High Performance Database Logging using Storage Class Memory

Ru Fang[#1], Hui-I Hsiao[#2], Bin He[#1], C. Mohan[#2], Yun Wang[*3]

[#]*IBM Almaden Research Center*
650 Harry Road, San Jose, CA, USA
[1] {rufang, binhe}@us.ibm.com
[2] {hhsiao, mohan}@almaden.ibm.com

[*]*IBM China Research Laboratory*
Building 19, Zhongguancun Software Park, 8 Dongbeiwang West Road, Haidian District, Beijing, China, 100193
[3] wangyunw@cn.ibm.com

***Abstract*— Storage class memory (SCM), a new generation of memory technology, offers non-volatility, high-speed, and byte-addressability, which combines the best properties of current hard disk drives (HDD) and main memory. With these extraordinary features, current systems and software stacks need to be redesigned to get significantly improved performance by eliminating disk input/output (I/O) barriers; and simpler system designs by avoiding complicated data format transformations. In current DBMSs, logging and recovery are the most important components to enforce the atomicity and durability of a database. Traditionally, database systems rely on disks for logging transaction actions and log records are forced to disks when a transaction commits. Because of the slow disk I/O speed, logging becomes one of the major bottlenecks for a DBMS.**

Exploiting SCM as a persistent memory for transaction logging can significantly reduce logging overhead. In this paper, we present the detailed design of an SCM-based approach for DBMSs logging, which achieves high performance by simplified system design and better concurrency support. We also discuss solutions to tackle several major issues arising during system recovery, including hole detection, partial write detection, and any-point failure recovery. This new logging approach is used to replace the traditional disk based logging approach in DBMSs. To analyze the performance characteristics of our SCM-based logging approach, we implement the prototype on IBM SolidDB. In common circumstances, our experimental results show that the new SCM-based logging approach provides as much as 7 times throughput improvement over disk-based logging in the Telecommunication Application Transaction Processing (TATP) benchmark.

I. INTRODUCTION

In recent years, research and development efforts are underway worldwide on novel non-volatile memory technologies, collectively called storage class memory (*SCM)*. SCM blends the best properties of memory and hard disk drive (HDD), which include high performance, byte-addressable (similar to Dynamic Random Access Memory (DRAM)), data persistence, and more energy efficient (similar to HDD).

Phase Change Memory (PCM) is one of the technologies competing to become the SCM of choice. Studies [1][2] show that the access speed of PCM is two to four orders of magnitude faster than flash (SSD) or disk for typical file system I/Os, and within an order of magnitude of DRAM. Compared to flash, PCM has a much longer write endurance time. Currently, a flash bit will fail after being written 10^4–10^5 times, while PCM bit is designed to sustain 10^8–10^{12} writes. More importantly, flash devices only allow block-based access, which results in latency and system implementation complexity. PCM is byte-addressable; accessing PCM is similar to accessing DRAM. So the complex data format transformation can be eliminated.

Moreover, PCM has higher chip density than DRAM, thus creating opportunities for building affordable machines with large amount of main memory in the future, which will reduce or eliminate the need of frequent random I/O's to page-in and page-out data. I/O performance, both latency and bandwidth, has long been the key bottleneck of data management system and applications. Due to the big difference in access speed between memory and disk, "hiding" I/O during application execution has become a very important tactic in software system design. Researchers and practitioners have been putting significant effort in trying to reduce or hide I/O performance impact. This, unfortunately, makes data management system design and application building rather complicated.

PCM is byte-addressable, fast and non-volatile. It provides system software designers a new opportunity to rethink system architecture and data management algorithms. The objective of our work is to explore how to introduce SCM in the design of a DBMS to overcome major I/O bottlenecks and thereby building a more effective and streamlined DBMSs.

In current DBMSs, as shown in Fig. 1, transactions write data and log records in DRAM buffers first; then move them to a stable storage, such as HDD, to guarantee the durability as needed or upon transactions commit. The log records must be flushed to log files on disk before any associated data changes are made to the data files (write-ahead-logging) [3]. If a system failure occurs, such as hardware failure or system power interruption, the recovery process is started when the system restarts, to read the log files and apply log records to restore the database to a consistent state.

Writing and flushing log records to log files is an important step that cannot be bypassed during database transaction

processing. However, because of the disk I/O barrier, significant overhead is incurred during writing and flushing log records, which often becomes the key inhibitor for high performance transaction processing. To address this problem, most modern DBMSs use a system-throughput-oriented approach called "group committing" [4] which tries to reduce the number of disk writes by grouping log writes of multiple committing transactions into one single disk write. However, from a single transaction perspective, group committing degrades average response time. It also leads to increased lock/latch contentions.

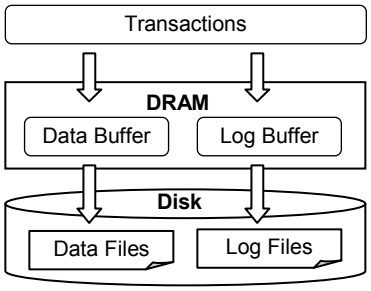


Fig. 1. Simple View of Current DBMS

In the initial stage, the cost per bit for SCM technologies will still be higher than that for HDD or flash. Overtime, the price will decrease dramatically and may—by the middle of the next decade—approach that of enterprise disks [1]. Currently, the density of SCM chip is not as high as that of HDD or flash. PCM prototype chip from Samsung holds 512Mb. It is estimated that by 2020, the capacity of a single PCM chip can reach 1TB.

Given the current relatively high cost of PCM, we are seeking for the best strategy to leverage its advantages in a practical and economic way. In this paper, we try to use the PCM to address one of the most critical bottlenecks in transaction processing: logging. Our goal is to reduce and/or eliminate disk I/O impacts on throughputs and response times of transactions. Specifically, we design an SCM-aware logging algorithm and directly write log records to SCM-based logging space. Optionally, log files are copied from SCM to HDD asynchronously for long-term preservation. The new DBMS architecture is shown in Fig. 2.

