Flag Commit: Supporting Efficient Transaction Recovery in Flash-Based DBMSs

Sai Tung On, Jianliang Xu, *Senior Member*, *IEEE*,
Byron Choi, *Member*, *IEEE*, Haibo Hu, and Bingsheng He

Abstract—Owing to recent advances in semiconductor technologies, flash disks have been a competitive alternative to traditional magnetic disks as external storage media. In this paper, we study how transaction recovery can be efficiently supported in database management systems (DBMSs) running on SLC flash disks. Inspired by the classical shadow-paging approach, we propose a new commit scheme, called `flagcommit`, to exploit the unique characteristics of flash disks such as fast random read access, out-place updating, and partial page programming. To minimize the need of writing log records, we embed the transaction status into flash pages through a chain of commit flags. Based on `flagcommit`, we develop two recovery protocols, namely commit-based flag commit (CFC) and abort-based flag commit (AFC), to meet different performance needs. They are flexible to support no-force buffer management and fine-grained concurrency control. Our performance evaluation based on the TPC-C benchmark shows that both CFC and AFC outperform the state-of-the-art recovery protocols.

Index Terms—Flash memory, database, transaction recovery.

1 Introduction

Flash disks (or simply flashes) have been becoming increasingly popular to serve as external storage media in portable devices, as well as new-generation laptops and enterprise servers, thanks to their fast random access, low power consumption, high shock resistance, small dimensions, and lightweight [20]. In particular, single-level-cell (SLC) flashes are known to be ideal for such enterprise applications as OLTP and databases that require the highest read/write performance and reliability [1], [7]. There has been a stream of recent research on flash-based DBMSs [9], [18], [25], [29], [32], [37]. In this paper, we contribute to flash-based DBMSs by investigating how transaction recovery can take advantage of the unique characteristics of SLC flashes to improve performance.[1]

Transaction recovery, as an inseparable part of DBMSs, enforces atomicity and durability of transactions, among others [21]. In brief, atomicity ensures that for each transaction, either all or none of its actions are performed, and durability guarantees that the data written by committed transactions persists even in the event of a system failure. There are two predominant approaches to support transaction recovery for DBMSs operating on magnetic disks, namely *write ahead logging* (WAL) [22], [31] and *shadow paging* [21], [35]. Both approaches have been implemented in real systems. In WAL, *in-place* updating is used. Updates to data pages can be performed only after they have been logged, which guarantees the ability to undo the updates during transaction rollback or system restart. Furthermore, the log records are forced to stable storage (e.g., disk) before a commit is completed, which ensures the ability to redo the updates during system restart. In addition to normal update logs, WAL writes some special log records, such as compensation and commit protocol-related records, to record significant events during transaction processing. In general, writes are frequent in WAL, especially with short transactions. Even worse, when there is a rollback, WAL may require many undo operations if the updated pages have been flushed to the disk. Therefore, WAL may not be preferable on flash disks, where write operations are more expensive than read operations.

In comparison, shadow paging handles updates by *out-place* updating. In shadow paging, data pages are accessed through a *page mapping table*, which maps page IDs to disk addresses. When a transaction wants to update a page, it allocates a *shadow page* elsewhere on the disk, performs the update on the shadow page, and then changes the *current* mapping of the page. Only when the transaction commits, the current mapping is flushed to the disk and becomes *persistent*. Otherwise, if the transaction is aborted, we can simply discard the shadow page and the current page mapping. As such, the total number of writes in shadow paging is often smaller than that in WAL.

Shadow paging has not been as popular as WAL for magnetic disk-based DBMSs because of several performance issues, which however no longer exist for flash disks. First,

1. There are two types of flash chips: single-level-cell and multilevel-cell (MLC) chips. SLC chips store one bit in each memory cell whereas MLC chips can store two or more bits in each cell. The two types of flash chips tend to target on different applications due to their performance differences in the access time and lifetime. In contrast to SLC flashes, MLC flashes are more suited for read-intensive applications such as web or video serving [1]. In this paper, we focus on SLC flashes for DBMS applications and hereafter refer to SLC flashes simply as flashes for brevity.

- *S.T. On, J. Xu, B. Choi, and H. Hu are with the Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, KLN, Hong Kong. E-mail: {ston, xujl, bchoi, haibo}@comp.hkbu.edu.hk.*
- *B. He is with the School of Computer Engineering, Nanyang Technological University, Nanyang Avenue, Singapore 639798. E-mail: bshe@ntu.edu.sg.*

Manuscript received 26 Feb. 2010; revised 19 May 2010; accepted 3 May 2011; published online 26 May 2011.
Recommended for acceptance by N. Bruno.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2010-02-0116.
Digital Object Identifier no. 10.1109/TKDE.2011.122.

maintaining the page mapping table is an indispensable function of flash disks, rather than an overhead. Second, the high commit overhead of flushing the *current* page mapping can be alleviated by embedding the mapping in the shadow page. Third, shadow pages may be scattered on a flash disk. Since flashes support fast random read access, this does not cause a performance problem. Last, shadow paging may result in obsolete pages which can be handled by garbage collection of flashes without additional work. These render shadow paging an appealing solution to supporting efficient transaction recovery on flash disks.

Recently, Prabhakaran et al. [34] have developed a shadow-paging-based scheme, called *cyclic commit*, for flash-based *file systems*. They establish a cyclic linked list among all shadow pages of a committed transaction. By determining the existence of such a list during recovery, the file system can determine whether a transaction has been committed or not. Two protocols were derived from this idea, i.e., simple cyclic commit (SCC) and back pointer cyclic commit (BPCC). However, as we investigate in this paper, adapting these two protocols to flash-based DBMSs suffers from several deficiencies. First, to form a cyclic linked list, both protocols require the current shadow page of a transaction to be buffered until the arrival of the next shadow page. Second, to maintain data consistency, SCC requires frequent erasures of uncommitted pages and BPCC requires a complicated garbage collection mechanism that leaves many obsolete pages irreclaimable (see Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2011.122, for more details). Moreover, these two protocols do not consider buffering and fine-grained concurrency control support, which are common concerns of DBMSs.

In this paper, we address these issues by proposing a new shadow-paging-based `flagcommit` scheme for efficient transaction recovery in flash-based DBMSs. It is well known that flashes have an *erase-before-write* limitation—in general, a page must be erased before it can be overwritten. Nevertheless, an often overlooked feature is *partial page programming*, which allows a flash page to be programmed/updated a few times before an erasure becomes mandatory, as long as the update is a change from bits "1" to bits "0" [15], [38].[2] In `flagcommit`, we exploit this partial page programming feature to keep track of the transaction status. Specifically, `flagcommit` stores a flag about the transaction status in each shadow page. Since flags can be updated in place once, transaction commit processing is accomplished by an update of some flag. This results in fewer erasure operations than SCC and BPCC as well as a simpler garbage collection mechanism compared to BPCC.

We develop two transaction recovery protocols based on the idea of `flagcommit`, namely *commit-based flag commit* (CFC) and *abort-based flag commit* (AFC). They differ in the semantics of flags, the times when flags are set or updated, and the actions of garbage collection. Consequently, they exhibit different performance behaviors and suit different workloads. On the whole, the contributions of this paper are summarized as follows:

2. Note that partial page programming is not supported in MLC flashes. We discuss the extension and challenge of applying `flagcommit` to MLC flashes in Section 7.

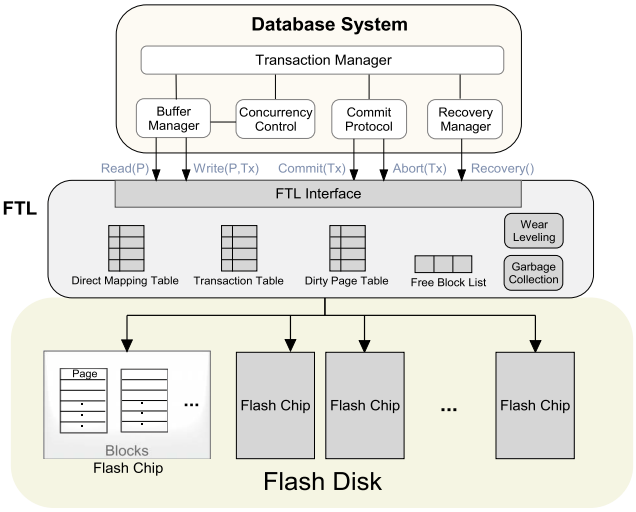

Fig. 1. Flash-based system architecture.

- We propose shadow-paging-based `flagcommit` that supports efficient transaction recovery in flash-based DBMSs. To the best of our knowledge, this is the first work on transaction recovery management for flash-based DBMSs.
- We develop two transaction recovery protocols based on `flagcommit`. For each protocol, the algorithms for normal transaction processing, garbage collection, and recovery are presented.
- We extend our protocols to general DBMS scenarios where no-force buffer management and fine-grained concurrency control are supported.
- We conduct an extensive performance evaluation of our proposed protocols. The results show that they outperform the existing shadow-paging-based protocols by up to 47 percent and outperform the WAL-based protocol by up to 83 percent in terms of the transaction throughput.

Organization. The rest of this paper is organized as follows: Section 2 introduces the background of our research, including the flash characteristics and the flash translation layer (FTL). Section 3 presents `flagcommit`, two flag commit protocols, and their implementation issues. Section 4 discusses how to extend the flag commit protocols to no-force buffer management and fine-grained concurrency control. We present the performance evaluation results of `flagcommit` in Section 5. In Section 6, we give a survey on flash-aware data management techniques. Finally, we conclude this paper and discuss future directions in Section 7.

2 Background

In this section, we describe the relevant characteristics of SLC flashes as well as the flash translation layer.

2.1 Flash Characteristics

Like magnetic disks, flashes are nonvolatile storage. As illustrated in the lower part of Fig. 1, a flash disk consists of a number of flash memory chips. A flash memory chip is

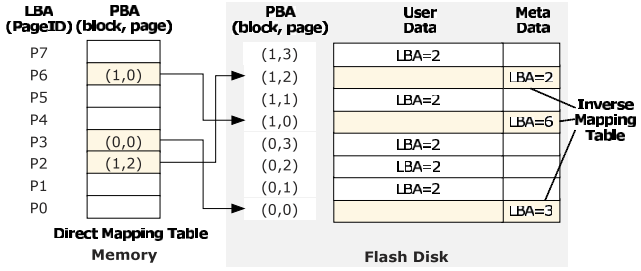

Fig. 2. FTL mapping table.

organized in many blocks, and each block is composed of a fixed number of pages. As of 2010, a typical page size is $2K + 64$ bytes, where the 2K bytes constitute the *data* area and the 64 bytes constitute the *spare* area. The spare area often stores metadata such as the error correction code (ECC) and logical block address (LBA). A typical block size is $128K + 4K$ bytes (i.e., 64 pages). Erasure operations must be performed at the block level, while read and write operations are performed at the page level.

