Transactional In-Page Logging for Multiversion Read Consistency and Recovery[*]

Sang-Won Lee [#1], Bongki Moon [*2]

[#]*School of Info. & Comm. Engr., Sungkyunkwan University*
Suwon 440-746, Korea
[1]swlee@skku.edu

[*]*Dept. of Computer Science, University of Arizona*
Tucson, AZ 85721, U.S.A.
[2]bkmoon@cs.arizona.edu

***Abstract*—Recently, a new buffer and storage management strategy called In-Page Logging (IPL) has been proposed for database systems based on flash memory. Its main objective is to overcome the limitations of flash memory such as erase-before-write and asymmetric read/write speeds by storing changes made to a data page in a form of log records without overwriting the data page itself. Since it maintains a series of changes made to a data page separately from the original data page until they are merged, the IPL scheme provides unique opportunities to design light-weight transactional support for database systems. In this paper, we propose the *transactional IPL* (*TIPL*) scheme that takes advantage of the IPL log records to support multiversion read consistency and light-weight database recovery. Due to the dual use of IPL log records, namely, for snapshot isolation and fast recovery as well as flash-aware write optimization, *TIPL* achieves transactional support for flash memory database systems that minimizes the space and time overhead during normal database processing and shortens the database recovery time.**

I. Introduction

Since solid state drives (SSDs) based on NAND type flash memory were introduced to the storage market a few years ago, great strides have been made in overcoming the poor random write performance and increasing the bandwidth and throughput of flash memory SSDs. Due mainly to their superior characteristics such as high throughput for random I/O and low energy consumption, flash memory SSDs are now considered crucial or even indispensable for building high performance large scale data systems. As the preliminary experience and analysis show, flash memory SSDs outperform magnetic disk drives with respect to transaction throughput and energy conservation for OLTP systems [1]. Therefore, a large-scale adoption of flash memory SSDs in database machines [2] and tier zero storage for data centers [3] is not surprising any more.

Such a successful adoption of flash memory SSDs is in part attributed to the impressive research efforts that have been made in the past few years. Most of the work has been focused on the efficient use of flash memory for storage and database systems, just to name a few for example, flash translation layers [4], indexing structures [5], [6], buffer and storage management [7], [8], query processing [9], and logging and temporary data spaces [10], [11]. However, there has been little (if any) work on transactional support for flash memory based database systems. Given that the transaction throughput can increase by orders of magnitude by utilizing flash memory SSDs instead of magnetic disk drives, it is imperative that the atomicity, consistency and durability are ensured for a large number of concurrent transactions with as much efficiency.

The main objective of this work is to develop efficient transactional support and fast recovery for flash memory based database systems. We present a novel *Transactional In-Page Logging* (*TIPL*), which will be built on the In-Page Logging (IPL) scheme as an underlying platform for buffer and storage management. The In-Page Logging has been developed for flash memory database systems to overcome the limitations of flash memory such as erase-before-write and asymmetric read/write speeds [7]. It attempts to address the limitations by storing changes made by a transaction in a form of physiological log records without overwriting data pages themselves. By maintaining a series of changes made to a data page separately from the data page kept intact until they are merged, the IPL scheme provides us with unique opportunities for utilizing the log records to design an efficient transactional support strategy for database systems.

Most disk-based database systems rely on traditional techniques for transactional support such as write-ahead logging (WAL), steal and no-force buffer management, and a multi-phase recovery procedure [12]. In contrast, the *TIPL* scheme supports snapshot isolation [13] for concurrent transactions with an elaborate *read consistent merge* algorithm by enabling different versions of a data page to be reconstructed efficiently from the unmodified data page and its log records. Furthermore, *TIPL* achieves a light-weight but robust transactional support by eliminating the need of write-ahead logging and by enabling fast redo-only recovery or even instant restart procedures.

Due to the dual use of IPL log records, namely, for multiversion read consistency and fast recovery as well as flash-aware

[*]This work was partly supported by MEST, Korea under NRF Grant (No.2010-0025649) and NRF Grant (No.2010-0026511). This work was also sponsored in part by the U.S. National Science Foundation Grant IIS-0848503. The authors assume all responsibility for the contents of the paper.

write optimization, *TIPL* achieves transactional support for flash memory database systems that minimizes the space and time overhead during normal database processing and shortens the database recovery time. The key contributions of this work are summarized as follows.

- This paper investigates opportunities provided by in-page logging and refines the strategy further to minimize or eliminate the need of write-ahead logging during normal database processing and undo recovery at system restart.
- In order to support snapshot isolation [13], *TIPL* provides a mechanism that can reconstruct different versions of a data page for concurrent transactions with different timestamps very efficiently by utilizing the data page and its log records co-located in the same flash memory block. Optimization techniques are provided for read consistent merges at the presence of long running transactions.
- *TIPL* provides two alternative protocols, *three phase commit* and *redundant logging*, for committing transactions. While the former eliminates the need of redo recovery so that a database system can restart *instantly* from a failure, the latter minimizes the commit-time delay and allows a database system to restart with redo-only recovery by resorting to a redundant system-wide logging.

The rest of this paper is organized as follows. Section II reviews the IPL scheme and describes its unique features that can be utilized for transactional support. Section III presents the support for snapshot isolation and the read consistency merge algorithm of *TIPL*. Section IV proposes novel strategies to ensure the durability of committing transactions, and presents instant restart and redo-only recovery procedures for fast recovery. Section VI evaluates the performance impact of *TIPL*. Lastly, Section VII surveys the related work, and Section VIII summarizes the contributions of this paper.

II. Motivations

Most existing database systems maintain a single copy of each data object in disk drives and perform update operations *in place*. The previous versions of a data object are often kept in a separate data space called rollback segment or version store, if they are required for snapshot isolation or multiversion concurrency control. On the other hand, the In-Page Logging (IPL) scheme, which has been proposed to optimize the write performance of flash memory devices [7], stores changes made to a data page in a form of log records leaving the data page itself intact until the log records are merged to the data page. This section will review the fundamentals of the IPL scheme, discusses its unique features that can be utilized to realize transactional support, and identifies tasks to fulfill required for developing *TIPL* as a light-weight and robust transaction management strategy.

A. Fundamentals of IPL

Due to the erase-before-write limitation of flash memory, even a single record update may require an entire page containing the record to be copied to a clean page. This is one of the main causes of slow random writes, and this also leads to rapid consumption of clean blocks and shortened life span of flash memory devices. IPL avoids this problem by storing only the change (or redo) *log* of an update instead of writing the updated page in its entirety. Furthermore, multiple log records belonging to the same data page are written together to a log sector in flash memory such that the log sector is *co-located* with the data page in the same flash memory block (also known as an erase unit). Log records are flushed to flash memory when the data page is evicted or the in-memory log sector becomes full.

When a flash memory block runs out of its free log sectors, IPL allocates a clean block and *merges* the data pages and log sectors from the current block to the new one by applying the log records to their corresponding data pages. On a page fault, a data page must be read along with all of its log records from a flash memory block, and then the latest version of the page has to be computed by applying its log records.

