PCMLogging: Reducing Transaction Logging Overhead with PCM

Shen Gao Jianliang Xu
Hong Kong Baptist University
{sgao, xujl}@comp.hkbu.edu.hk

Bingsheng He
Nanyang Technological University
bshe@ntu.edu.sg

Byron Choi Haibo Hu
Hong Kong Baptist University
{haibo, bchoi}@comp.hkbu.edu.hk

ABSTRACT

Phase Changing Memory (PCM), as one of the most promising next-generation memory technologies, offers various attractive properties such as non-volatility, bit-alterability, and low idle energy consumption. In this paper, we present PCMLogging, a novel logging scheme that exploits PCM devices for both data buffering and transaction logging in disk-based databases. Different from the traditional approach where buffered updates and transaction logs are completely separated, they are integrated in the new logging scheme. Our preliminary experiments show an up to 40% improvement of PCMLogging in disk I/O performance in comparison with a basic buffering and logging scheme.

Categories and Subject Descriptors

H.2.4 [Database Management]: Transaction Processing

General Terms

Algorithm, Performance

Keywords

Transaction Management, Logging, Database Recovery, PCM

1. INTRODUCTION

Over decades, non-volatile random access memory (NVRAM) has been actively investigated as the next-generation memory technology. The goal of the NVRAM technology is to take the best of the hard disk and DRAM: with non-volatility like the hard disk and with read/write speeds like DRAM. Recently, PCM has been considered one of the most promising NVRAM technologies. PCM is not only a persistent fast random-access memory but also has a higher density than the floating-gate based memory. Manufacturers have already started the mass production of PCM at a reasonable price. For example, Numonyx (later acquired by Micron) and Samsung released their first PCM products in 2010 [4, 5]. It has been envisioned that PCM will be integrated into the memory/storage hierarchy in the near future [2]. An important question brought by a recent research on using PCM in databases [1] is "how should database systems be modified to best take advantage of this emerging trend towards PCM?" In this paper, we study how PCM can be leveraged to improve the transaction processing performance in disk-backed relational databases.

Compared with DRAM and recently popular flash memory, PCM offers various attractive hardware features to database transaction processing. Compared with DRAM, PCM has the advantage of persistence with a comparable read speed. Compared with flash memory, PCM has over two orders of magnitude faster read/write speeds. Unlike flash memory with an erase-before-write constraint, PCM is bit-alterable without a separate erase step [5]. These unique hardware features of PCM motivate us to revisit and improve the core components in transaction processing.

Previous work [3] has demonstrated that logging is a critical component for transaction processing in disk-based databases. Therefore, we consider leveraging PCM to reduce the transaction logging overhead. In general, PCM can be used for both data buffering and transaction logging. Owing to its fast data access performance, dirty pages evicted from main memory can be buffered in PCM to minimize disk I/Os. Meanwhile, the non-volatile nature of PCM makes it an ideal place for archiving transaction log records. Hence, a simple design is to divide the PCM into a buffer pool and a log pool. However, this simple scheme has several drawbacks. First, buffering a full page in PCM may not be very cost-efficient since most updates in OLTP workload are small writes. Second, a transaction update kept in the log might also be buffered in a dirty page, which leads to data redundancy. Third, the space management of the PCM becomes a challenge as it is shared by the buffer pool and the log pool.

To address the drawbacks of the basic scheme, we propose a new logging scheme called PCMLogging. In this scheme, we eliminate the explicit logs, such as REDO and UNDO logs, by integrating implicit logs into buffered updates to ensure durability and atomicity of database transaction processing. Meanwhile, this makes checkpoint unnecessary and enables a simple and cheap recovery algorithm. We also develop a trace-driven simulator to study the performance of PCMLogging. The evaluation results based on a TPC-C benchmark trace show that, compared to the basic scheme mentioned above, PCMLogging saves up to 40% I/Os to the external disk and 97% write traffic to the PCM device.

The rest of this paper proceeds as follows. In Section 2, we give some background of PCM and review the related work. Section 3 presents the PCMLogging scheme. In Section 4, we report our performance evaluation results. Finally, Section 5 concludes the paper and points out some future directions.

2. BACKGROUND AND RELATED WORK

PCM is based on a new storage material called *chalocogenide glass*, which retains information by a large resistance contrast between two states, namely *amorphous* and *crystalline*. To switch between these two states, different kinds of electronic currents are applied on chalocogenide glass. To crystallize the phase change material, a *set* current pulse is applied to heat chalocogenide glass for a sufficiently long time. On the other hand, a *reset* current is a sudden pulse which melts chalocogenide glass to largely increase its resistance. Sensing the resistance and retrieving the information requires only a very low current.