Flashes possess a number of unique I/O characteristics [12], [19], [36]. First, without involving any mechanical components, the seek time is negligible in accessing a flash page. Hence, random access is efficiently supported. Second, flashes are subject to an *erase-before-write* constraint. Writing (or programming) a page involves clearing bits from "1" to "0"; SLC flashes allow partial page programming, that is, a flash page can be (re)-programmed a few times without performing an erasure operation [15], [38].[3] However, the only way to set bits (i.e., from "0" to "1") is to erase the whole block where the page resides. Third, each block can survive only a limited number of erasure operations (typically, 10,000-100,000 times). Thus, to address the erase-before-write constraint and achieve wear leveling, *out-place* updating is often supported through the use of a software layer called flash translation layer [23].

In addition, some flash memories enforce a sequential ordering for the writes within a single flash block (e.g., [6]), with the objective to reduce the effect of internal write disturbance. Since this is not a fundamental constraint of flash memories, this restriction is not considered when we present our proposed protocols in the next few sections. We shall discuss in detail the impact of this restriction on the design and performance of the proposed protocols in Appendix C which can be found online in the supplemental materials.

2.2 Flash Translation Layer

As illustrated in the middle part of Fig. 1, FTL, residing between the flash disk and high-level applications, provides an interface to read flash pages and support transactional operations (page writes, commit, abort, and recovery). The core of FTL is to support mapping between logical and physical page addresses. As shown in Fig. 2, a *direct mapping table* recording the mappings from LBAs to physical addresses is maintained in memory to speed up read access; an *inverse mapping table* is stored as metadata in the spare areas of flash pages, which is used to rebuild the direct mapping table at boot time. By maintaining the mappings, out-place updating can be readily implemented on FTL. Note that each out-place update leaves an obsolete page on the flash. FTL maintains a list of free blocks and has a *garbage collection* module to reclaim obsolete pages. Specifically, upon being triggered, this module selects a block (e.g., based on the number of obsolete pages), copies valid pages to some free block, erases the selected block, and puts it to the free block list. To lengthen the lifetime of flashes, FTL also supports *wear leveling* techniques to spread writes/erasures uniformly across the entire disk space.

3. In today's flash memory chips, partial page programming is usually supported at the level of a byte or a word [4], [5].

(a) An in-progress / aborted transaction (b) A committed transaction

Fig. 3. Chained commit flags in CFC.

3 Basic Flag Commit Protocols

Recall that transaction recovery guarantees the atomicity and durability properties of transactions in the face of system failures. For this purpose, we need to keep track of the updates made to the database as well as the transaction status, so that we can undo the updates of aborted/in-progress transactions and redo the updates of committed transactions during system recovery. To do so, shadow paging performs out-place updating and logs the transaction status (i.e., aborted or committed) during normal processing [21]. This section proposes a new `flagcommit` scheme based on the shadow-paging approach, inspired by the cyclic commit scheme [34].

The basic idea is to use shadow pages to keep track of the updates. Meanwhile, we encode the transaction status and update history in shadow pages to support undo/redo actions. In contrast to the prior cyclic commit scheme (detailed in Appendix A, which can be found online in the supplemental materials), `flagcommit` stores in each shadow page a link pointing to the *preceding* shadow page that belongs to the same transaction, then a chain of *commit flags* (or simply *flags*) is used to maintain the transaction status. Fig. 3 shows an example of `flagcommit`, where we have three shadow pages $P2$, $P3$, and $P6$ updated by a transaction. The *TRUE* or *FALSE* value in each page indicates the flag of the page. Specifically, `flagcommit` stores a flag in the spare area of each shadow page, which can be locally updated at least once (through partial page programming) without a block erasure. `flagcommit` determines whether a transaction is committed or not by checking the flag chain of the corresponding shadow pages. In addition, `flagcommit` stores in each shadow page a page version number and the ID of the transaction that updated the page. Combining these with the links of shadow pages, `flagcommit` can determine the update history of transactions and subsequently the redo/undo actions during recovery.

In DBMSs, a buffer pool is typically used to cache the frequently accessed disk pages of the database. The buffer management policy has an impact on the transaction recovery mechanism. If a *steal* policy is used, a page updated by a transaction is allowed to be flushed to the disk before the

transaction commits. This is not allowed in a *no-steal* policy. On the other hand, if a *force* policy is used, a transaction is not allowed to commit until all pages updated by it are flushed to the disk. In contrast, a *no-force* policy allows a transaction to commit at any time. To optimize system performance, steal and no-force policies are adopted by most DBMSs. However, steal implies that some undo work may have to be performed during normal or restart rollback, while no-force implies that some redo work may have to be performed during recovery. In this section, we present two basic `flagcommit` protocols, CFC and AFC, that work with steal and force polices, where logging is not needed at all. We also assume that a page-level concurrency control protocol is in place to handle update conflicts. We shall discuss how to extend them to support no-force buffer policy and record-level concurrency control in the next section.

For ease of exposition, in the rest of this section, we assume a *write-through* buffer: each shadow page, after updating, is immediately flushed to the flash disk. This is not a requirement of `flagcommit`. With a *write-back* buffer, the `flagcommit` protocols remain almost the same except that a shadow page is not created on the flash disk until the page is evicted from the buffer pool or the corresponding transaction commits.

3.1 Commit-Based Flag Commit

Commit-based flag protocol uses flags to indicate a *committed* transaction, as shown in Fig. 3. The commit flags in all shadow pages are initially set to *FALSE* during the execution of a transaction (Fig. 3a). When the transaction commits, the commit flag of the last shadow page (i.e., P6) is updated to *TRUE*, as shown in Fig. 3b.

Definition 3.1. *In CFC, a transaction is* committed *if and only if there is* at least one *of its shadow pages whose commit flag is* TRUE.

Definition 3.2. *In CFC, a page is* committed *if the transaction that updated the page has* committed.

For example, consider Fig. 3 again. In the case of Fig. 3a, no commit flag is *TRUE*. Hence, the transaction is not committed. In the case of Fig. 3b, the commit flag of *P6* is *TRUE*. Hence, the transaction is committed, and in this case, *P2*, *P3*, and *P6* are all committed pages.

In-memory tables. To realize CFC, several tables should be maintained in memory (shown in the left-hand side of Fig. 4):

- *Transaction table.* The transaction table maintains the status for each transaction (i.e., in progress, committed, or aborted). In the actual implementation, we only need to store the transaction status of in-progress and aborted transactions. A transaction missing from the transaction table implies the transaction is committed. For each in-progress and aborted transaction, it also keeps track of the number of shadow pages (*NPage*) and the last shadow page (*LastPage*) in runtime. The purpose of keeping *NPage* is as follows: once all shadow pages of a transaction are reclaimed, its corresponding entry can be removed from the transaction table. The benefit of maintaining *LastPage* is twofolded. First, an in-progress transaction can easily identify its last shadow page to set the link in the current page. Second, for a to-be-committed transaction, we can directly identify the latest shadow page to set the commit flag.
- *Dirty page table.* The dirty page table keeps track of the physical block address (*PBA*) of the last *physical* page (*LastPBA*) for each dirty page updated by an in-progress transaction. This is necessary for restoring the original data in the event of a rollback. A dirty page entry will be removed from the table once the corresponding transaction is committed or aborted.
- *Direct mapping table.* The direct mapping table maintains the up-to-date logical-to-physical address mapping for each page.

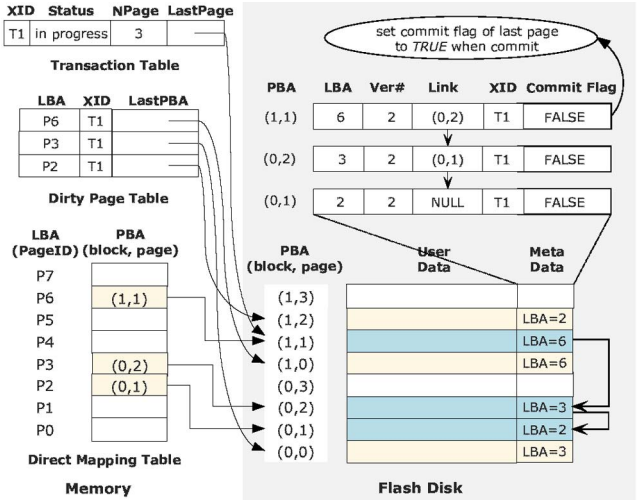

Fig. 4. CFC under normal execution.

Page format on flash disk. The right-hand side of Fig. 4 shows the format of CFC pages on the flash disk. As discussed in Section 2.1, a flash page consists of a data area and a spare area. In CFC, the contents of a spare area include the following fields:

1. *LBA* represents the logical page ID;
2. *Version #* is the number of times the logical page has been updated;
3. *XID* stores the ID of the transaction that caused the latest update of the page;
4. *Link* points to the preceding shadow page of the same transaction (*NULL* if it is the first page), by which all shadow pages of a transaction are chained together; and
5. *Commit Flag* is a bit value indicating whether the transaction is committed (*TRUE*) or not (*FALSE*).

In the actual implementation, we use bit "0" to represent *TRUE* and bit "1" to represent *FALSE*. Hence, *FALSE* can be *reprogrammed* to *TRUE* through partial programming.

Next, we detail the normal transaction processing, garbage collection, and recovery for the CFC protocol.

3.1.1 Normal Processing

To describe how CFC processes a transaction, we present the detailed steps of updating a page, transaction commit, and transaction abort:

- *Update.* When a transaction T updates a (logical) page P, CFC performs the following four steps:
  1. A shadow page P' is created on the flash disk. Fill in the *LBA*, *Version #*, *Link*, *XID*, and *Commit Flag* (with an initial value of *FALSE*) fields in the spare area of P'.
  2. In the transaction table, the *NPage* value of T is incremented by one, and the *LastPage* of T is updated to P'.
  3. P is added to the dirty page table.
  4. In the direct mapping table, P is mapped to P'.[4]
- *Commit.* When a transaction T commits, CFC identifies the *LastPage* of T from the transaction table and updates the *Commit Flag* of *LastPage* to *TRUE*. Then, T is removed from the transaction table (implying a "committed" status). Next, the updated pages of T are removed from the dirty page table, and their last physical pages are marked as obsolete.
- *Abort.* If a transaction T aborts, CFC will undo the updates caused by T. CFC marks the pages updated by T as obsolete. It then locates the last physical pages of the updated pages by checking the dirty page table, and restores them by updating the corresponding mapping entries in the direct mapping table.

