Deferred Updates for Flash-Based Storage

Biplob Debnath #, Mohamed F. Mokbel *, David J. Lilja #, David Du *

#*Department of Electrical and Computer Engineering*
**Department of Computer Science and Engineering*
University of Minnesota, Twin Cities, USA.
Email: biplob@umn.edu, mokbel@cs.umn.edu, lilja@umn.edu, du@cs.umn.edu

***Abstract*—The NAND flash memory based storage has faster read, higher power savings, and lower cooling cost compared to the conventional rotating magnetic disk drive. However, in case of flash memory, read and write operations are not symmetric. Write operations are much slower than read operations. Moreover, frequent update operations reduce the lifetime of the flash memory. Due to the faster read performance, flash-based storage is particularly attractive for the read-intensive database workloads, while it can produce poor performance when used for the update-intensive database workloads. This paper aims to improve write performance and lifetime of flash-based storage for the update-intensive workloads. In particular, we propose a new hierarchical approach named as *deferred update methodology*. Instead of directly updating the data records, first we buffer the changes due to update operations as logs in two intermediate in-flash layers. Next, we apply multiple update logs in bulk to the data records. Experimental results show that our proposed methodology significantly improves update processing overhead and longevity of the flash-based storages.**

I. Introduction

NAND flash memory is increasingly adopted as main data storage media in mobile devices, such as PDAs, MP3 players, cell phones, digital cameras, embedded sensors, and notebooks due to its superior characteristics such as smaller size, lighter weight, lower power consumption, shock resistance, lesser noise, non-volatile memory, and faster read performance [7], [11], [15], [16]. Recently, to boost up I/O performance and energy savings, flash-based Solid State Disks (SSDs) are also being increasingly adopted as a storage alternative for magnetic disk drives by laptops, desktops, and datacenters [1], [2], [9], [14]. Due to the recent advancement of the NAND flash technology, it is expected that NAND flash based storages will greatly impact the designs of the future storage subsystems [4], [9], [11], [12].

A distinguishing feature of flash memory is that read operations are very fast compared to magnetic disk drive. Moreover, unlike disks, random read operations are as fast as sequential read operations as there is no mechanical head movement. However, a major drawback of the flash memory is that it does not allow *in-place* update (i.e., overwrite) operations. Figure 1 gives an overview of a flash-based storage device. In flash memory, data are stored in an array of flash blocks (as shown in Figure 1). Each block spans 32-64 sectors, where a sector is the smallest unit of read and write operations. Sectors are also commonly referred as pages. However, in the rest of this paper, we use sector in order to distinguish it from a conventional database page. Sector write operations in a flash memory must be preceded by an erase operation. Within a block, sectors must be written sequentially (in low to high address order) [4]. The *in-place* update problem becomes complicated as write operations are performed in the sector granularity, while erase operations are performed in the block granularity. The typical access latencies for *read*, *write*, and *erase* operations are 25 microseconds, 200 microseconds, and 1500 microseconds, respectively [4]. In addition, before an erase operation is being done on a block, the live (i.e., not overwritten) sectors from that block need to be moved to pre-erased blocks. Thus, an erase operation incurs lot of sectors read and write operations, which makes it a performance critical operation. Besides the asymmetric read and write latency issue, flash memory exhibits another limitation: a flash block can only be erased for limited number of times (e.g., 10K-100K) [4].

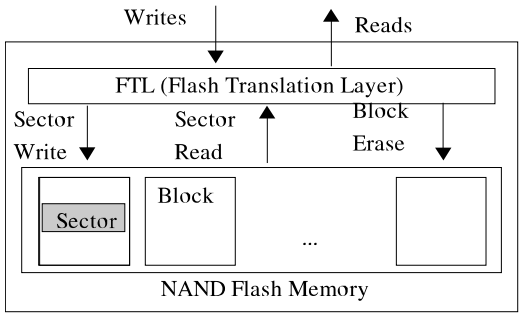


Fig. 1. NAND Flash-Based Storage

Faster read performance of the flash memory will be particularly good to speed up the read-intensive type workloads, e.g., decision support systems (DSS). However, flash memory can produce poor performance when used for other workloads that require frequent random update operations. Examples of such workloads include online transaction processing (OLTP), mobile applications, and spatio-temporal applications. These update-intensive applications perform a lot of small-to-moderate size random write operations that are much smaller than the flash sector size [11]. This will be very problematic for the flash memory as once data is written in a flash sector, no further data can be written without erasing the entire block containing that sector. Thus, update-intensive applications suffer from performance degradation as erase operations are much

978-1-4244-7153-9/10/ $26.00 @2010 IEEE

slower than write operations. Moreover, frequent erasure of the blocks will decrease lifetime of flash memory.