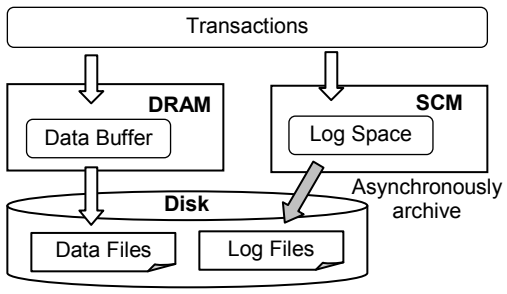


Fig. 2. Overview of the DBMSs with SCM-based Logging

Compared to traditional DBMSs logging approach, there are two major advantages of this new SCM-based logging approach:

- *Simplified log architecture*

In this new SCM-based logging approach, transactions allocate chunks from SCM log space directly, and then write log records into these SCM chunks without first caching in DRAM and then flushing to persistent storage device. Since there are no log buffers in DRAM and log files on disk, complex buffer management and slow disk I/Os can be avoided. The logging component design and implementation are extremely simplified. Since the need for writing log data to log files on disk is removed from transaction processing, the disk I/O bottleneck is eliminated from logging. Complicated data transformation can also be avoided. All these simplifications bring significant improvement on DBMS performance.

- *Better concurrency support*

In traditional approaches for DBMS logging, when a transaction wants to make a log record persistent, it has to wait for the log records in font of this log record to be filled out, to compromise block-based disk I/Os. This design becomes the major concurrency bottleneck in DBMS logging design.

In contrast, because of the byte-addressability of SCM, our new logging design does not have such a constraint. When a transaction wants to write a log record, it does not need to wait for other transactions to fill out log data in front of this log record. Unlike the much more complicated latching design in traditional logging, our design uses an extremely simplified latching method to minimize the concurrency overhead and improve transaction latency.

On the other hand, although the intriguing features of SCM inspire us with a new logging design, there also are challenges that we must address to ensure a correct system recovery. In particular, there are three major problems we need to solve:

- *Hole detection*

If system crashes when a transaction just allocates chunks in SCM but has not written anything yet, there can be empty log records (holes) in the SCM log space. Holes detection and processing becomes a major challenge for system recovery during restart.

- *Partial write detection*

If system crashes when a transaction has not fully finished writing log data into SCM space, during system recovery process, this partially written log record should be identified and processed appropriately.

- *Any point crash recovery*

Byte-addressable persistency is an attractive feature of SCM, but it can also be a headache in system design, because any write on SCM is persistent right away. The design thus must be able to handle any point crash during SCM operations. It can be data writes, checkpoint writes, metadata writes, and space allocation. The data structures and algorithms need to be careful designed to deal with any point crash.

In this paper, we focus on introducing the new design of the SCM-based database logging approach and solve the above problems. We have implemented our design on IBM SolidDB [5] and ran several experiments on the TATP benchmark [6]. The contributions of this paper include:

- To our knowledge, we are the first effort to introduce SCM into database design. In particular, we focus on logging: A critical bottleneck for today's transaction processing.
- We discuss the necessary hardware support and design a new SCM-based approach for DBMSs logging with simplified log architecture, better concurrency and latency.
- We develop efficient SCM based log management and system recovery algorithms to solve three major problems: hole detection, partial write detection, and any point crash recovery.
- We implement a prototype to demonstrate the effectiveness of our design. In the experiments, the new SCM-based logging approach can provide 7 times performance improvement over disk-based logging on the TATP benchmark in common circumstances. For some special cases, for example, when all the transactions are write transactions, the performance improvement can be as much as 29 times.

The rest of this paper is organized as follows: We begin with a discussion of related work in Section 2. In Section 3, we introduce the details of PCM device and discuss the necessary hardware support. In Section 4, we give an overview of the system architecture we proposed, including the OS system call design for SCM and the new logging approach in DBMSs. In Section 4, we present the detailed design of the new logging approach, including SCM log data structure, and log management procedures and algorithms. In Section 6, we introduce the recovery algorithm based on our new SCM-based logging approach. We then sketch our prototype on the IBM SolidDB product, and present our experiments on the TATP benchmark together with the results analysis in Section 7. We conclude our work in Section 8.

II. Related Work

Nowadays, the price of DRAM has come down to a affordable range. Tens or hundreds of gigabyte DRAM in an enterprise computer system is not uncommon. This triggers a renewed focus on studying in-memory database technologies, which can load most if not all of database data into main memory and then manipulate data in memory. Compared to disk-based databases, in-memory databases have faster and more predictable performance, which fit well for applications that require fast and reliable response time, such as telecommunications application.

However, in-memory databases still need to guarantee data durability to support system recovery. Currently, there are two approaches. The first one is to asynchronously write dirty data to local disks or flash drives, as is done by IBM SolidDB, Oracle TimesTen [7] and Altibase [8]. To improve system performance, the write interval should be configured as a relatively long time. However, this has an associated risk. If the system were to crash before the data in memory is backed up to a persistent storage, recovering will take a long time.

The other approach is to leverage high-speed networks and synchronously replicate data to memories of remote machines. If one machine were broken, then it can obtain the remote copy and recover the local system. This approach is common in data center or cloud environment, such as RamCloud [11], Dynamo[11] and Cassandra[12]. However, this approach would at least double or triple the cost of memory, which is a dominant factor in the overall system cost. It also requires the construction of reliable high-speed networks and even so still suffers network failure problems. Furthermore, replication in memory depends on a reliable power source, so that power outages do not take down all of the replicas simultaneously. All these issues make crash recovery considerably more expensive.

Recently with the development of Solid State Drive (SSD), NAND-based flash memory is considered to have tremendous potential as a new storage medium that can replace magnetic disks. The research has been started to explore redesigning database system on NAND-based flash memory [18][20] Journal file systems technology is introduced to leverage the fast random write advantage and avoid erase-before-write limitation of flash memory. However, the current flash-based database still uses flash memory as hard disk drive. Data and log are written into flash memory through traditional I/O interface and the basic page-based logging structure is not changed. Complicated I/O protocol and data transformation still can not be avoided.

In the SCM-based logging approach, data durability can be guaranteed by using SCM at the local machine synchronously during transaction processing at high performance. Hence, the SCM-based logging design can be applied for not only the disk-based or flash-based DBMSs, but also the above in-memory data stores, in which data durability and high transaction performance can be achieved at the same time.

III. PCM and Hardware Support

PCM, as the new class of memory technology of SCM, employs the reversible phase change in materials to store information, and provides non-volatility and byte-addressability. Unlike DRAM,, PCM is made from a chalcogenide glass, a material that can be switched between two "phases", crystalline and amorphous, by heating it to 650 °C and then cooling it either slowly or rapidly. These phases have different resistivity, which are used to represent 0 and 1.