For the past few years, we have witnessed remarkable improvements of flash memory SSDs, in particular, for the random write performance. This can be attributed mainly to the advances of a software module called FTL (flash translation layer) [14] that is run by an SSD controller. Most contemporary FTLs carry out a page overwrite operation by creating a new version of the page at the granularity of pages. Thus, even when only a very small fraction of a page is updated, an FTL will write a new page in its entirety including the unchanged portion of the page. Evidently, the change logs between the old and new versions of the page are not preserved at all.

In contrast, IPL keeps track of all the changes at the transactional semantic level, and tends to consume flash memory much less than the FTLs, which are developed for general purpose applications and are oblivious to the transactional database semantics. Since the unit of write by IPL is a sector, which is typically one fourth of a page, the benefit of IPL with respect to write amplification may diminish if sector write is no longer supported by flash memory chips (for example, MLC chips). However, SLC flash memory chips are expected to continue to support sector writes, and enterprise OLTP database servers prefers to use SLC-based SSDs for performance concerns. For this reason, the effectiveness of IPL will remain valid for enterprise class databases.

B. Towards Transactional IPL

Since it maintains a series of changes made to a data page separately from the original data page kept intact until they are merged, the IPL scheme provides unique opportunities to design light-weight transactional support for database systems. Below we describe the IPL features we can take advantage of to provide transactional support, and we propose how they should be extended or redefined to realize *TIPL* as a fully functioning, light-weight transaction management system.

Conflicting Order Preserved: In general, log records must accurately reflect the order in which update operations are actually executed. This is because, when a database system restarts from a failure, the recovery system will re-execute some of the updates that happened before the failure. However, it is not necessary that the log records reflect the order of all updates. The order needs to be maintained only for the *conflicting* ones, for which relative order makes a difference [15].

Apparently, as log records are scattered across numerous flash memory blocks, there is no way IPL maintains a global ordering for the entire log records. Nonetheless, the IPL log records can still be used to recover a database system consistently from a failure. This is because every pair of conflicting updates must have been performed on the same data page, and all the update logs belonging to the same data page are recorded in the same log sector in chronological order as they are produced.

Efficient Version Reconstruction: IPL accumulates the changes made to a data page in concise physiological log records instead of propagating the updated page in its entirety. By the unique way of propagating changes, IPL inherently maintains multiple versions of a data page in a space efficient manner. By controlling which versions are maintained for how long, those versions of a data page can be used for the purpose of snapshot isolation and multiversion concurrency control for concurrent transactions. Note that most existing database servers store not only the information about old data pages in the rollback segment but also the undo and redo images of changed data in the system log for the purpose of recovery and multiversion concurrency control [16]. In contrast, *TIPL* does not store redundant log information, unless redundant logging is explicitly requested to minimize commit-time delay of transactions. (See Section IV-C.)

The exact scope of versions to maintain for a data page must be determined by the life spans of transactions that access the data page, and this mechanism will have to be incorporated into the *TIPL* scheme. Full details of the *TIPL* scheme regarding version reconstruction and multiversion read consistency will be given in Section III.

No WAL Protocol: Write-Ahead Logging (WAL), combined with force-write of the log tail at commit time, is the fundamental rule that ensures the last committed value of each data item is always available in stable storage [15]. When a data page is updated and propagated under the IPL scheme, however, only the redo log records are written to stable storage (*i.e.*, flash memory) leaving the data page intact in the database. This may make it unnecessary to follow the traditional WAL protocol as an undo rule. A database system could be relieved of the burden of writing undo log records prior to writing the data page to stable storage. Without such overhead as synchronizing the propagation of logs and data (*e.g.*, inter-process communication [17]), the database system would be more efficient during normal database processing.

Note that the redo log records will eventually be merged to their corresponding data pages, and they cannot be rolled back once merged. Therefore, the redo records to be merged must be determined carefully by the status of corresponding transactions. Again, this mechanism will have to be incorporated into the *transactional IPL* scheme.

Fast Recovery: Regardless of support for multiversion concurrency control, most conventional disk-based database systems adopt a multi-phase recovery procedure that typically includes a redo phase followed by a undo phase (*e.g.*, the ARIES recovery algorithm [12]). With the steal policy for dirty page propagation, a database may contain updates made by uncommitted transactions at any moment during normal online processing. Therefore, when a database system is about to restart from a failure, undo recovery must be performed to roll back the uncommitted updates. These uncommitted updates can be rolled back by the undo log records stored in the system log or by the old versions stored in the rollback segments if they are available.

On the other hand, even with the steal policy, the IPL scheme avoids propagating a dirty page itself immediately to the database by writing only the change logs into a corresponding log sector in flash memory. This implies that we can get away with the undo recovery as long as those uncommitted change logs are not merged to their data pages until the corresponding transactions commit. With such an elaborate merge mechanism in place, it would be unnecessary to explicitly perform undo actions for incomplete transactions at the time of a failure. Instead, any necessary undo action could be performed implicitly as part of online database processing by preventing any uncommitted change logs from being merged to the data pages [7]. Section IV presents two novel commit protocols for the *TIPL* scheme and recovery procedures that restarts the system instantly or with only redo recovery.

III. Multiversion Read Consistency

The multiversion storage model has been adopted by many commercial and open-source database servers (*e.g.*, Oracle, SQL Server, PostgreSQL) to support snapshot isolation [13] and multiversion concurrency control [18]. One of the key ingredients for the snapshot isolation and multiversion concurrency control is to provide concurrent transactions with an individual snapshot of database as of the start time of the transactions. The snapshot of database is essentially a set of specific versions of data determined by the start time of a transaction. While an update operation is always performed on the current version (subject to a write lock availability), the snapshots of database are used to allow concurrent transactions to read different versions of a data object without blocking each other.

Most database servers maintain multiple versions by storing the current version of a data object in the data page and storing the previous versions separately in a data space called rollback segment (or version store). When a transaction attempts to read a data object, it must be ensured that the transaction reads the *correct* version of the data object. The correct version is defined as the most recent version whose timestamp precedes the (start) timestamp of the transaction. The correct version can be obtained by reconstructing a data page from the current data page and the old data stored in the rollback segment or version store. This process of version reconstruction may be costly if it requires accessing versions scattered across several rollback segments [11].

In this section, we describe how the *TIPL* scheme utilizes redo log records to reconstruct versions efficiently, and present an elaborate merge algorithm to support multiversion read consistency as well as its optimization techniques for long running transactions. The novelty of our approach is that the IPL log records are used to serve the dual purposes, namely, (1) multiversion and recovery support and (2) write optimization in a seamlessly integrated way.

A. Version Reconstruction

TIPL stores multiple versions of a data object but as a combination of a data page current as of a certain point in past time and a series of redo log records. We assume that each IPL log record is associated with a timestamp and stores the id of a transaction that creates the log record. The timestamps are globally unique and ever-increasing. They reflect the chronological ordering of update operations applied to the database. In this regard, the timestamps are similar to the log sequence numbers (LSNs) used by traditional recovery systems, but different in that the same timestamp generation mechanism is used to provide timestamps for transactions as well as log records.