Example 3.1. Consider a new transaction T1 which updates pages P2, P3, and P6, in sequence, over a database snapshot shown in Fig. 2.[5] In Fig. 4, when the first page P2 (originally stored in (1,2)) is updated, the following is performed:

1. P2 is written to a shadow page (0,1). On the shadow page (0,1), the metadata is filled in, where Version # is incremented over the last version # of P2, Link = NULL, and Commit Flag = FALSE.
2. A new entry is added to the transaction table to record NPage and LastPage of T1 (i.e., 1 and (0,1), respectively).
3. P2 is added to the dirty page table to record its LastPBA (1,2).
4. The direct mapping of $P2$ is updated to (0,1) accordingly.

Next, when P3 (originally stored in (0,0)) is updated, a shadow page (0,2) is created. Its Link is set to (0,1). Other metadata of (0,2) is also filled in accordingly. In the transaction table, NPage of T1 becomes 2 and LastPage of T1 becomes (0,2). P3 is added to the dirty page table. Afterwards, when P6 (originally stored in (1,0)) is updated to the shadow page (1,1), its Link is set to (0,2), NPage and LastPage of T1 in the transaction table become 3 and (1,1), and P6 is added to the dirty page table. The above steps give us the data structures shown in Fig. 4.

Finally, if T1 commits, we locate its LastPage (1,1) and update its Commit Flag to TRUE. In addition, T1 is removed from the transaction table, and P2, P3, and P6 are removed from the dirty page table. Otherwise, if T1 eventually aborts, no action is performed on the flash disk, but the last physical addresses of P2, P3, and P6 will be restored (so as to "undo" the changes) in the direct mapping table.

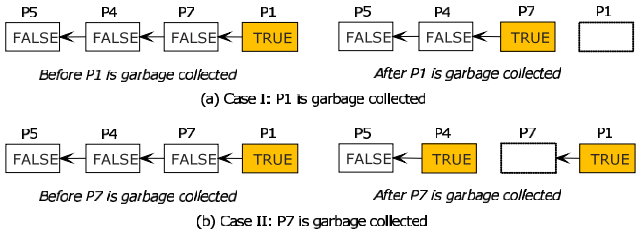

Fig. 5. CFC garbage collection.

3.1.2 Garbage Collection

When the amount of free space on the flash disk becomes lower than some preset threshold, garbage collection is triggered to reclaim the space occupied by obsolete pages. In the CFC protocol, a page can be reclaimed if: 1) the page is uncommitted; or 2) the page is committed but already out-of-date (i.e., there exists another newer *committed* version of the page). Obviously, a committed page with only newer *uncommitted* versions cannot be reclaimed. Note that after an uncommitted page is reclaimed, the *NPage* value of its corresponding transaction is decremented by one in the transaction table.

Reclaiming uncommitted pages is trivial. Here, we focus on the actions required by reclaiming an out-of-date committed page P. There are two possible cases:

- *Case 1. P's Commit Flag is TRUE.* As P holds the commit flag, we should move this flag to P's preceding page (via *Link*) before reclaiming P, in order to ensure that one commit flag still exists for the committed transaction (Definition 3.1). This is done by updating the commit flag of the preceding page. For instance, Fig. 5a shows that the commit flag of $P1$ is moved to $P7$ when $P1$ is reclaimed.
- *Case 2. P's Commit Flag is FALSE.* In this case, P may be in the middle of the shadow-page chain. Once P is reclaimed, the chain will be split into two subchains. Thus, we maintain a commit flag in each subchain, as illustrated in Fig. 5b. These two subchains would then be treated like two transactions. For example, when $P7$ is reclaimed, a commit flag is added to $P4$.[6] By doing so, the recovery procedure (to be discussed shortly) is simplified. Moreover, the transactions trivially conform to Definition 3.1—there is at least one commit flag for each committed transaction. In Fig. 5b, suppose we do not set the commit flag for $P4$ before erasing $P7$, then if $P1$ is reclaimed later, there would be no commit flag for this transaction. Enforcing a commit flag for each subchain can address this issue.

It is worth noting that, unlike the cyclic commit scheme [34] which uses *logical* addresses as *next-link* pointers, *physical* addresses are used in our protocol to maintain the

4. We here assume an *immediate-update* (cf. *deferred-update*) strategy, as implemented in most DBMSs [11].

5. For a better illustration, an animated demo for this example is available at: http://www.comp.hkbu.edu.hk/~db/flash/cfc.ppt.

6. In the implementation, the commit flag should be added to $P4$ before $P7$ is to be reclaimed. Otherwise, if the system fails right after $P7$ is reclaimed, we cannot add the commit flag to $P4$.

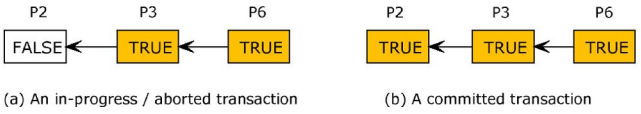

Fig. 6. Chained commit flags in AFC.

chain structure. The benefit is that we no longer need to keep page version numbers in the direct mapping table, which saves memory space. However, using physical addresses as *Link* pointers brings a new minor issue. The chain may be broken due to a reallocation of some physical pages (during garbage collection, all valid pages in a block will be relocated before the block is reclaimed). To handle this issue, we require a slight modification of garbage collection—when a page is relocated, its commit flag must be set to *TRUE* if the corresponding transaction has been committed. Thus, even though the original chain is broken into more than one subchain, there is at least one *TRUE* flag for each subchain of a committed transaction.

3.1.3 Recovery

After a normal shutdown or system failure, a recovery procedure is invoked when the system restarts. It recovers the last committed version for each page and rebuilds the direct mapping table.

The recovery procedure works by scanning the physical flash pages as follows: Let S denote the set of nonempty pages and R denote the set of pages that are backward pointed by some page in S. Both S and R can be obtained by a scan of the spare areas of all physical pages.[7] Obviously, $S - R$, denoted by U, is the set of head pages for the commit-flag chains (e.g., $P4$ and $P1$ in the right part of Fig. 5b). Whether a transaction is committed or not can be judged by the commit flag of the corresponding head page in U. Thus, we divide U into two sets: U^+ for the pages with *TRUE* flags and U^- for the pages with *FALSE* flags. By recursively following the link in each page of U^+, we obtain the whole set of committed pages, S^+, which is formally defined as follows:

$$S^+ = U^+ \cup \{p \in S \mid \exists u \in U^+, \text{ there is a path from } u \text{ to } p\}.$$

After the set of committed pages S^+ is obtained, the recovery procedure identifies the committed page with the largest version number for each logical page, and builds the direct mapping table accordingly. The time complexity of the CFC recovery is $O(n)$ (n is the number of flash pages), because each flash page is visited at most twice.

There is a subtle issue for discussion. Recall that due to the use of physical *Link* pointers, a shadow-page chain could be broken into more than one subchain during garbage collection (their head pages are traced by the transaction table). If this happens before a transaction commits, then at the time of transaction commit, we should set the commit flag to *TRUE* for the head pages of all the subchains. Once this is successfully done, each subchain of the transaction has one *TRUE* flag, consistently indicating the transaction is committed. However, if the system fails during this process of setting commit flags, it may leave the transaction status

7. In order to reduce the recovery time, one alternative is to duplicate the metadata of the pages within a block into the last free page (called per-block summary page), when the block is about to be filled up [34]. In this way, we only need to read the summery pages for those filled blocks.

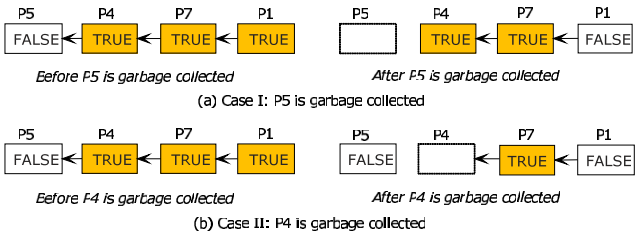

Fig. 7. AFC garbage collection.

inconsistent: some subchains of the transaction are committed while some others are not. Since such cases can be observed only after a system failure, the correctness of our protocol preserves for normal system running. Yet, the recovery procedure triggered by a system failure should handle such cases. We consider a transaction with inconsistent status as aborted. After U^+ and U^- are obtained, the recovery procedure takes an extra step: if a head page in U^+, e.g., P, shares the same transaction ID with some head page(s) in U^-, P will be removed from U^+.

3.2 Abort-Based Flag Commit

During garbage collection of CFC, Fig. 5 shows that commit flags may need to be moved or added for *committed* transactions. This would incur significant overhead if most transactions are committed. In this section, we present an alternative abort-based flag commit protocol that updates the flags of *aborted* transactions during garbage collection. This might be preferable since more transactions are committed than aborted in practice.

Definition 3.3. *In AFC, a transaction is* committed *if and only if no commit flag of its shadow pages is set to* FALSE.

AFC adopts almost the same page format as CFC except that in AFC, there are two bits in a *Commit Flag* (to be explained later). Different from CFC, the commit flags of all shadow pages *except* the first page (e.g., $P2$ in Fig. 6a) are initially set to *TRUE*. Note that we have to initialize the commit flag of the first page to *FALSE*, because otherwise if the system fails right after it is written to the flash disk, the transaction would be judged as committed when the system restarts. When the transaction commits, the commit flag of the first page is changed to *TRUE*. As such, a transaction is considered *committed* if no *FALSE* flag is found in the flag chain (Definition 3.3). Thus, even if the chain may be later split into several subchains, there is no *FALSE* flag in each subchain. As a result, no extra maintenance is needed for committed transactions during garbage collection. On the other hand, AFC needs to take extra actions to maintain the *FALSE* flags for aborted transactions. The process is similar to that of CFC except that we now maintain the *FALSE* flag instead of the *TRUE* flag. AFC has a special requirement that keeps track of the last shadow page for an aborted transaction before the first page is reclaimed. This is to ensure that AFC can quickly find an appropriate page to hold the *FALSE* flag during garbage collection.