The access latency of PCM is at the level of hundreds of nanoseconds, which is only 2–5 times slower than DRAM. To take advantage of this access speed and byte-addressability feature, it is proposed in [13] that PCM can be placed directly on the memory bus, side-by-side with DRAM. The 64-bit physical address space will then be divided between volatile and non-volatile memories, so the CPU can directly address PCM with common loads and stores.

To provide safety and consistency, software must reason about when and in what order the writes to PCM are made

durable. Existing cache hierarchies and memory controller may reorder writes to improve performance. As with the problem of ordering, software system also needs to enforce write atomicity with respect to power failure. Thus, if a write to the persistent memory is interrupted by a power failure, the PCM could be left in an intermediate state, violating consistency. In [13], it proposes that the hardware could provide two primitives, *atomic 8-byte write* and *epoch barrier*, to guarantee the atomicity and ordering of SCM writing.

The atomic 8-byte write requires that in case of power outage, the PCM hardware has enough remaining power to ensure an 8-byte write in PCM is either fully written or not. That is, partial write will not occur within an 8-byte write. This primitive makes a pointer update an atomic operation on PCM.

The epoch barrier is a command, which allows software developers to explicitly communicate ordering constraints to PCM hardware. When the hardware sees an epoch barrier, it will make sure all the PCM writes before the epoch barrier are written through from caches before the PCM writes after the epoch barrier are executed.

In our design, we use the same two hardware primitives as the basis of our SCM-based logging approach. Also the same as in [12], we assume that in order to prolong the usable life of PCM, proper wear levelling should be provided by the hardware to guarantee all blocks of PCM are approximately written at the same frequency.

IV. System Overview

From a high-level point of view, the SCM-based logging approach, as one of the fundamental functions in DBMS, manages log data in the SCM log space and provides logging interface to the other DBMS sub-systems to write log records into this log space. Internally, the logging system invokes operating system (OS) interfaces to allocate and initialize a chunk of persistent memory from the underlying SCM device as the log space to store log records, as shown in Fig. 3.

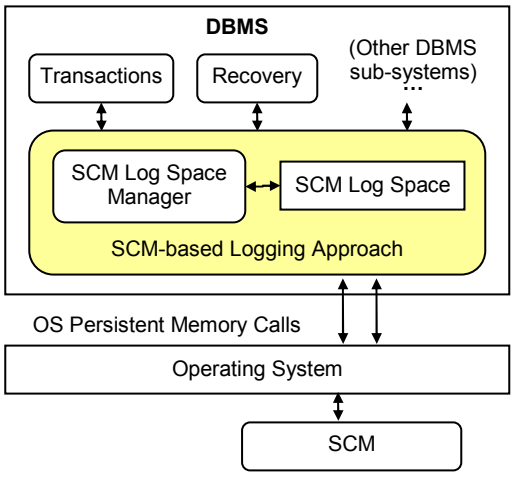


Fig. 3. System Overview

4.1 Proposed OS Interfaces for SCM

Operating System (OS) layer support is necessary for designing SCM-based logging approach. However, since SCM devices are not yet available commercially, there has not been any work on the memory management support in OS for SCM. To make our design and implementation complete, we first studied the current OS memory management functions and compared different memory management interfaces. Then based on the inter-process communication (IPC) shared memory calls formats and process [14], we design the following OS interfaces for SCM:

- *int shpm_get (key_t key, size_t size, int shpmflg)*
- *void* shpm_at (int shpmid, void *shpmaddr, int shpmflg)*
- *int shpm_dt(const void *shpmaddr)*
- *int shpm_ctl(int shpmid, int pm_cmd, struct shpmid_ds *buf)*

As demonstrated in Fig. 4, at runtime, an application first gets a unique key (name) and then invokes *shpm_get()* function, by passing in the key, to create or find a segment in SCM, which returns the unique identifier of this SCM segment. After that, the application attaches this SCM segment to its virtual address space by calling *shpm_at()* function. Other applications can also attach to the same SCM segment by using its unique identifier. Upon successful completion, *shpm_at()* returns the start address of the attached SCM segment. Then, the application can write or read data to this SCM segment in the same it uses main memory.

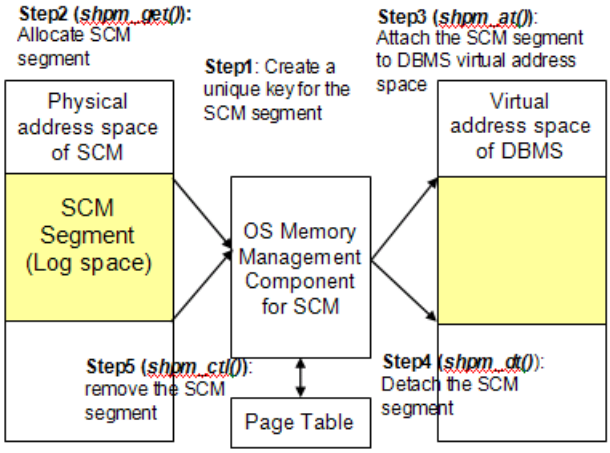


Fig. 4. OS Memory Management Component for SCM

When an application does not need anymore an SCM segment located at a specific start address *shpmaddr*, it invokes *shpm_dt()* function to detach this segment from its virtual address space. Then, this application can not access this SCM segment. But the data is not discarded until this segment is actually removed.

Shpm_ctl() function is used to control and get metadata of a segment. When the creator or administrator wants to remove an SCM segment, this function is called, passing in the segment's unique identifier, to set it as "removable". After all

applications have detached from the SCM segment, OS will actually remove this SCM segment.

Using file system and memory mapped file technology [15], we have implemented a prototype to simulate SCM and the above OS SCM interfaces on Linux. The following design and discussions are based on this OS interface design.

4.2 SCM-based Logging Approach

Our SCM-based logging approach includes two parts, *SCM log space* and *SCM log space manager*, as illustrated in Fig. 3. SCM log space is a segment in SCM, which replaces the current log buffers in main memory and log files on disk to store both log space metadata and transaction logs. SCM log space manager, the management component, is responsible for communicating with OS to initialize and remove an SCM log space; allocate chunks from SCM log space for other DBMS sub-systems to write their log records in and manage the log records in the SCM log space. For disaster recovery purposes, the log data may need to be archived to disk or tape periodically. So the SCM log space manager also supports the archive function which copies log data to other persistent storage asynchronously. The details of these functions and associated data structures are introduced in the following sections.

