive computing, wireless sensor networks, and distributed systems. He has published more than 110 technical papers in these areas. He was a vice chairman of ACM Hong Kong Chapter and is a senior member of IEEE.

Theo Härder obtained his Ph. D. degree in Computer Science from the TU Darmstadt in 1975. In 1976, he spent a post-doctoral year at the IBM Research Lab in San Jose and joined the project System R. In 1978, he was associate professor for Computer Science at the TU Darmstadt. As a full professor, he is leading the research group DBIS at the TU Kaiserslautern since 1980. He is the recipient of the Konrad Zuse Medal (2001) and the Alwin Walther Medal (2004) and obtained the Honorary Doctoral Degree from the Computer Science Dept. of the University of Oldenburg in 2002. Theo Härders research interests are in all areas of database and information systems in particular, DBMS architecture, transaction systems, information integration, and XML database systems. He is author/coauthor of 7 textbooks and of more than 280 scientific contributions with $>$160 peer-reviewed conference papers and $>$70 journal publications. His professional services include numerous positions as chairman of the GI-Fachbereich Databases and Information Systems, conference/program chairs and program committee member, editor-in-chief of Computer Science Research and Development (Springer), associate editor of Information Systems (Elsevier), World Wide Web (Kluver), and Transactions on Database Systems (ACM).

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an assistant professor in Division of Computer Science, School of Computer Engineering of Nanyang Technological University, Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.

Byron Choi received the bachelor of engineering degree in computer engineering from the Hong Kong University of Science and Technology (HKUST) in 1999 and the MSE and PhD degrees in computer and information science from the University of Pennsylvania in 2002 and 2006, respectively. He is now an assistant professor in the Department of Computer Science at the Hong Kong Baptist University.

Haibo Hu is a research assistant professor in the Department of Computer Science, Hong Kong Baptist University. Prior to this, he held several research and teaching posts at HKUST and HKBU. He received his PhD degree in Computer Science from the Hong Kong University of Science and Technology in 2005. His research interests include mobile and wireless data management, location-based services, and privacy-aware computing.