When a transaction T_i with timestamp *ts(T_i)* attempts to read a data page p, *TIPL* reconstructs the version of p current as of *ts(T_i)*. This is initiated by fetching the copy of p from flash memory (unless the correct version of p is already cached in buffer) and collecting all its committed log records from the log sectors in buffer and flash memory. Then, while inspecting each of the collected log records l in chronological order, l is applied to p if *ts(l)* $<$ *ts(T_i)*. In other words, all the committed log records of p whose timestamps are older than *ts(T_i)* are applied to p to reconstruct the correct version of p for T_i.

In the version reconstruction steps described above, it is implicitly assumed that the copy of p currently available in the database is not newer than *ts(T_i)*. If the copy is newer than *ts(T_i)*, that is, it contains any data object updated at a time later than *ts(T_i)*, the correct version of p for T_i cannot be reconstructed, because *TIPL* does not maintain any undo log record. Section III-B describes how this problem of being unable to reconstruct required versions can be prevented.

Cost of Version Reconstruction: In most existing database servers with multiversion support, the current version of a data object is stored in the data page, and the previous versions of the data object are stored separately in rollback segments or version stores. The current version stored in the data page contains a pointer to its previous version, which may in turn contain a pointer to even earlier version of the data object. To reconstruct the correct version of a data page p for a transaction T_i with timestamp *ts(T_i)*, the chains of pointers may have to be followed (in reverse chronological order) starting from the current version until the correct versions of all data objects (*i.e.*, those with the latest timestamp less than *ts(T_i)*) are found. If there exists a data object that has been updated by multiple transactions, then its versions may have been *scattered* physically across multiple rollback segments, because a transaction is typically assigned its own rollback segment. The obvious drawback is that the cost of reconstructing a correct version could be high if a long chain of pointers had to be chased, causing as many random I/O operations as the pointers to chase.

In contrast, *TIPL* stores log records in the same flash memory block where their corresponding data pages are stored. Furthermore, the log records belonging to the same data page are clustered in one or more log sectors in the block. Therefore, even for a data object frequently updated by many transactions, the correct version would be reconstructed quickly by reading only a few log sectors sequentially from the same flash memory block.

B. Read Consistent Merge

If a flash memory block runs out of free log sectors, a merge operation is triggered by the IPL storage manager. When it occurs, for each of the data pages stored in the block, all the corresponding log records are applied to the page to compute the latest version. All the newly computed data pages are then moved to a clean flash memory block allocated by the IPL storage manager, and the old flash memory block is garbage-collected and erased.

If the merge operations are requested without consideration of the status of transactions, the database system may fall into an unrecoverable state [7]. Suppose a log record is about to be merged to a data page, and the transaction T_i that has created the log record is still active. If the transaction T_i gets aborted or the entire system crashes after the merge, then there is no way the change made by T_i is rolled back. To address this concern, a *selective merge* algorithm has been proposed [7]. The selective merge guarantees recoverability from aborted or incomplete transactions, simply by keeping log records from being applied to data pages if the corresponding transactions are still active at the time of merge.

The selective merge algorithm, however, does not support the multiversion read consistency for concurrent transactions, because it allows all committed change logs to be merged regardless of the timestamps of change logs and the timestamps of active transactions. To ensure that read consistency as well as recoverability is supported, a log record l can be merged to its data page p, only if (1) the transaction that created the log record l has committed, and (2) there is no active transaction that will access the version of p created by the log record l. We propose a new merge algorithm called *read consistent merge* to deal with the requirements. As is presented in Algorithm 1, the read consistent merge algorithm satisfies the condition (2) by allowing a log record to be merged only if its timestamp is less than the timestamp of the oldest transaction among all active ones (denoted by *ts(T_{oldest})*). That is, a log record l in a block will be applied to its data page p if $ts(l) < ts(T_{oldest})$. Otherwise, the log record l will be carried over to a new block without being merged to p.

Note that the read consistent merge algorithm is more conservative than the selective merge algorithm in that the

Algorithm 1: Read Consistent Merge

```
Input:   $B_o$: an old flash block to merge
Output:  $B$: a new flash block with merged content

procedure $Merge(B_o, B)$
1:  allocate a free flash block $B$
2:  if carry-over-fraction $> \tau$ then
3:      attach $B$ to $B_o$ as an overflow log
4:      return $B_o$ as $B$
    endif
5:  for each committed log record $l$ in $B_o$ do
6:      if ts(l) $<$ ts($T_{oldest}$) then
7:          apply $l$ to its data page $p$
        endif
    endfor
8:  write all data pages to $B$
9:  compact and write all remaining log records to $B$
10: erase and free $B_o$
```

number of log records merged by the read consistent merge is always no more than the number of log records that would be merged by the selective merge. This may have a negative impact on the overall performance of *TIPL* with respect to the utilization of flash memory and the cost of version reconstruction. If there exists a long running transaction, then the timestamp of the oldest active transaction, *ts(*T_{oldest}*)* in Algorithm (1), may become too old to merge a sufficient number of log records in the block. Then, the log sectors will be wasted because too many log records have to be carried over to a clean flash memory block, and the cost of version reconstruction will increase because more log records have to be applied to compute a correct version for a transaction.

C. Optimization of Read Consistent Merge

One way of dealing with the performance concern caused by a long running transaction is to define the notion of database snapshot *narrowly* for each transaction. That is, the snapshot is taken not for the entire database but only for the set of relational tables accessed by a transaction, assuming that the information is available from the query compiler or optimizer. If that is the case, the timestamp *ts(*T_{oldest}*)* in Line 6 of Algorithm 1 can be refined to

$$min\{ts(T_i) \mid active\ T_i\ accesses\ l's\ table\}.$$

Since this timestamp is no smaller (or older) than *ts(*T_{oldest}*)*, more log records are expected to be merged to data pages without being carried over to a clean flash memory block.

Another way of dealing with a long running transaction (say T_{long}) is to exclude the transaction from consideration when the read consistent merge is executed. In other words, the value of *ts(*T_{oldest}*)* in Line 6 is determined by all active transactions excluding T_{long}.

In an example scenario shown in Figure 1, the timestamp of T_{long} would be used as the value of *ts(*T_{oldest}*)* by the

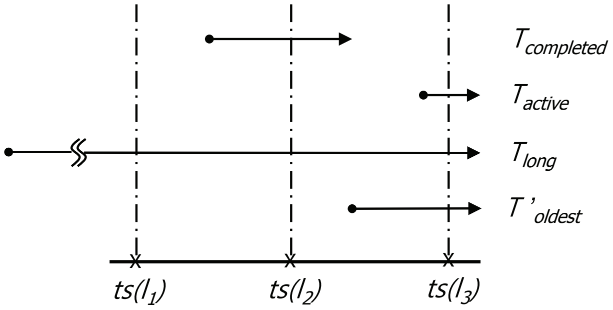

Fig. 1. A Scenario with a Long-Running Transaction

read consistent merge algorithm, if T_{long} were not excluded. Furthermore, none of the three log records l_1, l_2 and l_3 would be merged to their data pages, because their timestamps would be all newer than the timestamp of T_{long}. With T_{long} excluded, however, the timestamp of T'_{oldest} will be used as the value of *ts(*T_{oldest}*)* instead of T_{long}, and the log records l_1 and l_2 can be merged to their data pages. Here, T'_{oldest} denotes the oldest among the active transactions excluding T_{long}.