Table 1 summarizes the hardware performance of several current storage technologies including DRAM, NAND flash, PCM, and hard disk [1]. Besides non-volatility and higher density,

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.

Parameter	DRAM	NAND Flash	Hard Disk	PCM
Density	1X	4X	N/A	2-4X
Read latency (granularity)	20-50ns (64B)	~25μs (4KB)	~5ms (512B)	~50ns (64B)
Write latency (granularity)	20-50ns (64B)	~500μs (4KB)	~5ms (512B)	~1μs (64B)
Endurance	N/A	10^4~10^5	∞	10^6~10^8

Table 1: Comparison of Storage Technologies [1]

PCM has the following hardware features [1, 2, 5]:

- *Finer-grained access granularity*. PCM has an access granularity as small as DRAM. Compared to other non-volatile memory technologies such as flash, it breaks the constraints of erase-before-write and page-based access. It is byte-addressable (or word-addressable) and bit-alterable. This feature enables PCM to support small in-place updates.
- *Asymmetric read/write latency*. The write speed of PCM is about 20 times slower than the read speed.
- *Endurance limitation*. Similar to flash memory, PCM endures a limit number of writes, on the scale of 10^6 to 10^8 writes on each unit.

Recently, system researchers have started revisiting system design and optimizations with PCM. Fig. 1 shows two representative architectures of using PCM in the memory hierarchy [2]: a) PCM as an auxiliary memory attached to the main memory; b) PCM as the whole main memory. Based on the storage properties of PCM, the former method using PCM as an auxiliary memory might be more practical. The first reason is that PCM has endurance limitation, and its write latency is still much higher than DRAM. Secondly, the capacity of PCM is still small in the coming few years, compared with DRAM. In this study, we mainly consider the memory architecture of using PCM as an auxiliary memory.

There has been research on how to take advantages of non-volatile memory for improving database performance. Early work focused on the use of non-volatile memory as an extension of the storage hierarchy [6]. As the technology matures, in 2011, Chen *et al*. [1] presented a pioneer study on how database algorithms should be adapted to PCM technology. They improved two fundamental database algorithms, *i.e.*, B+-tree and hash join, by reducing their write operations to PCM. In contrast, in this study we focus on how PCM can be used to improve transaction processing performance in disk-based databases.

3. PCMLOGGING

We consider the memory architecture as shown in Fig. 1(a). Without modifying the buffer manager of the main memory (DRAM), we develop a new scheme, called PCMLogging, where the buffered updates and transaction logs are combined.

3.1 Overview

The basic idea of PCMLogging is to integrate the transaction logs into buffered updates, by exploiting the persistency feature of PCM storage. For ease of exposition, we assume that the PCM buffering granularity is a page in this section. We also assume that the concurrency control is on a page level. That is, a page is updated by at most one transaction at one time. To further improve the transaction processing performance, we extend the design to tuple-based buffering and tuple-level concurrency control in Section 3.3.

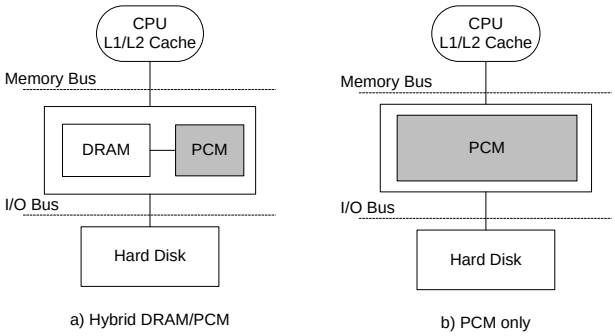


Fig. 1: Memory Architecture Alternatives

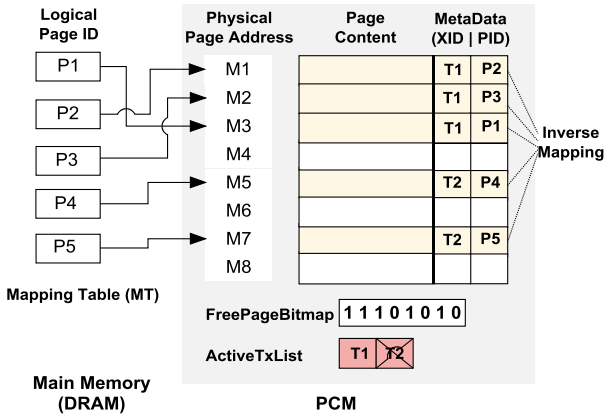


Fig. 2: Page Format and Mapping Table

To support buffering in PCM, we maintain a *Mapping Table* in the main memory to map logical page IDs to physical PCM addresses, as well as *Inverse Mapping* and *FreePageBitmap* in the PCM (see Fig. 2). Inverse Mapping is used to construct the Mapping Table at the boot time and FreePageBitmap is used to keep track of the free page slots. Note that in PCMLogging, only dirty pages evicted from the main memory are buffered in order to minimize the disk write I/Os.