V. DESIGN OF SCM-BASED LOGGING FOR DBMSS

In this section, we present design details of the SCM-based logging approach for DBMSs, including data structures in SCM log space, functions of SCM log space manager and core processes of this new logging approach during normal processing. Section 6 will discuss the recovery process after system crash.

5.1. Data Structure of SCM Log Space

SCM log space is managed as a linear and continuous space, as shown in Fig. 5.

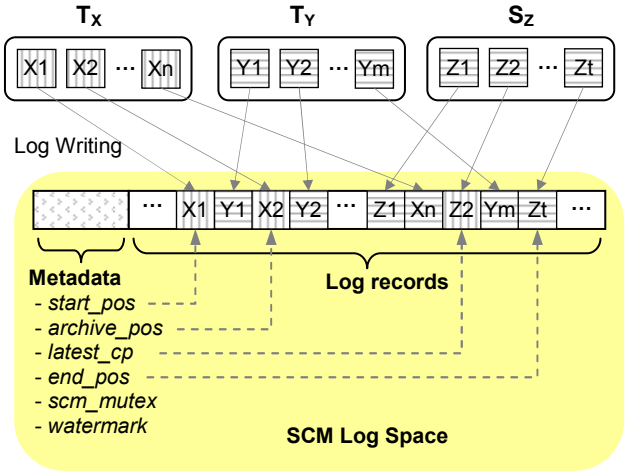


Fig. 5. Data Structure of SCM Log Space

5.1.1 SCM Log Space Metadata

A set of attributes (the metadata) describing the log space and log data are saved in the header of SCM log space, such as start position and end position of log data, the last checkpoint position, latch information, etc. This set of metadata can be retrieved or changed at runtime by invoking the logging interfaces provided by SCM log space manager.

As shown in Fig. 5, *start_pos* and *end_pos* point to the first and last log record in SCM log space, respectively; *archive_pos* indicates the location up to which all the log records have already been archived to other persistent storage. *scm_mutex* is the latch of SCM log space.

Another user configurable parameter is *watermark*. It is used to specify the level (percentage) of log space used before automatically kicking off the log archival process. SCM log space manager periodically checks the size of log data. Once it is larger than *watermark*, part of the log data must be archived to disk, and the archived log data space can then be freed. Any completed log records can be archived to disk.

Every time the DBMS takes a checkpoint, we also save the location of this checkpoint log record by updating the parameter *latest_cp* in the header of SCM log space. Thus, when DBMS recovery process starts, recovery system can easily find the latest checkpoint log record and it needs to process only the checkpoint log records after this log record and apply the changes to the data volumes.

Because of the atomic 8-byte write primitive provided by PCM (as we discussed in Section 3), an update of any of these pointers is done in an atomic way. No partial write will occur for these pointer updates.

5.1.2 Log Record Data Structure

Following the SCM log space metadata is the real log data. As depicted in Fig. 5, each log record is allocated as one chunk in SCM log space. Log records are packed sequentially in the log space and a new log record is always appended immediately after the last log record. During the release process (after log archival, for example), log records are removed from the head of log stream and the released space is reclaimed and later reused during future log writes. In the example shown in Fig. 5, transaction threads, TX and TY, and system thread, SZ, such as a checkpoint thread, write log records, X_i, Y_j, Z_k ($i \in \{1,2,...,n\}$; $j \in \{1,2,...,m\}$; $k \in \{1,2,...,t\}$), directly into this log space, without caching in log buffers or writing to log files.

We follow the ARIES WAL [3] to design the log record data structure. However, we add two more fields at the beginning and the end of a log record separately, as shown in Fig. 6. The first field is the length of this log record. When a transaction inserts a log record, it writes this field at first. The writing of this field is an atomic write because the length of this field is 8 bytes. The second field is the end bit as the end of a log record, which is used to tell if the writing of a log record is finished or not. If the writing is finished, the end bit is 1 otherwise it is 0. With these two fields, the recovery subsystem can determine if is there is any hole in the log space, or if there is any partial write of a log record.

To guarantee the correctness of the recovery process during system crash, it is critical make sure the correct ordering of

writing these fields of a log record. Epoch barriers (as discussed in Section III) are used to ensure such ordering. In Section 5.2, we will discuss in details about ordering of writing and how epoch barriers are used.

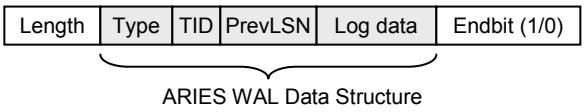


Fig. 6. Data Structure of a Log Record

5.2. SCM Log Space Manager

The core management functions provided by SCM log space Manger include: initializing and removing SCM log space, allocating and releasing chunks in SCM log space, latch management, metadata management and archiving log data to other persistent storage if necessary.

5.2.1 Initialize / Remove SCM Log Space

SCM log space manager communicates with the OS to initialize and remove SCM log space.

When a database is first created and started (opened), Log Space Manager gets the configuration information of SCM log space from a configuration file. The configuration information includes the request size of log space, a suggested virtual start address, etc. Subsequently, SCM log space manager invokes OS system interfaces, *shpm_get()* and *shpm_at()*, to allocate and attach an SCM segment. Based on the design of our OS SCM interfaces, OS returns a unique identifier to the SCM log space manager. SCM log space manager then saves this unique identifier in the configuration file for future restart, After that, SCM log space manager creates the metadata in the header of SCM log space, and set all bits as 0 in the rest of SCM log space.

On a subsequent restart of a database, SCM log space manager reads the identifier of SCM log space for the database from the configuration file. Then using this identifier, it invokes OS SCM interfaces, *shpm_get()* and *shpm_at()* to find and reattach to the SCM log space. After that, DBMS checks if the database was abnormally shutdown previously, for example, due to power failure or system error. If it was, DBMS recovery process is started to recover the database state. The recovery process reads log records directly from the SCM log space instead of log files on disk. Other than that, the recovery algorithm does not change; it follows the current mechanism to bring the database to the latest consistent state. After the recovery process is finished, the metadata in the log space header is also updated. When the initialization and recovery process (if needed) are done, the SCM log space is ready for use.

When a database is shutdown, OS SCM interface, *shpm_dt()*, is invoked by the SCM log space manager to detach this log space. And when a database is dropped, SCM log space manager calls *shpm_ctl()* function to physically remove the SCM log space.