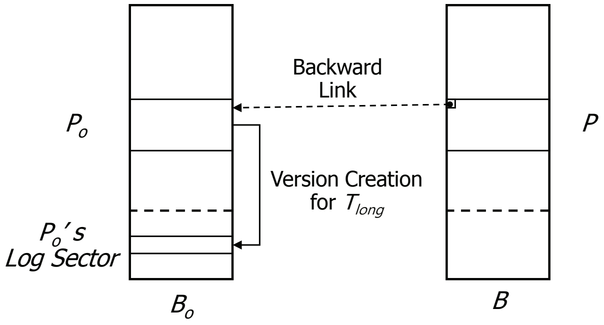

Fig. 2. Backward Link between the Old and New Blocks

This optimization strategy will merge log records more aggressively, because any log record whose timestamp is older than T'_{oldest} can be merged. However, when a log record l is merged to a data page p, if the following conditions are satisfied:

(1) $ts(T_{long}) < ts(l) < ts(T'_{oldest})$, and
(2) p is accessed by T_{long},

then the read consistent merge algorithm should not erase and free the old block (denoted by B_o in Algorithm 1) but instead insert a backward link from the new block (denoted by B) to the old one (B_o). (See Figure 2 for an illustration.) Then, a correct version of the data page p can be reconstructed for T_{long} from the old version of p and its redo log records stored in B_o. The old block B_o can be erased and freed after T_{long} terminates.

In effect, this strategy allows to provide database snapshots for different transactions from different sets of versioned data pages. Consequently, at the cost of additional space to keep old flash memory blocks (*e.g.*, B_o for T_{long}), the cost of version reconstruction will remain nominal for all the rest of transactions without giving up the read consistency for a long running transaction. It is worth noting that this strategy can also be used to support time travel by deferring garbage collection for old flash memory blocks like B_o.

D. Atomicity of Merge

The read consistent merge plays a key role in achieving snapshot isolation and recovery under the *TIPL* scheme. A merge operation involves copying data pages from an old flash memory block to a newly allocated clean block. Since the size of a flash memory block is typically much larger than a page (*e.g.*, typically 256KB for a block and 4KB for a page), a merge operation will take long and may be prone to be interrupted by a system failure. Obviously, without the atomicity of merge (*i.e.*, all or nothing), *TIPL* would not guarantee the consistency of database.

Fortunately, it is relatively simple to ensure the atomicity of merge. In most flash memory storage devices, there is a software layer called flash translation layer (FTL) that maintains a mapping table for flash memory blocks for logical-to-physical address mapping. *TIPL* can guarantee the atomicity of merge by allowing the physical address of a new block B to replace the physical address of an old block B_o in the mapping table, *only after* copying pages from B_o to B is finished successfully. In fact, this is similar to the shadow paging mechanism proposed for atomic page propagation [19].

IV. Recovery

A system failure is inevitable in large scale database servers. When it happens, the system must be recovered to a consistent state such that the atomicity and durability of transactions are ensured. In this section, we describe how *TIPL* eliminates the need of undo recovery except for transactions aborted during normal online processing. We then propose two novel strategies called *three phase commit* and *redundant logging* to ensure multiversion read consistency and durability of transactions at low cost.

A. No Undo Recovery

The *steal* is a buffer management policy commonly adopted by most database servers. Under this policy, a dirty page frame may be stolen from the buffer and written to the database even before the transactions commit that have updated the page. Consequently, when a database system restarts from a failure, undo recovery must be carried out to roll back the changes made by incomplete transactions.

The *TIPL* scheme also adopts the steal policy and allows any in-memory log sector to be flushed to flash memory before transactions that have added log records to the log sector commit. It usually happens when an in-memory log sector becomes full or its corresponding data page is evicted from the buffer pool. This implies that, when it happens, some of the changes made by an uncommitted transaction may be written to stable storage.

A major departure from the conventional way of propagating updates is, however, the fact that *TIPL* leaves original data pages intact in the database even after in-memory log sectors are saved in flash memory. Since the original data pages remain intact in the database, *TIPL* can roll back uncommitted updates simply by eliminating (or ignoring) them from flash memory log sectors by the read consistent merge (shown in Algorithm 1). In summary, though *TIPL* adopts the steal policy, the net effect is equivalent to that of the no-steal policy. Hence the undo recovery becomes completely unnecessary under the *TIPL* scheme, as long as it is ensured that uncommitted log records are prevented from being merged to data pages. Even though uncommitted log records stored in flash memory will survive a failure, they will be eventually discarded by read consistent merge operations during normal online processing resumed after the failure.

Individual Rollback: The only exception to the *no undo recovery* rule by *TIPL* is rolling back transactions that are aborted during normal online processing (*e.g.*, by a deadlock). When a transaction T is aborted, all the locks held by T are usually released immediately making all the uncommitted changes visible unless they are rolled back. Thus, all the uncommitted changes made by T must be rolled back before T is removed from the list of active transactions (or the transaction table).

In addition to the standard steps taken such as adding an `abort` record for T to the system log, there are three things that should be taken care of for an aborted transaction:

(1) uncommitted updates reflected in the cached images of data pages,
(2) uncommitted updates still kept in in-memory log sectors, and
(3) uncommitted updates already flushed in flash memory log sectors.

For an aborted transaction T, if there is a cached image of any data page containing T's uncommitted updates, the cached image will be restored to a consistent state by reversing the effects of the uncommitted updates. The uncommitted updates are recorded in log records that may reside in the buffer pool or flash memory. Therefore, this process may involve reading log records from flash memory.

As for the steps (2) and (3), if T's log records are still kept in memory, they will be removed from the in-memory log sectors. In contrast, for T's log records that have already been flushed to flash memory, *TIPL* does not perform any explicit operation to remove them from the log sectors in flash memory. Instead, *TIPL* leaves them in flash memory until they are discarded by the read consistent merge algorithm, when the flash memory blocks containing the log records are merged.

One might be concerned that these uncommitted updates could present inconsistent database snapshots for other transactions, during the period while uncommitted updates remain in flash memory. However, *TIPL* still guarantees multiversion read consistency even at the presence of the uncommitted updates in flash memory. This is because *TIPL* can reconstruct correct versions from original data pages and committed updates only, as described in Section III-A. In other words, those uncommitted updates remain *invisible* to other concurrent transactions until they are removed permanently.

One might wonder if the benefits of *no undo restart* might be canceled by existing techniques such as deferred modifications that have been already adopted by commercial DBMSs. For instance, when the buffer pool is large enough, uncommitted dirty pages do not have to be stolen. Thus, undo log records are kept in memory and undo recovery can be avoided. The performance gain by deferred modification could be significant in practice. However, this deferred modification technique alone does not obviate the need of undo recovery altogether, especially when aggressive writes are desired, for instance, as required by incremental checkpointing for fast recovery [20]. This is the case when flash memory based DBMSs can benefit from the *TIPL*'s no undo strategy.

B. Three Phase Commit

The *no-force* is another buffer management policy commonly adopted by most database servers to avoid excessive I/O for committing transactions. *TIPL* also adopts the no-force policy. When a transaction commits, in-memory log sectors are forced out to flash memory if they contain at least one log record of a committing transaction. This is equivalent to flushing the log tail to a stable storage when a transaction commits in the conventional database systems, except that the log records to be forced out may come from one or more buffer frames in the pool instead of a single system-wide log tail. To expedite commit-time force-out operations, each transaction needs to keep track of dirty pages updated by itself.