Example 3.2. An example of AFC is shown in Fig. 7a, where the FALSE flag is moved to $P1$ after $P5$ is reclaimed. Another example is shown in Fig. 7b, where a FALSE flag is added to $P1$ when $P4$ is reclaimed.

TABLE 1
Comparison of Shadow-Paging Protocols

	CFC	AFC	SCC	BPCC
Protocol complexity	Simple	Simple	Simple	Complex
Recovery overhead	Low	High	Low	Low
Garb. collect cost	Low	Low	High	Moderate
Buffer requirement	None	None	1 page/tx	1 page/tx
Favored abort ratio	High	Low	Low	Low
Partial program support	Needed	Needed	No	No
Commit flag bits	1	2	-	-

The AFC recovery protocol differs from the CFC protocol in how they identify committed pages in U. While CFC identifies committed pages by the presence of a *TRUE* flag, AFC does by the absence of a *FALSE* flag. However, CFC would have a better recovery performance than AFC. During the recovery, for each chain of a committed transaction, CFC needs to traverse the chain if and only if the head page has a *TRUE* commit flag. In contrast, to identify committed pages, AFC requires to examine every flag in the chain to check the nonexistence of a *FALSE* flag, thereby incurring more page reads.

As mentioned, we define *Commit Flag* of AFC to be a two-bit value. In AFC, an initial *TRUE* value may need to be changed to *FALSE*, and vice versa. As such, in flash-based implementation, we use bits "10" and "11" to represent initial *TRUE* and *FALSE* values, respectively. They can be changed to "00" (also representing *FALSE*) and "10" (representing *TRUE*) through partial programming.

3.3 A Discussion of CFC and AFC

CFC and AFC may differ in their running performance. In this section, we give a brief analysis of these two protocols. To simplify the analysis, we ignore the effect of buffering, that is, every page read/write request incurs a page access on the disk. Let δ denote the transaction abort ratio, α denote the average number of page reads in a transaction, β denote the average number of page writes in a transaction, and γ denote the average number of blocks reclaimed by garbage collection when allocating a new flash page. We use k to represent the average number of nonempty pages in a block, and C_r, C_w, C_{bit}, and C_e to represent the costs of reading a page, writing a page, reprogramming a page, and reclaiming a block, respectively.

The average I/O cost for the CFC commit protocol is

$$C = \alpha C_r + \beta C_w + (1-\delta)C_{bit} + \gamma\beta k(1-\delta)C_{bit} + \gamma\beta C_e,$$

where the first three terms are the costs of reading involved pages, outputting shadow pages, and setting commit flags, the fourth item is the cost of extra actions taken during garbage collection, and the last item is the cost of reclaiming blocks.

Similarly, the average I/O cost for the AFC protocol is

$$C' = \alpha C_r + \beta C_w + (1-\delta)C_{bit} + \gamma\beta k\delta C_{bit} + \gamma\beta C_e.$$

By comparing C and C', CFC and AFC differ only in the cost of garbage collection, which is $\gamma\beta k(1-\delta)C_{bit}$ versus $\gamma\beta k\delta C_{bit}$. As such, AFC is expected to outperform CFC for normal cases when the abort ratio is less than 50 percent. On the other hand, CFC is preferred to AFC when the abort ratio is high or when the transaction recovery time is concerned (as discussed in the last section).

In addition, they also differ in some other aspects. First, AFC requires two flag bits while CFC only needs one bit, which results in different space usages. Second, in CFC, when a transaction commits, if the last shadow page of a transaction is buffered, the commit flag can be updated in main memory and then flushed to the disk. Hence, a page-reprogramming operation is saved. On the other hand, in AFC, when a transaction commits, we must first set the flag of the first shadow page to *FALSE*, and then change this flag to *TRUE after* all shadow pages are flushed to the disk. Thus, a page-reprogramming operation is mandatory to record the commit status. Table 1 summarizes the differences of these two protocols as well as the prior SCC and BPCC protocols [34]. We shall compare their performance through experiments in Section 5.

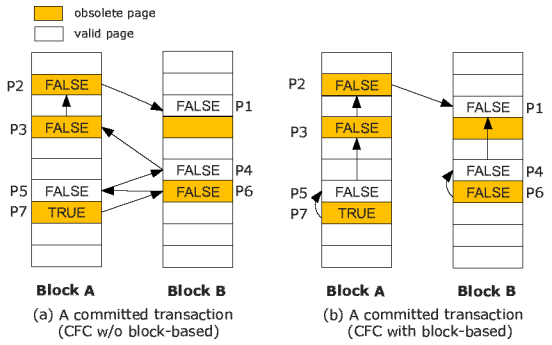

Fig. 8. Chained commit flags in CFC w/o and w/ block-based technique.

3.4 Block-Based Flag Technique

In the proposed CFC/AFC protocols, page reprogramming is needed to update the commit flag of the preceding shadow page when reclaiming a committed/uncommitted page in garbage collection. However, there are some special cases where such page-reprogramming operations can be saved. For example, if the preceding shadow page happens to reside in the same block to be reclaimed, we can update the commit flag for free when it is moved into the new block. This motivates us to propose a *block-based flag technique*. In the interest of space, the following discussion focuses on the CFC protocol as the case for the AFC protocol is similar.

We define a cluster as a group of linked shadow pages within the same block. The basic idea is to organize the shadow pages of a transaction by a chained list of clusters (cf. a chained list of individual pages in the original design). Fig. 8b shows a committed transaction with the block-based flag technique. The shadow pages $P1 \sim P7$ are divided into two clusters (i.e., $P2$, $P3$, $P5$, and $P7$ form a cluster, while $P1$, $P4$, and $P6$ form another cluster), and these two clusters are linked via $P2.Link$. In the following, we use an example to illustrate how the block-based flag technique would help to save page-reprogramming operations.

Example 3.3. Fig. 9 shows the block contents after $BlockA$ in Fig. 8 is reclaimed. Without the block-based flag technique, once $BlockA$ is reclaimed, to preserve the CFC protocol, two page-reprogramming operations are required to update the commit flags of $P1$ and $P6$ (see Fig. 9a). However, with the block-based flag technique, we just need to update the commit flag and Link of the valid page $P5$ and then move it to the new block (see Fig. 9b). In this way, no page-reprogramming operation is needed.

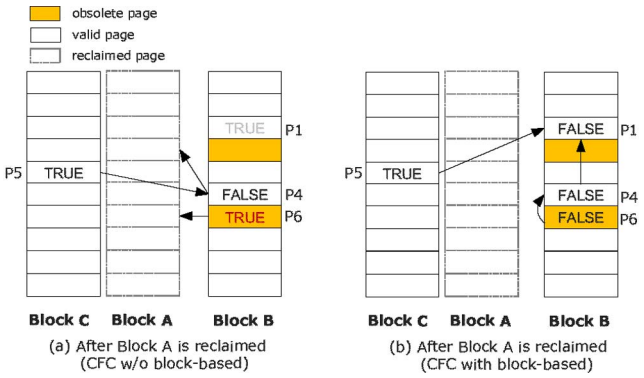

Fig. 9. Garbage collection in CFC w/o and w/ block-based technique.

Note that the block-based flag technique is not implemented without cost. It might impair the effect of buffering of CFC (discussed at the end of Section 3.3): if the last shadow page is not placed on a head cluster (i.e., a cluster not pointed by any other clusters), then at the time of transaction commit, a page-reprogramming operation is still needed even if this page is buffered. For example, in Fig. 8b, if the last page $P7$ is placed on $BlockB$ instead of $BlockA$, when committing, a page-reprogramming operation would be required to set a $TRUE$ flag on $P5$ regardless whether $P7$ is buffered. Nevertheless, we expect such overhead will not outweigh the benefit gained.

The recovery procedure is similar to that of the original CFC protocol except that we now perform the recovery on the granularity of clusters instead of individual shadow pages. Specifically, for each head cluster, if there is any shadow page holding a $TRUE$ flag, then by starting to traverse the chain from such a cluster, we can identify all committed pages.

4 Advanced `flagcommit` Protocols

In this section, we discuss how to extend the basic `flagcommit` protocols to support *no-force* buffer management and record-level concurrency control, which are commonly adopted in today's DBMSs.

4.1 Supporting No-Force Buffer Management

With a *no-force* buffer policy, shadow pages are not necessarily forced to the stable storage when a transaction commits. This improves the transaction response time and system throughput. However, in this case, while using shadow pages solely can still support undo work of aborted and in-progress transactions, it is not able to redo committed transactions in the event of failures [21]. Thus, following [21], we combine the basic `flagcommit` protocols with a redo log. In the following, we present a redo logging scheme adapted from the well-known recovery protocols [31], [35]. Yet the `flagcommit` protocols are not restricted to this; they may incorporate other logging schemes. The extended `flagcommit` protocols work as follows:

- Before updates to data are performed, *redo* log records are created and buffered (in a *log buffer*). The redo log record has the following format: $\langle XID, PID, RID, \ Opcode, Data, PrevLN \rangle$,[8] where XID is the transaction ID, PID is the page number, and RID is the record ID in the same page, $Opcode$ represents the operation type (insert, delete, or update), $Data$ stores the detailed operation information (e.g., the update content), and $PrevLN$ points to the previous log record generated by the same transaction. In the implementation, the log buffer can be indexed using hashing to facilitate the search on XID.
- When a shadow page is swapped out from main memory, we follow `cfc/afc` protocols to write it to the flash disk (including writing the metadata area and updating the mapping table etc.), and remove its corresponding log records from the log buffer since the update information can now be captured by the disk page.
- When a transaction commits, if some pages updated by it are still cached in main memory, we append a *commit* log record to the log buffer and force all buffered log records to the flash disk. Otherwise, the *commit* log record is not needed. In both cases, we then apply `cfc/afc` protocols to enforce a transaction commit for those on-disk pages.
- When a transaction aborts, we rollback its updated pages that are still in main memory and remove all its log records from the log buffer. For those pages updated only by this transaction since they were read from the disk, they can be simply discarded to save rollback overhead. The shadow pages that have already been flushed to the flash disk do not indicate a transaction commit, according to `cfc/afc` protocols. Hence, no extra action is needed to handle those pages.