5.2.2 Allocate / Release Chunks in SCM Log Space

When a transaction thread or a system thread needs to write a log record to the log space, SCM log space manager allocates a chunk in SCM log space according to the required size, and returns a pointer to this chunk to the thread. This thread can then write the log record to this chunk. The pseudo codes of this procedure are as follows:

```
Algorithm AllocateLogSpace
input: log size of nbytes

begin
1  if free space < nbytes + 9 then
2    archive log data to obtain enough free spaces
     request latch of allocation
3  allocate nbytes + 9 bytes from current end_pos
4  // The extra 9 bytes are used by the length and
      the endbit fields
5  write the length field to be nbytes + 9
6  write the endbit field to be 0
7  write epoch barrier
8  log_pointer = end_pos + 8
9  end_pos = end_pos + nbytes + 9
10 write epoch barrier
11 release latch
12 return log_pointer
end

Algorithm WriteALogRecord
input: log_pointer
begin
1  memcpy log data starting at log_pointer
2  write epoch barrier
3  update the endbit field to be 1
end
```

First, the SCM log space manager will allocate the space using the `AllocateLogSpace` Algorithm. Suppose a transaction thread or system thread wants to allocate a chunk with size, *nbytes*, in the SCM log space, at first the SCM log space manager checks if the size of free space is larger than *nbytes* + *length field* + *endbit field* (line 1). If it is not true, SCM log space manager tries to archive and remove some log data from SCM log space (line 2). Then, from the current end log position (*end_pos)* SCM log space manager creates the *length* field of the log record, and fill it out with the value *nbytes* + *9* (lines 3-5). The extra 9 bytes are used for the *length* and *endbit* fields. During allocation, SCM log space manager sets the *endbit* as 0, which means the writing of this log record has not been finished yet (line 6). After that, SCM log space manager moves the *end_pos* to the new position, which is *nbytes* + *9* bytes forward from the current position, and return the old *end_pos* + *8* (i.e., the starting address of the log data) to the caller thread (lines 8 – 12).

During the above allocation process, the *length* and *endbit* fields need to be filled out. The advantage is to address the hole and partial write detection problems with the same algorithm, as Section VI will discuss. Otherwise, we need different algorithms to handle these two problems, which brings extra complexities.

With the returned pointer, the transaction use Algorithm `WriteALogRecord` to write a log record. It first uses a common memory copy function, such as *memcpy()*, to write or copy the log record directly to the allocated chunk (line 1). After

writing the log data, the transaction updates the *endbit* field to 1 to indicate the completion of the log writing (line 3).

As Algorithms `AllocateLogSpace` and `WriteALogRecord` show, epoch barriers are used between some PCM writes to ensure the ordering of PCM writes. For instance, we want to make sure the *length* and *endbit* fields are fully initialized before writing real log data (line 7 of Algorithm `AllocateLogSpace`). We also want to ensure the log data is fully written on PCM before updating the *endbit* to 1 (line 2 of Algorithm `WriteALogRecord`).

To guarantee the *end_pos* is updated correctly in SCM log space metadata, we issue an epoch barrier after changing *end_pos* field (line 10 of Algorithm `AllocateLogSpace`). Since all the allocations need to change *end_pos*, it becomes a hot spot. Every request thread has to update this field in SCM and write an epoch barrier. As a result, epoch barriers are frequently used on the same hot field, which may affect the SCM write performance due to massive write through operations. Our optimization strategy is to maintain a DRAM copy of the *end_pos* value and synchronize it to SCM periodically. When a thread requests a chunk from SCM log space, the SCM log space manager uses the DRAM copy of *end_pos* to allocate space for it. We also keep a counter in DRAM to remember the allocated sizes since the last synchronization of *end_pos* field in SCM. When this requested size is equal to or larger than a predefined threshold (specified as *sync_threshold*), synchronization is issued to write the value of *end_pos* to SCM. After synchronization, SCM log space manager checks if the next *sync_threshold* size of space from current *end_pos* is free. If not, then SCM log space manager archives at least *sync_threshold* size of data to other media to guarantee no memory corruption when writing data to SCM Log Space in the future. With this optimization, line 9 of `AllocateLogSpace` can be done in DRAM in most cases, and correspondingly line 10 is unnecessary for those DRAM updates.

If the system crashes when the latest value of *end_pos* has not been written to SCM, then the recovery component scans the log data up to the position of *end_pos* + *sync_threshold* at most. We will introduce the details of the recovery algorithm in the next Section.

When a chunk in the SCM log space has to be removed, such as the SCM log space is full and log data has already been archived to some other persistent storage, SCM log space manager updates the *start*_pos field, and free the log data of request size from the beginning. During this chunk removing cycle, SCM log space manager also cleans the data (set all bits as 0) in this chunk. This guarantees the *length* field and the *endbit* field not to be messed up when this chunk is reused next time.

5.2.3 Latch Management for SCM Log Space

This function is for synchronizing the allocation of the SCM log space. Since the SCM log space is a linear and continuous space, a new log record must be appended following the last one. So, when a transaction thread or a system thread needs to request a chunk of the log space, it first acquires a latch from the SCM log space manager on the tail of the log stream. When the allocation is completed and log space metadata updated, it then releases this latch. The thread can write log data into the chunk allocated while other thread can get the latch for its own chunk allocation.

5.2.4 Update and Retrieve Metadata of SCM Log Space

SCM log space manager is also responsible for updating the metadata when any change occurs in the log space. For example, updating the data size attribute when new log records are added into SCM log space.

Other DBMS sub-systems can retrieve the metadata of SCM log space through the interfaces provided by SCM log space manager. For instance, the recovery sub-system can get the location of the latest checkpoint saved in the SCM log space header, which assists the recovery sub-system to find the location to start scanning the log records and recover the database appropriately.

5.2.5 Archive Log Data for Disaster Recovery

For disaster recovery consideration, the data in SCM log space should be asynchronously saved to disk or tape periodically. So optionally, SCM log space manager archives the log data to some persistent storage at a remote location. When/if disaster occurs, the archived log data can be used to restore the log data and recover the database.

VI. Recovery Based on SCM Logging

In the current DBMS recovery process, the recovery system loads log records from log files on disk to a recovery log buffer in memory. Then it proceeds in three phases: analysis, redo and undo. The analysis phase scans the log from a specific location, e.g., the latest checkpoint record, to identify the committed transactions and in-flight (uncommitted) transactions at the time of system failure. In the Redo phase, the recovery system reapplies logs, from the last checkpoint to the end of log, if the changes have not made on the corresponding data. In the last phase, actions of uncommitted transactions are undone by scanning logs in reverse chronological order from the end of log. Since the recovery log buffer is not likely to be large enough for containing all log records needed for recovery, the recovery system usually loads and processes the log records in batches. Depending on whether it is in redo or undo phase, recovery system will either apply the redo information in a log to the database data or apply the undo information to undone the effect of in-flight transactions. During this process, the loading and scanning of log records can be very time-consuming.