The flash translation layer (FTL) is an intermediate layer inside a flash-based storage (as shown in Figure 1) that hides the internal details of flash memory (i.e., *in-place* update problem) and allows existing disk-based application to use flash memory without any significant modifications [8]. Thus, update-intensive applications can be greatly benefited by using a flash-based storage equipped with FTL. However, not all flash-based storage devices use FTL [5], [15]. For these devices, flash driver can provide the internal flash information. In some cases, operating system provides FTL functionality. For example, Windows Mobile emulates FTL in software [15]. In this paper, we are focusing on this type of flash devices. In addition, we are focusing on flash-based storage with reconfigurable FTL. The intuition of using a reconfigurable FTL is as follows. For the most of flash-based storage products available in the market, internal hardware architectures and FTL designs are not well known [4]. As a result, applications are designed (or modified) based on generic FTL behavior. Although this is adequate for the applications designed for the general computing environment, but for the environments (i.e., high end servers or supercomputers) running a fixed set of applications (i.e., database management systems), huge performance gain could be obtained by using customized flash-based storages designed with application specific FTL. This benefit is demonstrated by a recent project [17], which implements FTL in a reconfigurable hardware (i.e., field-programmable gate array). Each running application has access to FTL and it can reconfigure FTL design based on its own requirements.

In this paper, we propose a novel hierarchical update processing strategy, named *deferred update methodology*, which significantly improves the *in-place* update processing overhead and longevity of the flash-based storage used for a database management system (DBMS). Our goal is to reduce the number of expensive erase operations due to the processing of *in-place* update operations through two intermediate flash storage layers. The main idea is that we always write the changes due to newly incoming updates as logs to the first intermediate layer. Once first layer is full, to make free space we populate the logs from the first layer to the second layer. The first layer acts as a scratch area for the second layer. Finally, when the second intermediate layer is also full, we populate its contents into their actual locations in the flash erase units. These two layers help to batch a set of update logs for the same erase unit together. Finally, we can apply them at once. This results in a huge saving of erase operations where a block is erased only once for a set of bulk updates. Since erase is the most expensive operation, therefore reduction of the number of erase operations helps to improve write performance. On the other hand, this will also help to increase the lifetime of the flash memory due to the limited number of erase operations allowed per block.

Our key contributions can be summarized as follows.

- A hierarchical update processing methodology named as *deferred update methodology* to improve slow write performance and lifetime of the NAND flash memory. The cost of achieving such improvements is only few flash memory blocks.
- A thorough theoretical analysis of the trade-offs in terms of erase operations, space overhead, and data retrieval overhead for different alternative designs compared to the *deferred update methodology*.

The remainder of the paper is organized as follows: Section II describes the related work. Section III describes the *deferred update methodology* in detail. Section IV gives analytical models of the different alternative designs. Section V explains our experimental results. Finally, Section VI concludes the discussion.

II. Related Work

There is substantial recent interest in utilizing flash memory for non-volatile storage in applications including databases and sensor networks. Here, we are discussing the works that are related to the update processing issues of the flash memory. The existing works can be classified into two main categories. **First:** designing flash-friendly data structures. For example, MicroHash [18], FlashDB [16], random sampling data structure [15], FD-tree [13], and Lazy-Adaptive Tree [3] propose new or modified index structures for flash-based storage. However, these works cannot be directly extended to improve flash memory's update processing problem as they mainly target specific index structures. In contrast, our goal is to design a generic solution for database table-spaces as well as index structures. **Second:** improving update processing performance for the flash-based database servers. This includes in-page-logging (IPL) technique [11]. Our work also falls into this category. In the rest of this section, we discuss IPL in detail and distinguish our work from it.