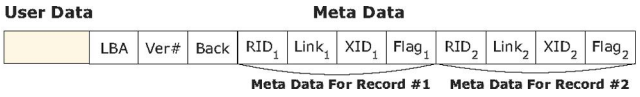

Fig. 10. Extended page format.

4.2 Supporting Record-Level Concurrency Control

Record-level concurrency control allows multiple transactions to update a single page at the same time. To achieve this, we associate a set of metadata attributes with each *updated record*, instead of each shadow page. More specifically, each transaction writes the new value of a record to the flash page as a *pending record*, which is associated with the metadata in the form of $\langle RID, Link, XID, CommitFlag \rangle$, where RID identifies the pending record in the page, $Link$ points to the page where the preceding pending record of transaction XID resides, and $Commit\ Flag$ indicates the transaction status. Fig. 10 illustrates an extended page format that holds two pending records, followed by a record-level CFC example shown in Fig. 11. Let P_i be the ith version of page P. In this example, the transaction $T1$ updating r_1, r_2, and r_5 and the transaction $T2$ updating r_3 and r_4 are allowed to update pages Q_1 and R_1 concurrently. The status of each transaction is encoded by its corresponding flag chain, which implies that $T1$ is committed and $T2$ is not (yet).

8. A physiological logging scheme [28] is adopted. Each log record keeps track of the update on a single page. For an update that involves multiple pages, it is decomposed into multiple miniupdates such that each miniupdate is confined to a single page.

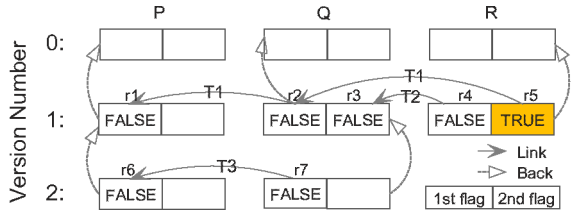

Fig. 11. An example of record-level CFC protocol.

4.3 Putting All Together

To work with record-level concurrency control, the extended `flagcommit` protocols in Section 4.1 are slightly changed for normal processing. The operations are now on the granularity of a pending record (and its associated metadata), rather than on a shadow page. Take Fig. 11 as an example. When the transaction $T3$ updating r_6 and r_7 is committed, the commit flag of r_7 on page Q_2 is updated to *TRUE*. Suppose that Q_2 is already flushed to the disk but P_2 is still cached in main memory, a commit log record will be appended to the log buffer and all log records preceding it (including the redo log record for r_6) will be forced to the disk. Meanwhile, the metadata of r_6 can be safely removed from the buffered copy. Otherwise, if both P_2 and Q_2 are already flushed to the disk, there is no need to write any log record. In addition, the abort logic is slightly different. For example, when the transaction $T2$ updating r_3 and r_4 is aborted, the pages holding these pending records will be discarded from main memory unless there exists a committed record (e.g., page Q_1 holding a committed record r_2). In this case, r_3 is marked as invalid and can be restored later by reading the current content from the disk. Algorithms 1 and 2 summarize the procedures of processing a record update and transaction commit/abort under record-level concurrency control and no-force buffer policy.

Algorithm 1. Updating a Record in Extended `flagcommit`
Procedure: Update (Transaction T, Record r)
1: Let P be the page containing r;
2: **if** (concurrency control allows T to update r) **and** (# updates on $p \leq$ *max_c*) **then**
3: **if** P is not found in the buffer pool **then**
4: **if** the buffer pool is full **then**
5: Select a victim page V and evict it from the pool;
6: **if** V is a dirty page **then**
7: Follow basic `flagcommit` to write V to the disk;
8: **if** any log record of V is still cached **then**
9: Remove the log record from the log buffer;
10: Read P from the flash disk and put it into the buffer pool;
11: Create a log record for this update in the log buffer;
12: Update r on P;
13: **if** the log buffer is full **then**
14: Force all log records in the log buffer to the flash disk;
15: **else**
16: Block this update until other transactions release the resources;

Algorithm 2. Commit/Abort Logic in Extended `flagcommit`
Procedure: Commit (Transaction T)
1: **if** some of T's pending records are still in the buffer pool **then**
2: Append a commit log record to the log buffer;
3: Force all log records in the log buffer to the flash disk;
4: Apply basic `flagcommit` to record the commit status in the on-disk pages of T, and mark obsolete pages for garbage collection
Procedure: Abort (Transaction T)
1: **for** each pending record r of T in the buffer pool **do**
2: Let P be the page containing r;
3: **if** $\exists\, r' \in P$, r' is committed but not yet written to the disk **then**
4: Mark r as invalid; // r will be restored on access later
5: **else**
6: Discard P from the buffer pool;
7: Remove T's corresponding log records from the log buffer, and mark obsolete pages for garbage collection;

We remark that there is a limit on the number of concurrent transactions allowed to update a page. Since every transaction may need to write metadata and change the commit flags in the flash page, the number of concurrent transactions should not exceed *max_c*, i.e., the maximum number of partial programming operations or the maximum number of pending records allowed in a flash page, whichever is lower. The effect of *max_c* will be examined in Section 5.3.1.

Over time, the *committed records* of a logical page may scatter over multiple physical pages of different version numbers. For example, the committed records of the logical page R in Fig. 11 are distributed on R_0 and R_1. To save disk space, the latest page content can be obtained by merging those versions. To facilitate this merging process, we add an additional backward link *Back* to the spare area of a flash page. Before writing a new version P_v of a logical page P, a pointer pointing to the last version of P will be added (i.e., $P_v.Back = P_{v-1}$).

Garbage collection. A logical page P may be related to a set of physical pages $\{P_1, \ldots, P_n\}$, each with a different version number. Under the extended protocol, a page P_v can be reclaimed when one of the following two conditions is satisfied: 1) all pending records of P_v are aborted; or 2) for any committed pending record r in P_v, there exists a page P_i, where the version number of P_i is greater than that of P_v and r is already committed in P_i (i.e., the page content of P_v is totally out-of-date). For example, P_0 in Fig. 11 can be reclaimed, as for the committed record of P_0, an updated/duplicated committed version can always be found in P_1. Similar to the basic CFC/AFC protocol, before P_v is reclaimed, the garbage collection process might need to move the commit flags to the preceding pending pages (see Sections 3.1 and 3.2).

Recovery. Although the committed records of a logical page may scatter over multiple flash pages, the recovery procedure only needs to identify the highest version of the

page which has at least one committed record. In particular, the committed records can be determined by scanning the redo log and the *Version #*, *Link*, and *Commit Flag* information stored in flash pages. Once we have identified the pages with such records, the direct mapping table is built by following the basic CFC/AFC recovery protocol. Finally, we perform the redo actions according to the redo log.

To speed up the recovery procedure during system restart, the system periodically performs fuzzy checkpointing [11]. It involves the following three steps: 1) the buffer pool is scanned to get a list of dirty pages; 2) instead of forcing these dirty pages to the flash disk, their page IDs are simply noted, and they are subsequently written to the disk (in the background) during normal processing; 3) a special log record $\langle checkpoint \rangle$ is written to indicate the end of the checkpointing. The next checkpoint is not taken until all dirty pages noted at the previous checkpoint have been written. Note that processing new transactions is allowed when the checkpointing is in progress. After the checkpointing, the log records before the second-to-last $\langle checkpoint \rangle$ can be discarded, because all updates recorded by these logs must have been written to the flash disk. Therefore, during recovery, only the log records that belong to a committed transaction and appear after the second-to-last checkpoint need to be redone. Algorithm 3 summarizes the recovery procedure.

Algorithm 3. Recovery Logic in Extended `flagcommit`
Procedure: Recovery ()

1: Scan the redo log and apply basic `flagcommit` to identify the set of committed records S;
2: **for** each committed record $r \in S$ **do**
3: P be the page containing r and P' the current mapped page in the direct mapping table;
4: **if** $P' = \text{NULL}$ **or** P'.Version $< P$.Version **then**
5: $P' = P$;
6: Perform redo actions according to the log records that appear after the second-to-last checkpoint;

5 Performance Evaluation

In this section, we evaluate the performance of our proposed `flagcommit` protocols (CFC and AFC) based on the TPC-C benchmark [8]. We first describe the experiment setup and then compare the CFC and AFC protocols to the cyclic commit protocols and the WAL-based commit protocol.

5.1 Experiment Setup

As with [34], we employ the trace-driven SSD simulator, which is a modified version of the DiskSim [10], [13], for our performance study. We configured the simulator to emulate a 32 GB SSD with eight fully connected 4 GB flash packages and used the I/O settings specified in the data sheet [6]. For garbage collection, the simulator has implemented a wear-aware policy [10]. Similar to [34], 10 percent of the flash blocks were reserved for handling garbage collection, and the threshold to trigger the garbage collection was set to 5 percent.

We implemented the basic CFC and AFC protocols with block-based flag technique and their extensions in the simulator. For performance comparison, we also implemented the cyclic commit protocols SCC and BPCC [34], and the traditional WAL-based commit protocol [31]. For each protocol, we employed the Strict Two-Phase Locking (2PL) protocol to detect and resolve access conflicts: a wait-for graph of transactions is maintained, and deadlock detection is performed each time a transaction is blocked. If a deadlock is discovered, the youngest transaction in the deadlock is chosen as the victim and restarted after a random backoff time. The system has an LRU-based buffer pool to cache previously accessed disk pages. We remark that SCC and BPCC protocols require the last shadow page of each in-progress transaction to be resident in the buffer pool. The performance was evaluated based on the TPC-C benchmark, which represents an online transaction processing workload. To get the workload trace, we executed TPC-C transactions (generated by DBT-2 [2]) on PostgreSQL 8.4 [3] and recorded their data access requests. In the generator, we set the client number to 50 and the number of data warehouses to 20. We configure the buffer pool size to be smaller than the database size in order to exercise disk I/O operations. As each TPC-C transaction accesses less than 50 pages on average, the default buffer pool size (512 pages) is able to hold about 20 percent of our TPC-C working set (i.e., 50 concurrent transactions $\times 50$ pages/transaction $= 2{,}500$ pages). For a fair comparison, the buffer pool excludes the memory space needed for holding the associated data structures (e.g., the direct mapping table) for each protocol. For other system parameters, the logical page size was set to 8 KB; the log record size was set to 50 bytes; the transaction abort ratio was set to 5 percent by default; and the cost of partial programming was set to be the same as that of a page write. We summarize the default parameter settings in Table 2.