However, there is a potential drawback in this approach with respect to propagating changes to the database. Note that there are two separate processes involved in the update propagation: (1) copying change logs from the in-memory log sectors to flash memory blocks, and (2) merging the change logs stored in the flash memory log sectors to the data pages in the flash memory block. (The merge operations are performed by the read consistent merge algorithm shown in Algorithm 1.) Note also that these two types of operations are performed independently from each other. Suppose a read consistent merge is initiated for a flash memory block containing a log record of a transaction T. If T had committed and its entry had already been removed from the transaction table, then the read consistent merge algorithm could not tell the status of T and which log records are committed.

One remedy for this problem is to keep the transaction status even for committed ones in the transaction table until all of their log records are merged to data pages in flash memory. Since there is no way to upperbound the time gap between the two types of operations without creating a superfluous interdependency between them, the status of committed transactions may have to be maintained in the transaction table for a long while. The size of a transaction table may grow quickly as the transaction throughput increases, and this may become a non-trivial burden on a large scale OLTP system.

We propose a *three phase commit* procedure for committing transactions to address this concern. This procedure has an additional phase that precedes the IPL no-force policy. When a transaction T is ready to commit, the three phase commit procedure is carried out as follows.

```
1:  Add an entrust log record to the in-memory log sector for each data page updated by $T$.
2:  Force out the in-memory log sector of each data page updated by $T$ to the corresponding flash memory log sector.
3:  Write a commit log record to the system log.
```

The `entrust` is a new type of log records that helps decide quickly which log records are committed or not. An entrust log record contains only two pieces of information, namely, the type identifier of a log record and the identifier of a committing transaction. Therefore, the additional overhead for adding entrust log records is expected to be negligible. The `commit` log record in Phase Three is no different from the commit record used by a conventional recovery system. No transaction is considered committed until the commit log record is written to the system log. Once a commit log record is successfully written, the entry of T can be removed from the transaction table.

There is still a remaining concern about recoverability. Suppose the commit procedure has been initiated for a transaction T. Then, for the data pages updated by T, entrust log records are added to their in-memory log sectors, which are then forced out to one or more flash memory blocks that store the data pages. Since the *TIPL* storage manager can initiate a read consistent merge independently of the commit procedure, any of those flash memory blocks can be merged before a commit record for T is written to the system log. What happens if the system crashes at this moment, that is, after one of the flash memory blocks is merged but before a commit record for T is written to the system log? As the commit record is not yet written to the system log, T is not considered committed and the changes made by T must be rolled back. However, it will be impossible to roll all of them back, because some of the changes have already been merged to the data pages in the flash memory block.

Fortunately, this potentially problematic scenario can be prevented with ease by disallowing any T's log record to be merged between the second and third phases of the commit procedure. Specifically, a log record l in Line 5 of Algorithm 1 is considered committed *only* if

(1) an entrust log record having l's transaction id is found in the log sector, *and*

(2) the transaction id of l is *not* found in the transaction table.

When a read consistent merge is initiated for a block, the merge algorithm can tell that a commit procedure has already started for a transaction T, if an entrust log record with T's id is found in a log sector of the block. Then, it looks up the transaction table and determines that T is committed indeed if T is not in the transaction table any longer. Otherwise, T is considered still active and T's log records will not be merged.

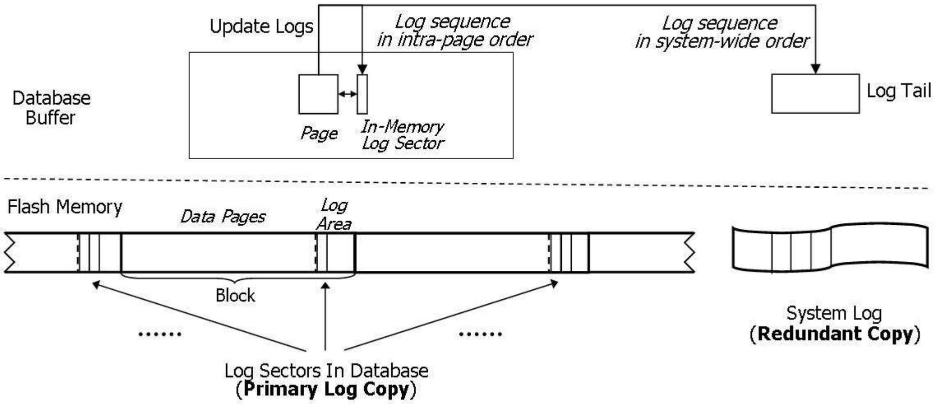

Fig. 3. Architecture for Redundant Logging

C. *Redundant Logging*

As will be discussed in Section V, the three phase commit can make the logic for a system restart extremely simple by eliminating the redo recovery altogether. However, it may incur additional I/O operations during the normal online processing. Whenever a transaction commits, one or more in-memory log sectors must be forced out to flash memory if they contain at least one log record of the committing transaction. This procedure will increase the likelihood of those in-memory log sectors to be flushed *prematurely* to flash memory before they become full of log records. For a transaction that updates a large number of data pages, the commit time delay will grow considerably because many in-memory log sectors need to be flushed. Besides, due to the premature force-out of in-memory log sectors, the flash memory blocks will be consumed and erased more frequently, which will in turn add to the overall cost of transaction processing and shorten the life space of flash memory drives.

To address this limitation, we propose an alternative to the three phase commit policy. In order to smooth out potentially spiky commit time cost, the alternative policy attempts to be more truthful to the notion of *no force* by reinstating the conventional system-wide redo logging in addition to the in-page logging, as shown in Figure 3. For the reason, we call this new policy a *redundant logging*.

Under the redundant logging policy, there will be three type of logging activities that occur in the system log during the normal online processing.

A: When a transaction updates a data object, a redo log record is added to the system log (tail) in addition to adding one to the in-memory log sector. (See the top half of Figure 3.)

B: When an in-memory log sector is flushed to flash memory, a `page-check` log record is added to the system log (tail).

C: When a transaction T commits, an `entrust` log record is added to the in-memory log sector for each data page updated by T, and a `commit` log record for T is added to the system log. Then, the system log tail including the commit record is flushed to a stable storage but *without* forcing out any in-memory log sector to flash memory. (See the bottom half of Figure 3.)

The `page-check` is a new type of log records that helps reconstruct in-memory log sectors at the system restart time. As will be described in detail in Section V, a page-check log record plays a role similar to *incremental checkpointing* [15] on per-page basis. It helps avoid repeating unnecessary I/O operations at the system restart, particularly for hot data pages, because their in-memory log sectors tend to be filled up and flushed to flash memory blocks frequently.

A page-check log record contains the id of a data page whose in-memory log sector is being flushed and a window of timestamps of all log records stored in the log sector. The timestamp window is not essential for recovery but added to the page-check log record as an additional fail-safe measure for matching an in-memory log sector with its page-check log record. Note that whether an update log record is considered committed or not is still determined the same way, by the two conditions given in Section IV-B.