The state-of-the-art technique of handling updates in flash memory is the in-page logging (IPL) approach [11]. The main idea of IPL is to reserve one of data pages in a flash erase unit as log page for storing the update logs. Each page consists of multiple flash sectors. When a data page becomes dirty, the changes are recorded as update logs in an in-memory update log sector. Once the log sector becomes full or dirty data page is evicted from the buffer, then the in-memory update log sector is written to the corresponding log page. Whenever a log page becomes full, the update logs in the log page are combined with the original data records in the data pages in that erase unit. The first problem of IPL is that if the data pages have very little update locality, then in-memory log sectors will contain small amount of data, as a result *log page* space will be under-utilized. This space under-utilization accelerates frequent erase operations, which will slow down performance and affects the lifetime of the flash memory. Second problem with IPL is that if power goes off, the update logs stored in the memory will be lost, thus data inconsistency problem will arise. Our work in this paper targets to develop an efficient update processing methodology that (1) reduces the number of erase operations, (2) increases the space utilization, and (3)

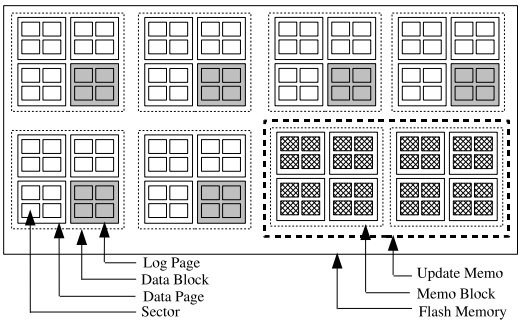


Fig. 2. Logical Flash Memory view for a DBMS in the *deferred update methodology*

avoids data consistency problem.

III. Deferred Update Methodology

In this section, we present the *deferred update methodology* that aims to improve write performance and increase lifetime of the flash-based database systems. The more detailed description can be found in the longer version of this paper [6].

The main idea of *deferred update methodology* is to process updates through an intermediate two-level storage hierarchy consisting of an *update memo* and *log page(s)*. This intermediate layer helps to reduce the number of expensive erase operations. Conceptually, we group the database pages by the erase unit containing them and named as *data blocks*, while *update memo* is a set of erase units which is used as a scratch space. In each *data block*, some data pages are also reserved for storing *update logs*. We name these reserved pages as *log pages*. Figure 2 represents a logical view of an NAND flash memory that employs our proposed *deferred update methodology*. The boundary of flash memory is depicted by the solid rectangle. In this example, flash memory has eight erase blocks, which are depicted by the fine dotted rectangles. Out of the eight erase blocks, two blocks are reserved as *update memo* blocks, which are bounded by the solid dotted rectangles; while the remaining six blocks are used as *data blocks*. In each *data block*, there are four data pages. One of the data pages will be reserved as *log page*, which is marked in shaded in gray. Each data page consists of four flash sectors as depicted by the smallest solid rectangles.

The left side of Figure 3 gives an overview of the update processing methodology through *update memo* and *log pages*. The update processing has three steps. **Step 1:** when an update transaction (i.e., `INSERT`, `UPDATE`, or `DELETE`) occurs, the changes made by the transactions are stored as *update logs* in the available sectors of the *update memo* blocks. **Step 2:** when *update memo* is full, the latest *update logs* are flushed to the *log pages* of the corresponding *data blocks*. A *timestamp* counter is used to identify the latest *update logs*. In addition, an index is used to speed up the flushing process. **Step 3:** when the *log pages* of a *data block* are also full, the *update logs* are stored in-place with the old data records. Without *update memo* and *log pages*, for every *in-place* update operation, we would need to erase a flash block. However, with the help of *update memo* and *log pages*, processing of the *in-place* update