An overview of the PCMLogging scheme is as follows. To achieve atomicity and durability of transactions, an *ActiveTxList* is maintained in PCM to record the in-progress transactions that have dirty pages buffered in the PCM; each buffered page records the XID of the last transaction that caused the page to be dirty. To guarantee atomicity, before the first dirty page of a transaction is written to the PCM, its corresponding XID should be recorded in the ActiveTxList. The XID is not removed until the transaction is requested to commit and all its dirty pages have been flushed to the PCM. Thus, during recovery, if the XID of a transaction is found in the ActiveTxList, it implies that the transaction is not committed before the crash; otherwise, the transaction is committed. Consequently, each PCM page can be recovered according to the status of the corresponding transaction. For example, if the PCM appears as shown in the right part of Fig. 2, during recovery, we can infer that *T1* is not yet committed while *T2* is committed. Thus, the pages stored in *M1*~*M3* would be discarded while the pages stored in *M5* and *M7* would be retained. Accordingly, the FreePageBitmap should be updated to "00001010". We note that, to avoid the hot-spot on the PCM, dynamic space allocation and wear-leveling techniques can be adopted to evenly distribute writes across the PCM space, which is, however, an orthogonal issue to our PCMLogging scheme.

As can be seen, PCMLogging totally eliminates the explicit transaction logs by integrating them into the dirty pages buffered in the PCM. This integrated design has several advantages. First, the data redundancy between buffered updates and transaction logs is minimized. Second, it avoids the challenging space management issue, which is a must if they are separated. Third, checkpoint becomes unnecessary since we do not maintain explicit logs. In addition, the recovery process becomes extremely simple and efficient. In the following, we detail the PCMLogging operations such as flushing, commit, abort, and recovery.

3.2 PCMLogging Operations

In PCMLogging, durability is achieved by forcing the corresponding dirty pages to the PCM when a transaction is requested to commit. Meanwhile, a *steal* buffer policy of the main memory allows a dirty page to be flushed to the 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 the main memory:

- *Transaction Table (TT)*. This table records all in-progress transactions. For each in-progress transaction, it keeps track of all its dirty pages stored in the main memory and PCM. The purpose is to quickly identify the relevant pages when the transaction is committed or aborted.
- *Dirty Page Table (DPT)*. This table keeps track of the previous version for each PCM page "overwritten" by an in-progress transaction. This is necessary for restoring the original page content 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.

Flushing Dirty Pages to PCM. When the main memory becomes full or a transaction is committed, some dirty pages may need to be flushed to the PCM. We first check the ActiveTxList in the PCM. For each dirty page, if the corresponding XID is not yet in the ActiveTxList, we add it to the list before flushing. In case a page *P* already exists in the PCM, we do not overwrite it in place. In order to support undo, instead, we create an out-of-place copy *M'* with the current timestamp. Then, *M'* is added to the Dirty Page Table and *P* is mapped to *M'* in the Mapping Table. Finally, the Transaction Table is updated.

Commit. Upon receiving a commit request, we first force all its dirty pages that are still buffered in the main memory to the PCM, by checking the Transaction Table. After that, we remove the XID from the ActiveTxList and indicate the transaction is committed. Next, if any of its pages is contained in the Dirty Page Table, the previous version is discarded. 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, by checking the Transaction Table. If any of its pages is contained in the Dirty Page Table, the previous version should be restored. Finally, we clear the XID in the ActiveTxList and the relevant entries in the Transaction Table and Dirty Page Table.

An Example: *Consider the example shown in Fig. 2, where T1 is in progress and T2 is committed. Suppose now a new transaction T3 updates the page P5. Before this dirty page is flushed, T3 points to the page P5 kept in the main memory (see Fig. 3(a)). When it is flushed to the PCM slot M8, T3 is added to the ActiveTxList in PCM (see Fig. 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 requested to commit, it is removed from the ActiveTxList; the previous version is discarded (the corresponding bit becomes 0 in the FreePageBitmap); and the relevant entries are removed from the Transaction Table and Dirty Page Table (see Fig. 3(c)). Otherwise, if T3 is finally aborted, the current version is discarded (the corresponding bit becomes 0 in the FreePageBitmap) and the previous version is restored in the Mapping Table; and the relevant entries are also removed from the ActiveTxList, Transaction Table, and Dirty Page Table (see Fig. 3(d)).*