If the system crashes after a transaction T commits, the only things that are guaranteed to survive the crash are the log records written to the system log. Since T's in-memory log sectors are not forced out when T commits, some of T's redo log records created by *TIPL* may be lost. This implies that adoption of the redundant logging policy necessitates the redo recovery at the system restart.

V. System Restart

When a database system restarts from a failure, the system executes typically a multi-phase recovery procedure in order to ensure the atomicity and durability of transactions. In general, it must redo the effects of committed transactions that have not reached the persistent storage before the crash, and undo the effects of incomplete transactions that have already reached the persistent storage before the crash. However, the *TIPL* scheme can make the restart procedure a great deal simpler than the conventional recovery systems. This section describes how a database system can restart *instantly* after a crash or recover by an efficient *redo-only* recovery procedure.

A. Instant Restart

As described in Section IV-A, except for transactions aborted during normal database processing, *TIPL* eliminates the need of rolling back any changes made by incomplete transactions. Consequently, the system restart procedure can dispense with the undo recovery altogether.

If the three phase commit policy presented in Section IV-B is adopted for committing transactions, the system restart procedure will be simplified even further. A transaction is considered committed only when a commit log record is written to the system log. For any transaction T that has committed following the three phase commit, T must have saved all of its changes in a form of redo log records flushed from in-memory log sectors to flash memory log sectors along with other log records such as entrust log records, before the system crashes.

Some or all of the T's redo log records may have been merged to data pages in flash memory, if read consistent merge operations had been invoked by the *TIPL* storage manager during the time period between the T's commit time and the time of system crash. Irrespective of that, however, it is still guaranteed that no change made by T is lost once T is committed under this policy. When the database server restarts after the crash, all the T's remaining redo log records are preserved, will survive any future system crash, and will be eventually merged to data pages during normal database processing.

The implication of this is indeed significant. Essentially, the three phase commit policy enables the database system to recover from a crash without explicit redo recovery. Since *TIPL* does away with the undo recovery as is mentioned above, with the three phase commit policy adopted for committing transactions, a database system can recover from a crash without any explicit or separate phases for undo or redo recovery. It can resume normal database processing instantly from the very moment when it restarts from a crash.

B. Redo-Only Recovery

If the redundant logging policy is adopted instead of the three phase commit, then redo recovery must be performed to recover a system from a failure. This is because redo log records are not forced out when a transaction commits under this policy. Some of the redo log records may have remained in in-memory log sectors and have been lost at the time of a system crash. Since the redo log records stored in in-memory log sectors are the only information that can be lost at a system crash, the goal of this *redo-only* recovery is to restore the in-memory log sectors up to the state at the time of crash.

The redundant logging adds a redo log record redundantly to the system log for each change made by a transaction. The system log is maintained in a stable storage, and all the redo log records in the system log will survive a system failure (possibly except for a few log records kept in the log tail). Thus, the in-memory log sectors can be restored from the log records saved in the system log by following a conventional multi-phase recovery procedure such as ARIES [12]. In fact, at a high level, the redo-only recovery procedure proposed in this section is quite similar to the first two phases of the ARIES recovery algorithm. We assume that a transaction table and a dirty page table are maintained during normal database processing and they are force-written to the system log by indirect checkpointing.

The first analysis phase of the proposed redo-only recovery is identical to that of the ARIES. It starts the analysis phase by fetching the transaction table and the dirty page table from the latest checkpointing record stored in the system log. It then scans the rest of the log records in the forward direction to restore the transaction table and the dirty page table up to the current state as of the time of a crash. Then, at the end of the analysis phase, the transaction table contains a list of all transactions active (*i.e.*, either committed or incomplete) at the time of the crash, and the dirty page table contains the information enough to determine the scope of redo recovery (*i.e.*, the location of a redo log record in the system log where the redo recovery must start) [12].

Then, in the redo phase, the history is repeated by re-executing all the log records in the system log including redo, commit, abort, transaction begin, transaction end, and `page-check` log records. A clear distinction from the ARIES recovery lies in the way redo log records are processed and page-check log records are utilized to speed up the redo recovery process. When a redo log record l is encountered, *TIPL* add l to its corresponding in-memory log sector, which will be allocated in the buffer pool if it has not been created yet. Note that this redo action does not involve fetching any data page from the database.

If a page-check log record c is encountered, *TIPL* repeats history simply by discarding the in-memory log sector for c and all its redo log records. This is because we can determine that the in-memory log sector for c has already been forced-out to flash memory from the fact that c is found in the system log.[1] Again this redo action does not involve fetching any data page from the database.

Towards the end of the redo phase, the database buffer pool may be left with in-memory log sectors that contain log records. These log records are yet to be flushed to flash memory, because there is no more matching page-check log record left in the system log. The in-memory log sectors containing the log records are *orphaned* in the sense that they are not accompanied by corresponding data pages in the database buffer pool. This abnormal situation happens, because data pages are never fetched from database during the redo-only recovery.

In order to restore the database system to a state consistent with all committed transactions in the transaction table, *TIPL* flushes all the orphaned in-memory log sectors to the corresponding log sectors in flash memory. Each time an orphaned in-memory log sector is flushed successfully, *TIPL* adds a

[1] Some of the redo log records by an incomplete transaction may have been flushed together to flash memory log sectors before the crash. Those incomplete log records will be eventually eliminated by *TIPL* read consistent merge operations.

page-check log record to the system log. Finally, when all the orphaned in-memory log sectors are flushed successfully, the system becomes ready to resume normal operations and accept new transactions.

Although it does not allow instant restart, the redo-only recovery supports fast recovery from a failure, because it performs the minimum I/O operations required for redo recovery. As for the I/O operations, the redo-only recovery does not read the system log records more than a conventional multi-phase recovery algorithm would do, does not fetch any data page from the database, and does not force out any in-memory log sector more than once by utilizing the page-check log records.

Crash during Recovery: For completeness, consider what would happen if a system crashes again while its recovery procedure is in progress. If it happens during the analysis phase, all the work done so far will be lost but there will be no effect on the state of database. Thus, when the system restarts again, the analysis phase will be repeated with the same IPL log, the same system log and the same database.

If the system crashes during the redo phase, some of the orphaned in-memory log sectors constructed during the redo phase may survive the crash if they have been flushed to flash memory and their page-check log records have been successfully written to the system log. When the system restarts again, the analysis and redo phases will be repeated but the amount of in-memory log sectors to be written during the redo phase will be less, because the in-memory log sectors that were flushed in the first redo will not be redone a second time due to the page-check log records.

VI. Performance Evaluation

In order to understand the performance implications of *TIPL*, we evaluate the time and space overhead incurred by read consistent merge operations using a workload trace obtained from a PostgreSQL server that supports multiversion concurrency control. We also evaluate the performance implications of *TIPL* with respect to the proposed recovery schemes.

Under the previous IPL scheme, log records are blindly applied to their data pages when a block containing them is merged. In contrast, *TIPL* merges log records selectively and carries over the rest in order to support both multiversion read consistency and fast recovery. Thus, log records being carried over are the only cause of additional overhead by *TIPL* over the original IPL. For this reason, we will use the average amount of carry-over log records as one of the key performance metrics.