TABLE 2
Default Parameter Settings

Parameter	Default Setting
Erasure block size	128KB
Physical page size	2KB
Page write/read latency	0.2ms/0.08ms
Partial programming latency	0.2ms
Partial programming level	2
Block erasure delay	1.5ms
Logical page size	8KB
Log record size	50 bytes
TPC-C warehouse/database size	20/2GB
TPC-C client number	50
Buffer pool size	512 pages
Transaction abort ratio δ	5%

We conducted our simulation study on a desktop computer running Windows XP SP2 with an Intel Quad 2.4 GHz CPU. We measured the transaction throughput (i.e., the average number of committed transactions per second), transaction execution time (i.e., the average elapsed time between transaction start and end), commit response time (the average latency between the time the commit command is issued and the time the transaction is completed), recovery cost, and garbage collection overhead. In the measurement, we ignored the CPU cost since the I/O cost of accessing flash pages dominates the overall performance. For each simulation run, a 30-minute TPC-C trace was used to warm up the SSD simulator and the buffer pool, then a 4-hour TPC-C trace was used for the evaluation. This arrangement ensures that the measurement reflects the long-term, stable performance of the simulated system.

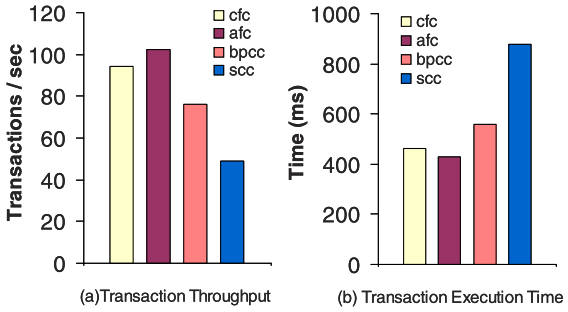

Fig. 12. Transaction throughput and execution time.

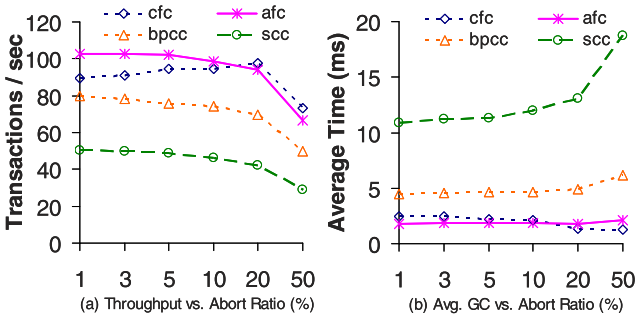

Fig. 14. Impact of transaction abort ratio δ.

5.2 Comparison with Cyclic Commit Protocols

In this section, we compare the proposed CFC and AFC protocols with the prior SCC and BPCC protocols. For a fair comparison, we configure the system with the same settings as SCC and BPCC presumed (i.e., force buffer management and page-level concurrency control).

As shown in Fig. 12a, AFC outperforms BPCC and SCC by 35 and 110 percent, respectively, in terms of the transaction throughput, while CFC achieves a slightly smaller improvement, i.e., 24 and 94 percent. Meanwhile, a similar performance trend is observed in Fig. 12b in terms of the transaction execution time. To gain more insight, we further measure their garbage collection overhead and plot the results in Fig. 13a. We also plot the ratio of garbage collection cost to the total execution cost for each protocol in the figure. As shown, AFC requires 13, 34, and 58 percent less time for garbage collection than CFC, BPCC, and SCC, respectively. This can be explained as follows: SCC has to explicitly reclaim uncommitted pages, which causes additional garbage collection activities and, hence, severely deteriorates its performance. Similarly, BPCC has to keep some obsolete pages, which incurs additional cost to move obsolete pages to new blocks during garbage collection. Worse still, as fewer pages can be reclaimed, garbage collection is triggered more frequently. From Fig. 13a, we can observe that the ratio of garbage collection cost to the total execution cost is over 35 and 55 percent for BPCC and SCC, respectively. On the other hand, AFC only maintains a flag for each aborted transaction and incurs little overhead at a low abort ratio (5 percent in the default setting). CFC maintains a commit flag for each committed transaction, which is a bit more costly when the abort ratio is low.

Fig. 13b shows the recovery performance results. For each protocol, we plot its time to recover the last committed version for each logical page during system restart. From the figure, we can see that AFC has a much longer recovery

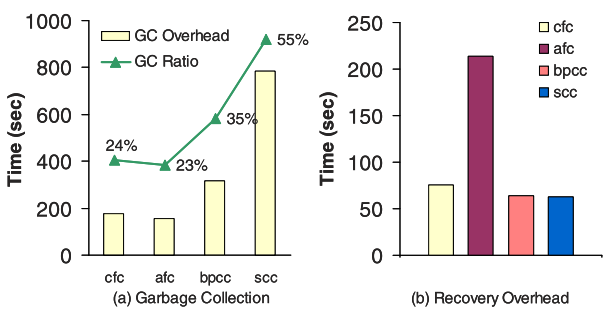

Fig. 13. Garbage collection overhead and recovery cost.

time than CFC, BPCC, and SCC. The main reason is that AFC needs many read accesses to traverse through each flag chain and decide the status of a shadow page. In contrast, CFC traverses a chain only if the head page/cluster holds a *TRUE* commit flag, thereby achieving comparable recovery performance to BPCC and SCC.

Next, we evaluate the four protocols with different transaction abort ratios (δ). Fig. 14a shows the transaction throughput results when δ is varied from 1 to 50 percent (by simulating the commit/rollback commands in the traces). We also plot the average garbage collection overhead per committed transaction in Fig. 14b. From these results, we can make the following observations. First, the transaction throughput decreases with δ for all protocols except CFC. This is because, when δ increases, more transactions are aborted, and hence more I/Os are wasted. As a result, the overall transaction throughput decreases in general. On the other hand, with a higher δ, CFC would save more partial programming operations. Therefore, its throughput is slightly increased when δ changes from 1 to 20 percent. Second, the improvement of the flag commit protocols over the cyclic commit protocols becomes more significant as δ increases. In particular, the performance gap between CFC and BPCC (SCC) increases from 12 to 47 percent (from 77 to 153 percent) when we increase δ from 1 to 50 percent. The reason is as follows: With a high δ, more uncommitted pages exist on the flash disk. As a result, for BPCC and SCC protocols, the average garbage collection overhead increases rapidly (see Fig. 14b)—when δ is increased from 1 to 50 percent, the garbage collection overhead of BPCC is increased by 38 percent, while that of SCC is even increased by 72 percent. Third, the performance gap between AFC and CFC decreases when δ becomes higher, and CFC outperforms AFC when δ is greater than 15 percent. This crossing point is smaller than the one predicted by our cost analysis because of the effect of buffering (as discussed at the end of Section 3.3) and a better use of the block-based flag technique (Section 3.4).

Finally, we compare the memory space requirement (holding associated data structures) of each protocol and summarize the results in Table 3. As shown in the table, compared with BPCC and SCC, CFC, and AFC save $0.4 \sim 3.2$ MB memory space. This can be explained as follows: the CFC and AFC protocols use *physical* addresses to maintain the chain structure so that the size of the direct mapping table is minimized (as discussed at the end of Section 3.1.2); moreover, the BPCC and SCC protocols need to keep track of uncommitted pages, which also requires extra memory space.

TABLE 3
Memory Usage for Holding Data Structures

Protocol	Memory Overhead
CFC Protocol	59.88MB
AFC Protocol	59.88MB
BPCC Protocol	63.10MB
SCC Protocol	60.24MB

5.3 Evaluation of Advanced `flagcommit` Protocols

Now, we evaluate the performance of the extended `flagcommit` protocols with no-force buffer management and record-level concurrency control, denote as CFC_ex and AFC_ex. We compare them to the WAL-based commit protocol. Here, the cyclic commit protocols are not included for comparison, as they are designed for file systems and it is not clear in [34] how they can be extended to support the no-force buffer policy and record-level concurrency control.

5.3.1 Impact of Partial Programming Level

We first investigate how the maximum partial programming level (*max_c*) would affect the concurrency control and the performance of the extended `flagcommit` protocols.[9] Fig. 15a shows the transaction throughput of CFC_ex and AFC_ex under different values of *max_c*. For a better understanding of the performance, we also measure the transaction restart ratio (i.e., the average number of times that a transaction has to restart before a successful commit) (see Fig. 15b). The throughput of CFC_ex and AFC_ex is improved by 13 percent when we increase *max_c* from 2 to 3. This can be explained as follows: With a higher *max_c*, more transactions are allowed to update a page at the same time. As a result, the chance of access conflicts is reduced and, hence, fewer transaction restarts are needed (as shown in Fig. 15b, the restart ratio is reduced by nearly 40 percent when *max_c* varies from 2 to 3). This leads to a higher transaction throughput. However, the performance of CFC_ex and AFC_ex slightly drops when *max_c* exceeds 3. This is due to the following reasons. First, as shown in Fig. 15b, the transaction restart ratio does not decrease much when we further increase *max_c*. Second, with more concurrent updates, the committed records of a logical page are more likely to scatter over multiple physical pages. As a consequence, the cost of reading the latest page content becomes more expensive. Third, as more commit flags coexist on a flash page, in the course of garbage collection, more partial programming operations are required to preserve the protocols. These results imply that a partial programming level of 2 or 3 is good enough to support record-level concurrency control for the TPC-C workload. We set *max_c* at 3 in the rest evaluation of the CFC_ex and AFC_ex protocols.

5.3.2 Comparison with the WAL-Based Commit Protocol

We now compare the performance of `flagcommit` to the WAL-based commit protocol, and investigate the performance effect of extending `flagcommit` to support the no-force buffer policy and record-level concurrency control. Fig. 16 shows the transaction throughput and the commit response time of the basic CFC/AFC protocols, the extended CFC_ex/AFC_ex protocols, and the WAL-based commit protocol, under the default system setting.

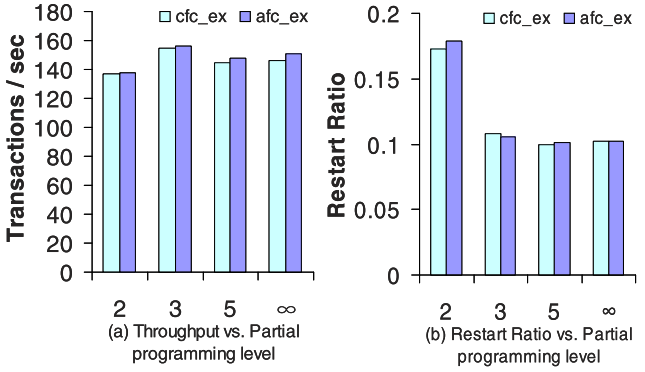

Fig. 15. Impact of the partial programming level (*max_c*).