With our new SCM-based logging approach, the log records do not need to be loaded from log files to log buffer. The recovery system only needs to know the address of SCM log space. It can also get the location of the latest checkpoint log record in the SCM log space header. Then, it can directly read the log records in SCM log space from this latest checkpoint log record, and apply the redo and undo functions to restore database state. After recovery, the system makes a checkpoint and updates the *latest_cp* field in SCM log space metadata.

As introduced in Section 1, with SCM-based logging, there are three major challenges to be addressed during the log scanning and analysis pass because of the byte-addressable persistent feature of SCM. In this section, we propose the following scan and analysis algorithm *RestartAnalysis* for SCM log space to solve these problems.

```
Algorithm RestartAnalysis
input: scm_log_start_address, sync_threshold

begin
1  access log metadata from scm_log_start_address
2  current_pos = the log position after the latest
checkpoint log record
3  while current_pos < end_pos + sync_threshold
4    length = read the length field of the log
              record at the current_pos address
5    if length = 0 then
6      break   // the end of the log
7    endbit = read the end bit field of the log
              record using current_pos and length
              information
8    if endbit = 0 then
9      // This is a hole or partially written log,
          skip it
10     current_pos = current_pos + length
11     continue
12   logdata = read the log data at current_pos
13   begin the normal database recovery analysis
     using logdata
end
```

During the restart analysis phase, the recovery system scans the log records from the position of the latest checkpoint log record (*latest_cp* in SCM log space metadata) (lines 1-2). Because we keep a copy of *end_pos* in the DRAM and system may crash when the DRAM copy has not been synchronized to SCM copy, we cannot simply scan the log records until the *end_pos*, because it may not be the latest end position of the log records. Therefore, in Algorithm *RestartAnalysis*, the recovery analysis pass scans the log records until the *end_pos* plus the predefined threshold (*sync_threshold*) at most (line 3). Then, for each log record, the *length* field is checked at first. If the value of the *length* field is 0, which means it is the end position of the log records, and the scanning can be stopped here (lines 4 – 6).

If the value of the *length* field is not 0, then the *endbit* position is calculated using on the value of the *length* field. If the end bit is 0, it means a transaction has allocated this chunk in SCM log space, and the writing of log data has not been started or but finished. This log record is a hole or a partial write log record and it should be bypassed during recovery (lines 7 – 11). Otherwise the writing of log data is finished; and the redo or undo can be continued on this log record (lines 12 – 13).

If the database just finishes taking a checkpoint, and has not created a checkpoint log record, or updated the *latest_cp* field in the SCM log space header, the analysis is started from the position of the next to the last checkpoint log record. The recovery performance may be degraded a little bit but the system correctness can be guaranteed.

VII. PROTOTYPE AND EVALUATION

7.1 Prototype Implementation

We have built a system to simulate PCM using DRAM. Our system also provides a set of OS interfaces for PCM as introduced in Section 4.1, which is implemented by Linux *mmap()* system calls and file system. We also created a special *memcpy()* function for SCM, *scm_memcpy()*. Currently, PCM has the access latency in the hundreds of nanoseconds, which is only 2–5 times slower than DRAM [13]. Thus, in the function *scm_memcpy()*, we actually invoked current *memcpy()* function 3 times to simulate this PCM access latency.

Based on these OS interfaces, we implemented a prototype of our new SCM-based logging approach in IBM SolidDB using *C* programming language on a 64-bit Linux platform. The reason we selected SolidDB as our code base for modification is that SolidDB is an in-memory database system, which can more easily demonstrate the benefit of SCM-based logging approach.

Specifically, SolidDB uses two memory log buffers for log writings: a log record queue and a log buffer. A log thread is used to move log data from the log buffer to log files on disk. In our prototype, we use SCM log space to replace the current two log buffers and log files. We used one SCM log space to replace these two log buffers. At runtime, the transactions write log records directly to the SCM log space without buffering in DRAM or writing to disk files.

The SCM log space is simulated by 524 megabyte of DRAM. It is managed as a linear space, and disk page format is not required in this log space. The first 512 byte, as the header, is used to save the metadata of SCM log space. The remaining log space is managed as a circular buffer, in which all the log records are packed one by one, and a new log record is always appended immediately after the last log record in the log space.

7.2 Experimental Evaluation

We ran a Telecommunication Application Transaction Processing (TATP) benchmark [6] to compare the performance of our prototype with that of the current SolidDB product. TATP is an open source workload designed specifically for high-throughput applications. It simulates a typical Home Location Register (HLR) database used by a mobile carrier. The HLR application is used by mobile network operators to store all relevant information about the subscribers and the services thereof.

TATP is based on seven pre-defined transactions and queries that insert, update, delete and query the data. The benchmark generates a flooding load on a database server and it runs for a pre-selected sampling time. In our experiments, we set the sampling time to 5 minutes, including 2 minutes for system warm up and 3 minutes for experiment measurement. During that period, the number of times each transaction executed follows the probability assigned to the transactions in the transaction mix.

We have run the database server and the TATP clients on an IBM X serials M3550 server, which has 4 cores, 6 gigabyte

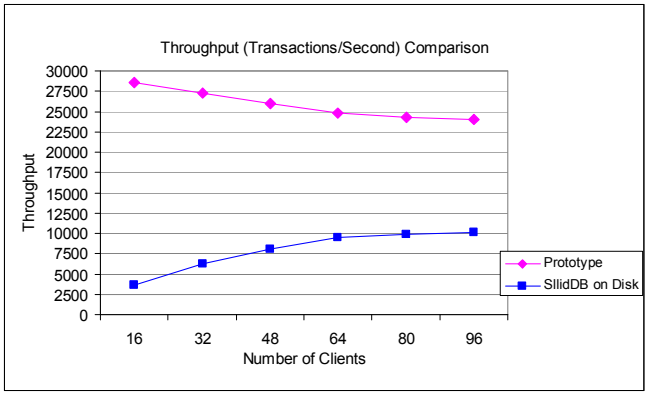
memory and 150G hard disk. The OS of this server is 64-bit Red Hat Linux Enterprise edition. For database server, we set the transaction durability to "strict mode", which means, when a transaction commits, its log records are guaranteed to be written to persistent storage, i.e. SCM segment for our prototype, and log files on disk in the original SolidDB system. Strict mode is the most common usage pattern in real enterprise applications.