A. Settings for TPC-C Benchmark

In order to understand the performance implications of read consistent merge operations, it is essential to measure how often and how much log records are carried over to a new block without being merged to data pages.

This in turn requires keeping track of transactions that read various versions of data objects. Since some of those read requests may be served by data objects cached in memory, not all the *logical* read operations can be captured by a database server logging process or even by a kernel level I/O tracing tool. We modified the source codes of the PostgreSQL server to obtain a trace of all I/O operations including those logical reads from a TPC-C benchmark.

The TPC-C database was populated with 10 warehouses of about 1.5GB data, and the benchmark was run on a Linux system (kernel 2.6.28) with an AMD dual-core 2.7GHz processor and 3GB RAM. The buffer cache of the PostgreSQL server was set to 50MB.

We implemented an event-driven simulator that models the read consistent merge of *TIPL* as described in Section III-B. When a read consistent merge is invoked for a flash memory block, the simulator determines whether merge can be performed immediately for individual log records in the block for read consistency by checking the timestamps of the log records and relevant transactions. When there is any log record that needs to be carried over, the simulator determines the amount of additional log space required for the log record.

B. Multi-version Read Consistency

Using the simulator and the trace described above, we observed how often log records are carried over for read consistency and measured the average volume of log data to be carried over. Among the 12,240 flash memory blocks (128KB each) used to store the TPC-C database, 10,299 blocks were updated during the benchmark and a read consistent merge was requested at least once for 6,529 blocks of those updated blocks. A total of 37,230 read consistent merges were requested, and 32,153 (or 86%) of them did not involve any log record to be carried over. In other words, *TIPL* did not incur any additional overhead (both in time and space) to maintain read consistency for about 86% of merge operations. For the rest of merges (*i.e.*, 14% of all merges), the average amount of log records to be carried over was about 2800 Bytes (*i.e.*, less than six sectors) per merge.

Overall, the amount of overhead for the read consistent merges was quite nominal at approximately 400 Bytes (or smaller than a sector) per merge each requested for a flash memory block of 128 KB.

Multiversion read consistency in general imposes a substantial amount of overhead on a database system in both space (for maintaining multiple versions) and time (for reconstructing relevant versions). In a traditional database server, a version reconstruction, as described in Section III, requires many random accesses for pages scattered across several rollback segments. In contrast, *TIPL* can reconstruct a version quickly by maintaining multiple versions concisely in a set of physiological log records and keeping them co-located with corresponding data objects. Considering the low cost of version reconstruction by *TIPL*, the space overhead of .5 KB per merge each requested for a 128 KB flash memory block observed above in the TPC-C benchmark appears to be perfectly acceptable.

C. Fast Recovery

As is described in Section IV-A, *TIPL* ensures that uncommitted log records are not merged to data pages. In order to evaluate its additional overhead, the simulator measures the amount of uncommitted log records that need to be carried over.

Due to a high degree of transaction concurrency, there existed at least one uncommitted log record at almost every moment of time when a merge was requested. For the reason, unlike the case of read consistency, log records needed to be carried over more often for the sake of undo-free recovery of *TIPL* for the same trace described above. However, the average volume of log records to carry over was still quite low at 200 Bytes per merge. This implies that *TIPL* can support both read consistency and undo-free recovery with additional overhead no more than 600 Bytes per merge (again, each requested for a 128 KB flash memory block) to deal with carrying over log records.

The three phase commit policy aims at eliminating the need of redo recovery. For the reason, the three phase commit policy is somewhat similar to the force commit policy in that changes made by a committing transaction are flushed to database. However, the absolute amount of writes required by the three phase commit is expected to be much smaller than the force commit policy would require, because *TIPL* writes only the physiological log records of the changes without writing the updated pages themselves. The reduction in the absolute amount of writes will be significant for OLTP workloads where most data pages are updated only once before propagation. Given the low latency of flash memory, the performance gain obtained by reduced amount of writes will be magnified in flash memory database systems.

In the TPC-C benchmark, we observed that the average volume of log records each transaction produced was approximately 3.4 KB in total or 140 Bytes per page. Given the sizes of an `entrust` log record and a `page-check` log record are no more than 8 Bytes and 16 Bytes, respectively, the space and I/O overhead required for the three-phase commit and redundant logging policies are considered negligible.

VII. Related Works

Recent advances in flash memory SSD technology have increased the I/O bandwidth and throughput of storage devices based on flash memory significantly, and it has been demonstrated that this new storage medium can be used to improve the throughput of large scale OLTP systems by orders of magnitude with much less energy consumption [21], [1]. This impressive development has been realized by various research and development endeavors for better hardware and software such as multi-channel architecture, over-provisioned capacity, large DRAM buffer, flash translation layers, wear leveling algorithms, and so forth. In particular, flash translation layers play a significant role in overcoming the limitations of flash memory such as erase-before-write and the limited number of erase cycles [22], [14], [4].

Nonetheless, if flash memory SSDs are used as a simple replacement of disk drives, a database system may not be able to fully exploit the advantages of flash memory nor fully overcome its disadvantages. Rather, as is demonstrated in previous work [7], [9], some of the key components of database server design may well be revisited so that they become *aware* of the distinct characteristics of flash memory. The *TIPL* scheme presented in this paper has been inspired by the same spirit and developed as a flash-aware design for transaction management.

Not much work has been reported in the literature about transactional support for flash memory based storage and database systems so far. In this section, we briefly review research results related to this work in the area of transactional file system, multiversion-based recovery, and more traditional recovery techniques.

Transactional Flash File Systems: There have been a few recent attempts to provide transactional support for file systems, for example, *transactional flash* [23] and *Light-weight Time shift FTL (LTFTL)* [24]. In order to support various useful features such as atomic writes, rollback, undelete, and time travel operations in a file system, they exploit the existence of multiple versions of a page, which are produced by out-of-place updates in flash memory, much like what we do in *TIPL*. In addition, unlike pure FTL-based approach, they assume more functional interfaces other than the usual read and write in order to provide tighter interaction between the file system and flash memory. Unlike *TIPL*, however, these approaches write a whole page as a unit of update propagation instead of writing change logs, and thus a force commit policy is mandatory for them to ensure atomicity and consistency. The main drawback of the approaches is the storage overhead for maintaining old version pages and the overhead of garbage collection (performed in the background) to clean up old version pages.

Multiversion Based Database Recovery: Several multiversion based database recovery techniques had been proposed in the early 1980s such as the *atomic actions* by Reed [18] and its variants [25], [26]. Some of the recent multiversion based snapshot isolation approaches[13], [16] can be traced back to these pieces of work. The existence of multiple versions makes it easier to implement multiversion concurrency control and guarantee the atomicity of transactions.

Like the transactional flash file systems, the multiversion based recovery approaches use a whole page as a unit of version and adopt a force policy for commit protocol. In addition, these approaches are often criticized for its coarse lock granularity, space overhead from multiple versions, and write bandwidth [27]. This criticism is also shared by the transactional flash file systems mentioned above. It is interesting to note that the transactional flash file systems and the multiversion based database recovery techniques attempt to achieve a similar goal at different layers of the system hierarchy.