Several observations are obtained from these results. First, by supporting the no-force buffer management policy and record-level concurrency control, we achieve 64 and 53 percent improvement in the transaction throughput, and 68 and 67 percent improvement in the commit response time for CFC and AFC, respectively. This is because with record-level concurrency control, the transaction restarts in CFC_ex and AFC_ex are greatly reduced, which consequently increases the transaction throughput, and with the no-force policy, committing a transaction only involves appending a commit log record and forcing the buffered log records to the flash disk (rather than forcing all its shadow pages), and thereby further improving the transaction throughput and shortening the commit response time. Second, CFC_ex and AFC_ex outperform WAL by 49 and 51 percent, respectively, in terms of the transaction throughput. This can be explained as follows: When a transaction aborts or restarts, WAL has an overhead of rollbacking the pages previously updated by this transaction, which involves getting these pages and their corresponding undo log records into the buffer pool, and restoring the contents of these pages according to the undo log records. In contrast, when a transaction rollbacks, CFC_ex and AFC_ex only need to remove the corresponding log records and discard (or invalidate) its shadow pages in the buffer pool. Furthermore, WAL requires writing more

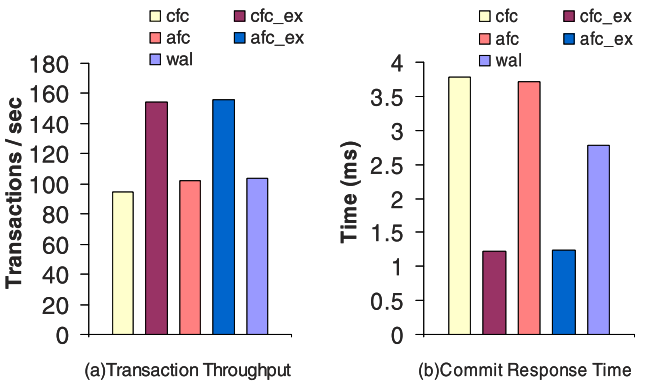

Fig. 16. Transaction throughput and commit response time.

9. Note that in practice the maximum partial programming level is usually up to 4-8 times on each flash page. The maximum number of concurrent transactions is also constrained by the size of the spare area of a flash page. In the experiment, we assume that there is always enough space for holding pending records.

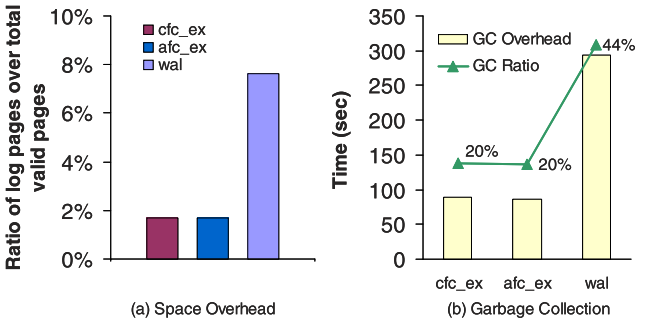

Fig. 17. Space overhead and garbage collection overhead.

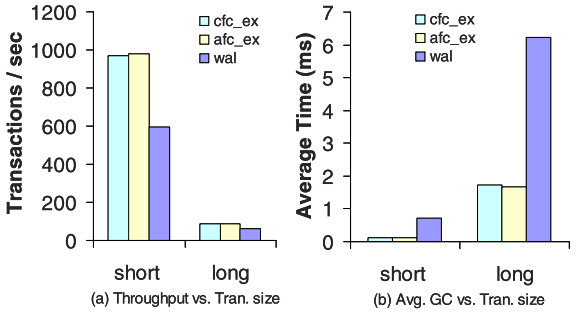

Fig. 18. Impact of transaction size λ.

log records (i.e., undo logs for aborted/restarted transactions and special logs such as transaction start/end and compensation records) than CFC_ex and AFC_ex during normal processing. Also, in WAL, before any updated page can be written back to the disk, its corresponding log records need to be forced to the disk if they are still buffered, while CFC_ex and AFC_ex only need to remove the corresponding log records from the log buffer. As a consequence, the number of log pages generated in WAL is larger than that in CFC_ex and AFC_ex (Fig. 17a). This makes its garbage collection overhead much higher than that of CFC_ex and AFC_ex (Fig. 17b).

5.3.3 Impact of Transaction Size

In this set of experiments, we study how the transaction size (i.e., the number of pages written by a transaction) would affect the performance of the commit protocols. Fig. 18 compares the throughput and the average garbage collection overhead per transaction for transactions of different sizes. This is done by splitting the original trace of transactions into two subtraces, according to the number of pages each transaction writes (denoted by λ). In specific, one subtrace is for short transactions ($\lambda \leq 10$), and the other is for long transactions ($\lambda > 10$). From Fig. 18a, we observe that the performance improvement of CFC_ex and AFC_ex over the WAL-based commit protocol is higher with short transactions. For example, CFC_ex outperforms WAL by 46 percent for long transactions, and such improvement increases to 64 percent for short transactions. The reason is as follows: For a short transaction, as it is more likely that all of its shadow pages have already been written back to the flash disk before it is committed, an overhead of appending a commit log record and forcing log records to the flash disk can be eliminated. This makes the advantage of CFC_ex and AFC_ex over WAL more obvious. In addition, recall that WAL incurs additional write operations when rollbacking those aborted pages already written to the flash disk, which consequently triggers more garbage collection activities. Such overhead is more significant for short transactions. As we can observe from Fig. 18b, CFC_ex and AFC_ex require 72-73 percent less time of garbage collection than WAL for long transactions, and 84-85 percent less time for short transactions.

5.3.4 Impact of Buffer Pool Size

In this section, we evaluate the impact of buffer pool size. Fig. 19a shows the transaction throughput of CFC_ex, AFC_ex, and WAL under various buffer pool sizes. To investigate the rationale behind the performance, we also measure their garbage collection overheads and plot the results in Fig. 19b. From these results, several observations can be made. First, the transaction throughput increases for all protocols when the buffer pool size is varied from 256 to 4,096 pages. This is because, with a larger buffer pool, the write and read hit ratios of the buffer pool are improved, and thus more I/O savings can be achieved. Second, the performance gap between AFC_ex and CFC_ex decreases as the buffer pool size increases, and they finally converge when the buffer pool grows larger than 2,048 pages. The reason is that, the bigger the buffer pool, the more the partial programming operations performed in main memory. As the performance of AFC_ex and CFC_ex differs in the number of partial programming operations, such a performance gap becomes smaller when the buffer pool size increases. Third, the improvement of AFC_ex and CFC_ex over WAL becomes more significant as the buffer pool size increases. In particular, the performance gap between CFC_ex/AFC_ex and WAL increases from 24/34 to 83 percent when we increase the buffer pool size from 256 to 4,096 pages. CFC_ex and AFC_ex benefit from a larger buffer pool in the following aspects. As discussed before, with a larger buffer pool, updating commit flags is more likely to be performed in main memory and thus more partial programming operations can be eliminated. Moreover, with a larger buffer pool, multiple updates issued to a logical page are more likely to be written on the same flash page and, hence, fewer versions of a logical page would coexist on the flash disk. Subsequently, fewer read operations are required when retrieving the latest content of a page. In addition, with fewer page versions, the cost of copying valid pages during each garbage collection is reduced. Meanwhile, each garbage collection process will reclaim relatively more obsolete pages and therefore less garbage collection activities are triggered.

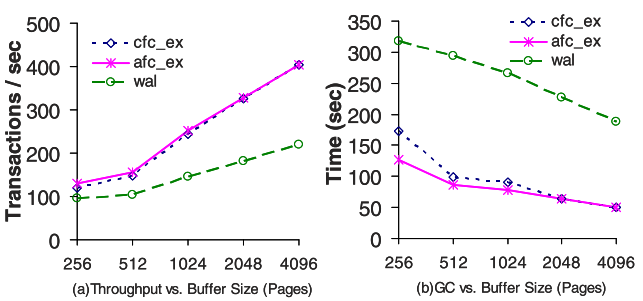

Fig. 19. Impact of buffer pool size.

As we can observe from Fig. 19b, the garbage collection overhead of CFC_ex and AFC_ex is decreased by 71 and 61 percent, respectively, when the buffer pool size is increased from 256 to 4,096 pages. In contrast, the garbage collection cost of WAL is reduced by 41 percent only, as it fails to benefit from the above aspects.

6 Related Work

There has been a stream of research on flash-aware data access techniques. Early work attempted to emulate the interfaces of traditional magnetic disks. This requires shielding certain unique drawbacks of flash chips from applications. For example, Kawaguchi et al. [23] proposed a flash translation layer to support transparent access to flash chips. As a result, conventional disk-based algorithms and access methods can work on flash disks without any modifications. Garbage collection mechanisms and wear-leveling techniques have been extensively investigated [16], [17].

Recently, the characteristics of flash disks have been exploited to enhance the performance of file systems and database systems from various aspects [19], [20], [27]. Several log-based storage mechanisms have been suggested to optimize write operations. Concerning flash-based DBMSs, Lee and Moon [26] presented a novel design of data logging called in-page logging (IPL). The idea is to log changes made to a data page in a reserved area of the flash block, instead of updating the page directly. When the log area is full, the change logs are merged to their data pages. Koltsidas and Viglas [25] suggested adding flash disks to the storage layer of a database system and investigated how to place pages based on their workloads. Chen [18] proposed *FlashLogging* for synchronous logging based on the observation that flash devices support sequential writes well. Nath and Gibbons [32] studied the problem of how to maintain very large random samples on flash storage. In addition, new buffer management algorithms (e.g., [24], [33]) have been proposed for flash devices. Flash-aware indexing and query processing techniques have also been intensively studied (e.g., [9], [29], [30], [37], [40], [41]).