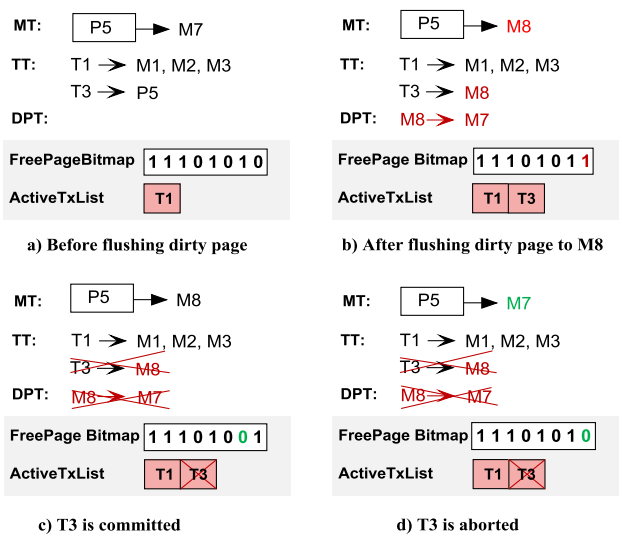


Fig. 3: An Example of PCMLogging

Recovery. A recovery process is invoked when the system restarts after a failure. It identifies the last committed version for each PCM page and constructs the Mapping Table. To do so, the recovery reads all valid pages based on the FreePageBitmap and the XIDs of the in-progress transactions from the ActiveTxList. We then simply discard the pages that belong to the in-progress transactions. Note that this process does not involve any disk I/Os.

Replacement in PCM. When the PCM is full (or the disk system is idle), we will select some committed pages and write them back to the external disk. Thus, some replacement policy (*e.g.*, LRU) can be used to select the victims.

3.3 Tuple-based Buffering

Recall that PCM supports byte-addressability and bit-alterability. Thus, in response to small writes in OLTP workload and tuple-based concurrency control, we propose to buffer dirty tuples, instead of dirty pages, in the PCM. The proposed PCMLogging scheme can be easily extended to tuple-based buffering. It works similarly except the following modifications: 1) In the PCM, the buffer slots should now be managed in the unit of tuples, rather than pages. To manage the free space, if the tuples are of variable size, a bitmap does not work; instead we may employ a slotted directory. 2) In the Mapping Table, we still keep track of dirty pages, but maintain the mappings for the buffered tuples in each dirty page. 3) If a read/write request is on a tuple granularity, the tuple can be accessed from the PCM directly, if available. Otherwise, we have to load the corresponding page from the external disk and merge it with the latest contents of the buffered tuples in the page. Note that loading page contents from the external disk and the PCM is a parallel process and the access latency of the PCM is negligible. 4) When a committed tuple is moved from the PCM to the external disk, we have to first load

the corresponding page from the external disk, and then merge it with the tuple before writing back.

4. PERFORMANCE EVALUATION

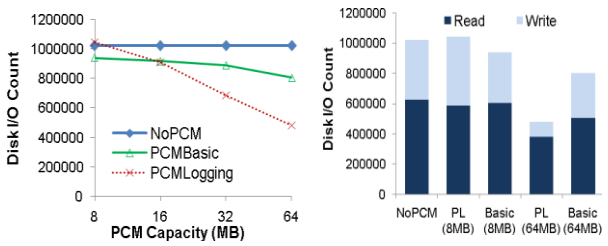
In this section, we report the preliminary results we obtained via simulation experiments. We have developed a trace-driven simulator based on DiskSim (http://www.pdl.cmu.edu/DiskSim/). We implemented a transaction processing model and a PCM model on top of a simulated disk. In the transaction processing model, we employ the strict two-phase locking protocol for concurrency control at a tuple level. Deadlocks are detected and resolved whenever a transaction is blocked. If a deadlock is discovered, the youngest transaction in the deadlock is restarted after a random backoff time. In the PCM model, the current implementation assumed only one memory chip [5]. The data access granularity in the PCM was fixed at 64B. We set its write latency at 1μs, and its read latency the same as that of DRAM access (*i.e.*, 50 ns).

The database trace on disk I/O accesses was obtained by running PostgreSQL 8.4 with the TPC-C benchmark for four hours. We set the client number at 50 and the number of data warehouses at 20. As the database size is 2.4GB, we fixed the size of DRAM main memory at 64MB (*i.e.*, ~2.6% of the database size). For simplicity, we assumed that each tuple has a size of 128B. We conducted our simulation study on a desktop computer running Windows XP SP2 with an Intel Quad 2.4GHz CPU. For a fair comparison, the results were collected after a fixed warmup period (*i.e.*, after the PCM becomes full under all schemes).