We compare the throughput (transactions per second) and response time (in millisecond) of our prototype with the current SolidDB. In addition, we also analyze the prototype code to show the reduction in the code size and the much simplified logging and recovery algorithms in the SCM-based logging and recovery system.

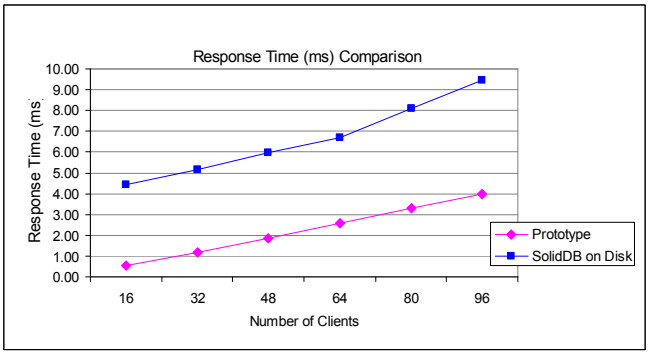
7.2.1 Experimental Results and Performance Analysis

We ran two types of experiments: one is to change the number of clients while keeping the read transactions (queries) to write transactions (R/W) ratio fixed at 80/20, which is believed to be the most representative access pattern in real life; the other one is to keep the number of clients constant, but setting the R/W ratio to 100/0, 80/20, 60/40, 40/60, 20/80 and 0/100 in sequence.

The results of the first type of the experiment are shown in Fig. 7. We increase the number of clients from 16 to 96, by adding 16 clients each time. Fig. 7 shows the times of performance improvement of our prototype compared to the current SolidDB running on hard disk.



(i) Throughput Comparison



(ii) Response Time Comparison

Fig. 7. Performance Comparison of Type 1 Experiment

In this experiment, the throughput of our prototype reaches its peak at 28548 transactions per second, when the number of clients is 16. At this point, the CPU is almost fully utilized, and there is no I/O writing (or waiting). The throughput of the original SolidDB running on disk saturates at 10141 transactions per second, when the number of clients is 96. At this point, the CPU utilization is around 35% while the I/O utilization is almost 100%, which clearly shows that I/O is still the bottleneck even in in-memory database system such as SolidDB.

Our prototype delivers the best performance when the number of clients is 16. At this time, the throughput of our system is 7.89 times higher than SolidDB running on disk; and both systems have the shortest response time, 0.56 millisecond for our prototype and 4.42 milliseconds for SolidDB running on disk.

We have also made a comparison of the response time when three systems have the maximum throughput, which is a common comparison method in the industry. In our experiment, that is when running 16 clients on our prototype, and 96 clients for SolidDB running on disk. Compared to SolidDB running on disk, the response time improvement of our prototype is 16.89 times. This again shows the great performance advantage of the SCM-based logging approach.

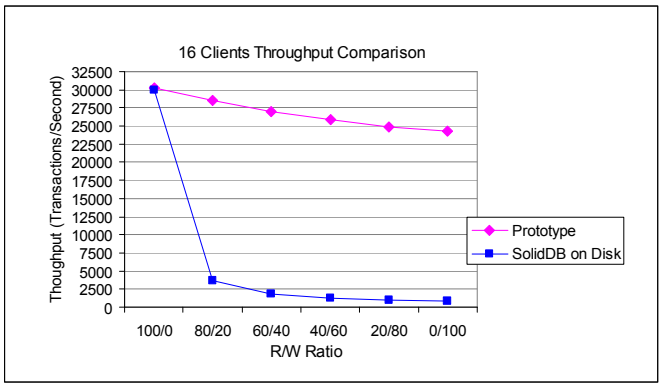
With the number of clients increasing, the throughput of both our prototype degrade gradually. There are two reasons for this degradation. The first is that the CPU time spent on concurrency control also increases when the number of clients rises. The other reason is that we run both TATP clients and database server on the same server. Since the CPU utilization is almost 100% at 16 clients, increasing the number of clients also increases the CPU cycles consumed by the TATP clients and thus taking away CPU cycles available for our prototype to process transaction requests. When there are 96 TATP clients, the CPU spent by the client code is about 45%.

Fig. 8 shows the result of the second type of experiment. In this experiment, the number of TATP clients is set to either 16 or 96, while varying the R/W ratio (transactions mix).

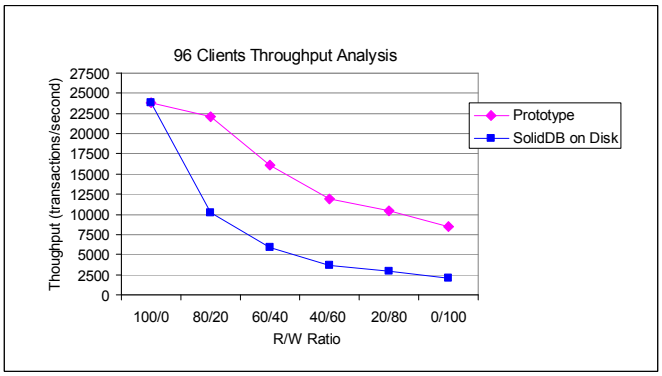
When the R/W ratio is 100/0, the throughputs of the two systems are almost identical. This is because there is no log writing in this case for all three systems tested.

With the probability of write transaction increases from 20% to 100%, the throughput improvement of our prototype over SolidDB running on disk increases from 7.89 times to 29.41 times with 16 TATP clients; and from 2.18 times to 4.03 times with 96 TATP clients. Clearly, the higher the ratio of write transaction, the better performance improvement our prototype will provide.

With 96 TATP clients, the throughput of all three tested systems degrade significantly, with disk-based SolidDB degrades the most, when the write ratio increases from 0% to 100%. The reason is that when the write ratio increases, the average CPU cycles consumed by each transaction, for log writing and for concurrency control, in the database server increases, and thus the throughput decreases.