Other Recovery Techniques: Besides the popular ARIES-style database recovery, several important recovery techniques have been proposed. We review two of them that are closely related to *TIPL*.

The shadow page mechanism has been proposed as a straightforward way of achieving database recovery [19]. The main advantage is that neither undo nor redo recovery is necessary under this approach. To the best of our knowledge, there is at least one commercial database product marketed for embedded systems with flash memory, called Polyhedra FlashLite [28], that uses this approach. However, the shadow paging mechanism has not been widely adopted by major database systems, because of its limited scalability due to garbage collection overhead, storage space overhead, heavy fragmentation of a table object, and inflexible concurrency control [29]. The same criticism will be still valid even when the shadow paging is used for flash memory.

There is another recovery strategy that has been proposed based on *undo at read time* approach [30]. Under this recovery method, undo is not done in the recovery phase, but is done at read time after restart. On fetching a data item, it is checked whether the data item is created by a successfully committed transaction. If not, it is ignored and then its previous version is retrieved. The *no undo* recovery by *TIPL* is quite similar to this approach. To some extent, this recovery method similar to the deferred rollback approaches [31], [20].

VIII. Conclusion

This paper proposes a novel scheme called *transactional IPL* (*TIPL*), which is built on the in-page logging (IPL) scheme as an underlying platform for buffer and storage management. *TIPL* makes dual use of log records for multiversion read consistency and recovery as well as write performance optimization for flash memory based database systems. The dual use of log records allows us to explore a new design space for flash-aware transaction support, and enables *TIPL* to achieve low-cost transactional support in a way drastically different from conventional disk-based transaction systems.

By eliminating the need of write-ahead logging for undo recovery, *TIPL* reduces the runtime overhead of transactional support during normal database processing. Besides, *TIPL* further reduces the runtime overhead required for multiversion read consistency by reconstructing versions efficiently and by providing optimization techniques for long running transactions. When a database system restarts from a failure, *TIPL* can shorten the recovery time significantly. By adopting the three phase commit or the redundant logging method, *TIPL* can recover the system instantly or by the redo-only recovery procedure from the failure.

Acknowledgment

The authors thank Mr. Woon-Hak Kang for assisting with the experiments.

References

[1] S.-W. Lee, B. Moon, and C. Park, “Advances in Flash Memory SSD Technology for Enterprise Database Applications,” in *Proceedings of ACM SIGMOD*, 2009.

[2] Oracle Corp., “A Technical Overview of the Sun Oracle Exadata Storage Server and Database Machine,” White Paper, Sep 2009.

[3] D. Reinsel and J. Janukowicz, “Datacenter SSDs: Solid Footing for Growth,” IDC Corporation, Tech. Rep., Jan. 2008.

[4] S.-W. Lee *et al.*, “A Log Buffer-based Flash Translation Layer using Fully-Associative Sector Translation,” *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 3, pp. 1–27, 2007.

[5] S. Nath and A. Kansal, “FlashDB: Dynamic Self-Tuning Database for NAND Flash,” in *Proceedings of IPSN*, Cambridge, MA, Apr. 2007.

[6] G.-J. Na, S.-W. Lee, and B. Moon, “Dynamic In-Page Logging for B^+-tree Index,” *IEEE Transactions on Knowledge and Data Engineering*, (To appear).

[7] S.-W. Lee and B. Moon, “Design of Flash-Based DBMS: An In-Page Logging Approach,” in *Proceedings of ACM SIGMOD*, Jun. 2007.

[8] Y. Ou, T. Härder, and P. Jin, “CFDC - A Flash-Aware Replacement Policy for Database Buffer Management,” in *DaMoN*, Providence, RI, U.S.A., Jun. 2009.

[9] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe, “Query Processing Techniques for Solid State Drives,” in *Proceedings of ACM SIGMOD*, Providence, RI, U.S.A., Jun. 2009.

[10] S. Chen, “FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance,” in *Proceedings of ACM SIGMOD*, Jun. 2009.

[11] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, “A Case for Flash Memory SSD in Enterprise Database Applications,” in *Proceedings of ACM SIGMOD*, 2008.

[12] 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 Transactions on Database Systems*, vol. 17, no. 1, pp. 94–162, 1992.

[13] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O’Neil, and P. E. O’Neil, “A Critique of ANSI SQL Isolation Levels,” in *Proceedings of ACM SIGMOD*, 1995, pp. 1–10.

[14] Intel Corporation, “Understanding the Flash Translation Layer (FTL) Specification,” Application Note AP-684, Dec 1998.

[15] P. Bernstein and E. Newcomer, *Principles of Transaction Processing*, 2nd ed. Morgan Kaufmann, 2009.

[16] K. Jacobs, “Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7,” Oracle White Paper, July 1995.

[17] S. Adams, “The Mysteries of DBWR Tuning,” in *Oracle Open World*, 1997, http://www.ixora.com.au/tips/mystery.doc.

[18] D. Reed, “Implementing Atomic Actions on Decentralized Data,” *ACM Transactions on Computer Systems*, vol. 1, pp. 3–23, 1983.

[19] R. A. Lorie, “Physical Integrity in a Large Segmented Database,” *ACM Transactions on Database Systems*, vol. 2, no. 1, pp. 91–104, 1977.

[20] T. Lahiri, A. Ganesh, R. Weiss, and A. Joshi, “Fast-start: Quick fault recovery in oracle,” in *Proceedings of ACM SIGMOD*, 2001.

[21] Intel Corporation, “OLTP Performance Comparison: Solid-State Drives vs. Hard Disk Drives,” Test Report, Jan 2009.

[22] A. Gupta, Y. Kim, and B. Urgaonkar, “DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings,” in *Proceeding of ASPOLOS*, 2009, pp. 229–240.

[23] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, “Transactional Flash,” in *8th USENIX Symposium on Operating Systems Design and Implementation(OSDI 2008)*, 2008, pp. 147–160.

[24] K. Sun, S. Baek, J. Choi, D. Lee, S. H. Noh, and S. L. Min, “LTFTL: Lightweight Time-Shift Flash Translation Layer for Flash Memory Based Embedded Storage,” in *Proceedings of the 8th ACM international conference on Embedded software*, 2008, pp. 51–58.

[25] P. M. Bober and M. J. Carey, “On Mixing Queries and Transactions via Multiversion Locking,” in *Proceedings of ICDE*, 1992, pp. 535–545.

[26] A. Chan, S. Fox, W.-T. K. Lin, A. Nori, and D. R. Ries, “The Implementation of an Integrated Concurrency Control and Recovery Scheme,” in *Proceedings of ACM SIGMOD*, 1982, pp. 184–191.

[27] J. Gray, “Notes on Data Base Operating Systems,” in *Advanced Course: Operating Systems*, 1978, pp. 393–481.

[28] E. Corporation, “Enea Polyhedra Database Systems - Data Sheet,” 2009, http://www.enea.com.

[29] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[30] R. Rappaport, “File Structure Design to Facilitate On-Line Instantaneous Updating,” in *Proceedings of ACM SIGMOD*, 1975, pp. 1–14.

[31] C. Mohan, “A Cost-Effective Method for Providing Improved Data Availability During DBMS Restart Recovery After a Failure,” in *Proceedings of VLDB*, 1993, pp. 368–379.