In contrast, not much work has been done on flash-aware transaction management, especially for database systems. Besides the *cyclic commit* scheme [34], other related work includes [39] and [14]. In [39], Wu et al. proposed a fast recovery scheme for *flash-based file systems*, which commits log records into a special check region in order to avoid scanning the entire flash storage during recovery. Byun [14] proposed a new locking scheme called flash-two-phase-locking (F2PL) for concurrency control in a flash-based DBMS. F2PL achieves a high transaction performance by efficiently handling slow write/erasure operations in lock management processes. However, the problem of transaction recovery for flash-based DBMSs has not been studied in the literature. This study is an attempt to fill this void.

7 Conclusions and Future Work

In this paper, we have proposed two novel `flagcommit` protocols, namely CFC and AFC, for flash-based DBMSs. Our main idea exploits the fast random read access, out-place updating, per-page metadata, and partial page programming feature of flashes to optimize the performance of transaction processing and recovery. CFC and AFC ensure the atomicity and durability properties of transactions by using shadow pages and commit flags stored in these pages, which minimizes the need of writing explicit log records. CFC is designed to support fast recovery while AFC aims to achieve the highest transaction performance under a low transaction abort ratio. Both protocols support no-force buffer management and fine-grained concurrency control. Our performance evaluation based on the TPC-C benchmark shows that both CFC and AFC outperform the state-of-the-art recovery protocols.

As for future work, we plan to extend the proposed `flagcommit` protocols to MLC flashes and other NVRAM storage technologies (e.g., PCM). While they still can exploit fast random access performance of these storage devices, new questions open up. In particular, as MLC flashes do not support partial page programming, updating commit flags might involve out-place updating and thus becomes more expensive. Hence, advanced techniques to reduce the frequency of flag updating should be further explored. On the other hand, for PCM technology, as it allows overwriting data without incurring any erasure operation, the `flagcommit` protocols can work on it without any modification, as long as we reserve some space on a page as "spare area." Nevertheless, further research should be carried out to minimize the extra overhead caused by the `flagcommit` protocols (e.g., the maintenance of page mapping table and garbage collection). We also plan to implement the proposed commit protocols on real flashes where we can have access to the FTL.

Acknowledgments

The authors are grateful to the editor and the anonymous reviewers for their constructive comments that significantly improved the quality of this paper. This work was supported by the Research Grants Council of Hong Kong (Grants 210808 and 211510), Hong Kong Baptist University (FRG2/09-10/054), and Natural Science Foundation of China (Grant 60833005). Jianliang Xu is the corresponding author.

References

[1] Intel Information Technology, "Solid-State Drives in the Enterprise: A Proof of Concept," http://download.intel.com/it/pdf/Solid_state_drives_in_Enterprise.pdf, 2009.
[2] OSDL Database Test 2, http://osdldbt.sourceforge.net, 2012.
[3] PostgreSQL, "The World's Most Advanced Open Source Database," http://www.postgresql.org/, 2012.
[4] STMicroelectronics NAND01G-B, http://pdf1.alldatasheet.com/datasheet-pdf/view/131762/STMICROELECTRONICS/NAND01G-B.html, 2012.
[5] Hynix HY27US08281A, http://pdf1.alldatasheet.com/datasheet-pdf/view/170158/HYNIX/HY27US08281A.html, 2012.
[6] Samsung K9XXG08UXA, http://www.samsung.com/Products, 2012.
[7] Super Talent Technology, "SLC vs. MLC: An Analysis of Flash Memory," http://www.supertalent.com/datasheets/SLC_vs_MLCwhitepaper.pdf, 2012.
[8] TPC Benchmark C, "Standard Specification," http://www.tpc.org/tpcc/spec/tpcc-current.pdf, 2012.
[9] D. Agrawal, D. Ganesan, R. Sitaraman, and Y. Diao, "Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices," *Proc. VLDB Endowment*, vol. 2, pp. 361-372, 2009.

[10] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," *Proc. Usenix Ann. Technical Conf. (USENIX '08)*, June 2008.

[11] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrecy Control and Recovery in Database Systems.* Addison Wesley, 1987.

[12] L. Bouganim, B.T. Jónsson, and P. Bonnet, "uFLIP: Understanding Flash IO Patterns," *Proc. Fourth Biennial Conf. Innovative Data Systems (CIDR)*, 2009.

[13] J.S. Bucy and G.R. Ganger, "The Disksim Simulation Environment Version 3.0 Reference Manual," technical report, 2003.

[14] S. Byun, "Transaction Management for Flash Media Databases in Portable Computing Environments," *J. Intelligent Information Systems*, vol. 30, no. 2, pp. 137-151, 2008.

[15] *Flash Memories*, P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni, eds. Kluwer, 1999.

[16] L.-P. Chang and C.-D. Du, "Design and Implementation of an Efficient Wear-Leveling Algorithm for Solid-State-Disk Microcontrollers," *ACM Trans. Design Automation Electronic Systems*, vol. 15, no. 1, pp. 1-36, 2009.

[17] L.-P. Chang, T.-W. Kuo, and S.-W. Lo, "Real-Time Garbage Collection for Flash-Memory Storage Systems of Real-Time Embedded Systems," *ACM Trans. Embedded Computing Systems*, vol. 3, no. 4, pp. 837-863, 2004.

[18] S. Chen, "FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance," *Proc. 35th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2009.

[19] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Survey*, vol. 37, no. 2, pp. 138-163, 2005.

[20] J. Gray and B. Fitzgerald, "Flash Disk Opportunity for Server Applications," *Queue*, vol. 6, no. 4, pp. 18-23, 2008.

[21] 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 Computing Survey*, vol. 13, no. 2, pp. 223-242, 1981.

[22] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Survey*, vol. 15, no. 4, pp. 287-317, 1983.

[23] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," *Proc. USENIX Technical Conf.*, 1995.

[24] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," *Proc. Sixth USENIX Conf. File and Storage Technologies (FAST '08)*, pp. 1-14, 2008.

[25] I. Koltsidas and S.D. Viglas, "Flashing up the Storage Layer," *Proc. VLDB Endowment*, vol. 1, pp. 514-525, 2008.

[26] S.W. Lee and B. Moon, "Design of Flash-Based DBMS: An In-Page Logging Approach," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '07)*, pp. 55-66, 2007.

[27] S.-W. Lee, B. Moon, and C. Park, "Advances in Flash Memory SSD Technology for Enterprise Database Applications," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '09)*, 2009.

[28] P.M. Lewis, A. Bernstein, and M. Kifer, *Databases and Transaction Processing: An Application-Oriented Approach.* Addison Wesley, 2002.

[29] Y. Li, B. He, Q. Luo, and K. Yi, "Tree Indexing on Flash Disks," *Proc. IEEE Int'l Conf. Data Eng. (ICDE)*, 2009.

[30] Y. Li, S.T. On, J. Xu, B. Choi, and H. Hu, "DigestJoin: Exploiting Fast Random Reads for Flash-Based Joins," *Proc. 10th Int'l Conf. Mobile Data Management (MDM '09)*, 2009.

[31] 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 Systems*, vol. 17, no. 1, pp. 94-162, 1992.

[32] S. Nath and P.B. Gibbons, "Online Maintenance of Very Large Random Samples on Flash Storage," *Proc. VLDB Endowment*, vol. 1, pp. 970-983, 2008.

[33] Y. Ou, T. Härder, and P. Jin, "CFDC: A Flash-Aware Replacement Policy for Database Buffer Management," *Proc. Fifth Int'l Workshop Data Management on New Hardware (DaMoN)*, 2009.

[34] V. Prabhakaran, T.L. Rodeheffer, and L. Zhou, "Transactional Flash," *Proc. Eighth USENIX Symp. Operating Systems Design and Implementation (OSDI '08)*, 2008.

[35] R. Ramakrishnan and J. Gehrke, *Database Management Systems.* McGraw-Hill, 2003.

[36] K. Ross, "Modeling the Performance of Algorithms on Flash Memory Devices," *Proc. Int'l Workshop Data Management on New Hardware (DaMoN)*, 2008.

[37] D. Tsirogiannis, S. Harizopoulos, M.A. Shah, J.L. Wiener, and G. Graefe, "Query Processing Techniques for Solid State Drives," *Proc. 35th ACM SIGMOD Int'l Conf. Management of Data (SIGMOD)*, 2009.

[38] D. Woodhouse, "JFFS: The Journaling Flash File System," *Proc. Ottawa Linux Symp.*, July 2001.

[39] C.-H. Wu, T.-W. Kuo, and L.-P. Chang, "Efficient Initialization and Crash Recovery for Log-Based File Systems over Flash Memory," *Proc. ACM Symp. Applied Computing (SAC '06)*, 2006.

[40] C.-H. Wu, T.-W. Kuo, and L.-P. Chang, "An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems," *ACM Trans. Embedded Computing Systems*, vol. 6, no. 3, article 19, 2007.

[41] D.Z. Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W.A. Najjar, "MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices," *Proc. USENIX Conf. File and Storage Technologies (FAST '05)*, 2005.

Sai Tung On received the BEng degree in software engineering from Tsinghua University, Beijing, China. He is currently working toward the MPhil degree in the Department of Computer Science at Hong Kong Baptist University. His research interest lies in data management on novel storage media. He is a student member of the ACM.

Jianliang Xu received the BEng degree in computer science and engineering from Zhejiang University, Hangzhou, China, in 1998 and the PhD degree in computer science from the Hong Kong University of Science and Technology in 2002. He is an associate professor in the Department of Computer Science, Hong Kong Baptist University. He is a member of the Database Group at Hong Kong Baptist University (http://www.comp.hkbu.edu.hk/~db/). 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 100 technical papers in these areas. He is a senior member of the IEEE.

Byron Choi received the BEng 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, Hong Kong Baptist University. Before this, he was an assistant professor in the School of Computer Engineering, Nanyang Technological University (NTU) for three years (2005-2008). He was a research associate at the University of Edinburgh in 2005 and a summer student intern for the Galax project at AT&T Labs, Florham Park, New Jersey. He visited the HKUST Theoretical Computer Science Group in 2003. He is a member of the ACM and the IEEE.

Haibo Hu is a research assistant professor in the Department of Computer Science, Hong Kong Baptist University (HKBU). Prior to this, he held several research and teaching posts at the Hong Kong University of Science and Technology (HKUST) and HKBU. He received the PhD degree in computer science from HKUST in 2005. His research interests include mobile and wireless data management, location-based services, and privacy-aware computing. He has published more than 20 research papers in international conferences, journals, and book chapters. He is also the recipient of many awards, including ACM-HK Best PhD Paper Award and Microsoft Imagine Cup.

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