(i)Throughput When Number of Clients = 16



(ii) Throughput When Number of Clients = 96

Fig. 8. Throughput Analysis of Type 2 Experiment

7.2.2 System Simplification Analysis

As described in Section 5, SolidDB has two log buffers in main memory: a log record queue and a log buffer in DRAM. At runtime, transaction threads first append log records to the end of the log record queue. Then, a dedicated log thread moves the log records from the log queue to the log buffer when it is time to write (force) log records to the log disk. The log thread organizes the log records in the log buffer as multiple of disk blocks, in order to satisfy the requirement of page-oriented I/O protocol. This incurs significant additional complexity in managing partial page log write and in handling crossing page-boundary log records, such as Ping-Pong algorithm [17]. Once the log buffer is full or a transaction commits, the log thread writes the pages in log buffer to log files on disk.

In our prototype, we have used SCM log space to replace the current log files and the two log buffers for log writing. And SCM log space manager replaces the current log thread code to manage the data in SCM log space. With SCM-based logging approach, more than 20% of the original source code for moving log records among different log buffers and log files are eliminated; and all the source code of managing log buffers and log files, over 10000 lines, is removed. What is more, complex algorithms of transforming log records to disk page format and handling the crossing page-boundary log records can also be avoided. Clearly the logging approach has become much simpler and efficient.

On the recovery side, SolidDB starts the recovery process by loading log pages (records) on disk to a recovery log buffer in batches, which is different from the log record queue or log buffer. Then, the recovery system scans log records in this recovery log buffer one by one. This loading and scanning of log records can be a very slow process.

In our prototype the recovery system can locate and attach to the SCM log space and even the location of the latest checkpoint log record by invoking logging interfaces provided by the SCM log space manager. Then, it can just scan the log records inside SCM log space from the latest checkpoint log record without loading them into its own buffer. The source code is reduced by more than 1000 lines, and the time-consuming log records loading process is avoided.

VIII. CONCLUSION

Storage class memory is an emerging memory technology, which has numerous advantages compared to the current DRAM and HDD, such as fast access time, non-volatility, byte-addressability and energy efficiency. With these great properties, SCM will have huge impact on the next generation system and software design in the near future. System performance can be drastically improved by eliminating traditional slow I/O bottlenecks; and a much simpler system design can be accomplished by avoiding complicated data format transformation and I/O latency hiding.

The objective of our work is to design new DBMS technologies to leverage these SCM advantages. Logging is one of the key components in DBMS in terms of transaction support. It is also one of the major performance bottlenecks in DBMSs because of the slow speed of disk I/Os. In this paper, we introduce a new design of an SCM-based database logging approach for DBMSs to reduce the logging overhead and simplify the database logging logic. In this new approach, traditional in-memory log buffers and disk-based log files are replaced by a single log space in SCM. Using SCM, log records can be directly written into this persistent memory, which simplifies the traditional two-layer logging design, provides better concurrency support and improves the transaction latency. We also studied several major problems when using SCM during system crash and developed new recovery algorithm to cope with these issues.

To prove the feasibility of our new design, we built a prototype based on the IBM SolidDB and an SCM simulator. In common circumstances, our experimental results show that the new SCM-based logging component can provide 7x throughput improvement over disk-based logging on the TATP benchmark. If all the transactions are write transactions, the performance improvement can be as much as 29x times.

ACKNOWLEDGMENT

Many thanks to Dr. Moidin Mohiuddin for his helpful comments and suggestions.

REFERENCES

[1] R. Freitas and W. Wilcke, "Storage-class memory: The next storage system technology," *IBM Journal of Research and Development*, Vol. 52, Issue 4, pp.439-447, July 2008.

[2] Benjamin C. Lee, Engin Ipek, Onur Mutlu, Doug Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," *in Proc. ISCA'09*, 2009, pp. 2-13.

[3] 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, pp. 94-162, Mar.1992.

[4] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett and A. Reuter, "Group Commit Timers and High Volume Transaction Systems," *in Proc. HPTS'1987*, 1987, pp. 301-329.

[5] IBM SolidDB In-memory Database Website. [Online]. Available: http://www-01.ibm.com /software/data/soliddb/

[6] TATP Benchmark Website. [Online]. Available: http://tatpbenchmark. sourceforge.net/

[7] Oracle TimesTen In-memory Database Website. [Online]. Available: http://www.oracle. com/database /timesten.html

[8] Altibase Main Memory Database Website. [Online]. Available: http://www.altibase.com /english/product/altibase/alti_dev_bg.jsp

[9] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, Y. Zhang, S. Madden, M. Stonebraker, J. Hugg and D. Abadi, "H-Store: A High-Performance, Distributed Main Memory Transaction Processing System," in *Proc. VLDB'08*, 2008, pp.1496-1499.

[10] VoltDB Product Website. [Online]. Available: http://www.voltdb.com/

[11] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann and R. Stutsman, "The case for RAMClouds: scalable high-performance storage entirely in DRAM," *ACM SIGOPS Operating Systems Review*. Vol. 43, Issue 4, 2009, pp. 92-105, Jan. 2010.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels: "Dynamo: Amazon's Highly Available Key-value Store," in *Proc. SOSP'07,* 2007, pp.205-220.

[13] Cassendra project Website. [Online]. Available: http://cassandra. apache.org/

[14] J. Condit， E. B. Nightingale， C. Frost，E. Ipek， B. Lee， D. Burger and D. Coetzee, "Better I/O Through Byte-Addressable, Persistent Memory," in *Proc. SOSP'09,* 2009, pp.133-146.

[15] W. Richard Stevens. *UNIX Network Programming*, *Vo. 2, Interprocess Communications*, 2nd ed, New Jersey, USA: Prentice Hall, 1998..

[16] A. Silberschatz, P.B. Galvin and G. Gagne. *Operating System Concept,* 7th ed, J. Wiley & Sons Incorporated, New Jersey, USA, 2004, pp. 348-350.

[17] Dynamical Memory Allocation Functions in Glibc Manual, [Online]. Available:http://www.gnu.org/software/libc/manual/html_mono/libc.html#toc_Memory

[18] J. Gray and A. Reuter. *Transaction Processing: Concepts and Technique,* 1st ed, Morgan Kaufmann, CA, USA, 1992, pp. 508-509.

[19] Sang-Won Lee and B. Moon. "Design of Flash-Based DBMS: An In-Page Logging Approach," in *proc ACM SIGMOD'07*, 2007, pp.55-66.

[20] Sang-Won Lee, B. Moon, C. Park, Jae-Myung Kim and Sang-Woo Kim. "A Case for Flash Memory SSD in Enterprise Database Applications," in *proc ACM SIGMOD'08*, 2008, pp.1075-1086.