Fig. 4 shows the disk I/O performance when we vary the PCM size from 8MB to 64MB. We compare the PCMLogging scheme with tuple-based buffering (denoted as *PCMLogging*) to the basic scheme (denoted as *PCMBasic*) presented in the Introduction. For PCMBasic, to minimize the disk I/Os, checkpoint is triggered whenever the log pool becomes full; under each PCM size setting, we tried different space partitions between the buffer pool and the log pool and plotted the result with the best partition. As a reference, we also include a scheme without PCM support (denoted as *NoPCM*).

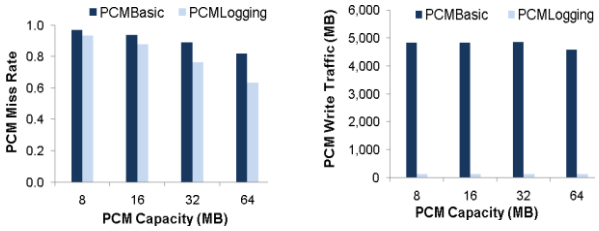
As shown in Fig. 4(a), both PCMBasic and PCMLogging achieve a better I/O performance than the NoPCM scheme, as expected. PCMLogging greatly improves the performance when the PCM size is larger than 16MB. At a PCM size of 64MB, it outperforms PCMBasic by 40%. However, PCMLogging is slightly worse than PCMBasic at a PCM size of 8MB. To explain this, we plot the I/O breakdown and PCM miss rate in Figs. 4(b) and 5(a), respectively. We can observe that when the PCM size is 8MB, the miss rate is high and hence not many read and write requests can be served by the buffered copies. Thus, the read I/Os are not much reduced, as compared with the NoPCM scheme (see Fig. 4(b)). On the other hand, due to a small size, PCM replacement happens frequently. Recall that PCMLogging incurs an extra read I/O (counted in write in Fig. 4(b)) when writing buffered tuples to the disk during PCM replacement. This makes its write performance even worse than NoPCM, thereby incurring a worse overall performance. When the PCM size increases, the miss rate is significantly reduced, especially for PCMLogging. For example, at 64MB, PCMLogging reduces the miss rate to 63% (vs. 81% for PCMBasic), as shown in Fig. 5(a). Moreover, PCMLogging has a larger buffering capacity with a tuple-based buffering granularity. As a result, PCMLogging gets much fewer disk I/Os for both reads and writes, as shown in Fig. 4(b).

Next, we investigate the write traffic to the PCM, which may affect its lifetime. As shown in Fig. 5(b), the write traffic of



(a) Disk I/O Count (b) I/O Breakdown

Fig. 4: Disk I/O Performance (PL=PCMLogging)



(a) Write Hit Rate on PCM (b) Write Trafic on PCM

Fig. 5: PCM Write Traffic

PCMLogging is greatly reduced; it is only 2.7% of PCMBasic on average. This is mainly because that the updates in OLTP workloads are mostly small writes, buffering only the dirty tuples in PCMLogging can avoid writing the clean part of a page. This result indicates that the lifetime of PCM can be significantly improved by PCMLogging.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a study on leveraging PCM to support efficient transaction processing. We have developed a new PCMLogging scheme that combines the buffered updates and log records, by taking advantage of the PCM hardware features. The preliminary results have been encouraging. It is shown that PCMLogging saves up to 40% disk I/Os and 97% PCM write traffic in comparison with the basic buffering and logging scheme.

This paper represents an initial step of our work towards improving database transaction performance with PCM. For future work, we will enhance the simulation experiments. We also plan to further improve our work in a number of directions, such as advanced PCM replacement policies, fine-grained tuple-based mapping, wear-leveling techniques, multi-version concurrency control, and integration with flash-memory technology.

Acknowledgement. This work is partially supported by GRF Grants HKBU210808 and HKBU211510.

6. REFERENCES

[1] S. Chen, P. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In CIDR, 2011.

[2] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In SOSP, 2009.

[3] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: A scalable approach to logging. VLDB, 2010.

[4] Samsung Press Release. Samsung ships industry's first multi-chip package with a PRAM chip for handsets, April 2010.

[5] Micron 128Mb P8P Parallel PCM Data Sheet, March 2011.

[6] E. Rahm. Performance evaluation of extended storage architectures for transaction processing. In SIGMOD, 1992.