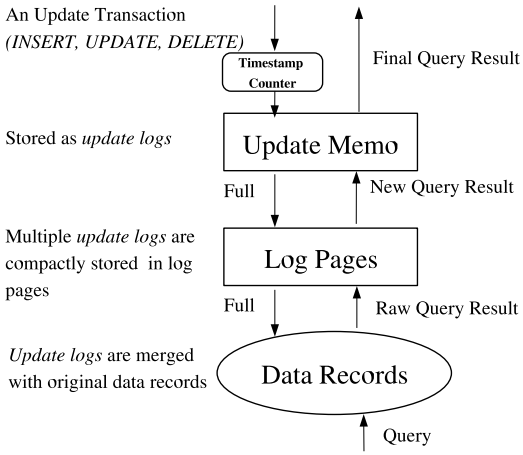


Fig. 3. Overview of the *deferred update methodology*

operations are deferred. The *update memo* acts as a buffer and supplies multiple *update logs* at once to the *log pages*. These logs are stored compactly in the *log pages*. Thus, *update memo* provides the opportunity to compact more *update logs* and to bulk updates once for each block, which internal *log pages* cannot do. Overall, *update memo* and *log pages* helps to reduce total number of block erase operations due to *in-place* updates by amortizing cost of single *data block* erase operation among multiple update operations.

The right side of Figure 3 gives an overview of the query processing steps in *deferred update methodology*. Queries are processed in the reverse order of the update processing method. It also consists of three steps. **Step 1:** a raw query result set is generated from the data records stored in the data pages using the traditional query processing techniques. **Step 2:** initial raw query result set is modified through the *update logs* stored in the relevant *log pages*. **Step 3:** the new result set is modified further by processing the latest *update logs* stored in the *update memo*. A flash-friendly hash index is used to expedite this step. The second and third steps ensure that query result is correct.

There is a trade-off between the gain in the update processing performance and query processing overhead. Keeping lot of *update logs* in the *update memo* and *log pages* improves update processing performance. However, this strategy increases the overhead of query processing as we have to scan larger *update memo* and more *log pages* to generate correct results. The *number of blocks* in the *update memo* and the number *log pages* in a *data block*, are the two tuning parameters to control the performance gain of the *deferred update methodology*.

IV. Analysis of Log Pages and Memo

We analyze *deferred update methodology* that uses both *update memo* and *log pages* compared to the other three alternative approaches to process N_u update transactions. **(1) No Log and No Memo Approach:** this is the existing design with no change for flash memory i.e., this model has no *log pages* and no *update memo*. **(2) Log Page Only Approach:** In this model, each *data block* contains n_l number of *log pages*. However, no *update memo* is maintained. This is similar

	Total Erase Operations (N_E)	**Space Requirements *in blocks***	**Query Processing Overhead ($q_{overhead}$) *in time to scan pages***
No Log and No Memo	N_u	$\frac{DB_{size}}{B}$	0
Log Page Only	$N_u * \frac{S}{n_l * P}$	$\frac{DB_{size}}{B - n_l * P}$	$\frac{1}{2} * N_q * n_l$
Update Memo Only	$N_u * (\frac{S}{B} + \frac{1}{n_{ul}})$	$\frac{DB_{size}}{B} + N_M$	$\frac{1}{2} * N_M * \frac{B}{P}$
Log Page and Update Memo (Deferred Update)	$N_u * (\frac{S}{B} + \frac{1}{\lfloor \frac{n_l * \frac{P}{S}}{\lceil \frac{n_{ul} * u_s}{S} \rceil} \rfloor * n_{ul}})$	$\frac{DB_{size}}{B - n_l * P} + N_M$	$\frac{1}{2} * (N_q * n_l + N_M * \frac{B}{P})$

TABLE II
ANALYTICAL COMPARISON OF DIFFERENT ALTERNATIVE DESIGNS USING *log pages* AND *update memo*

Symbol	**Description**
DB_{size}	Database size in bytes
B	Flash block size in bytes
P	Database page size in bytes
S	Flash sector size in bytes
n_l	Number of *log pages* per data block
u_s	Average *update log* size in bytes
n_{ul}	Average number of *update logs* each data block receiving during memo cleaning
N_u	Total number of update transactions
N_D	Total number of data blocks
N_q	Total number of data blocks containing data records satisfying query q
N_M	Total number of *update memo* blocks
N_E	Total number of block erase operations to process N_u update transactions
$q_{overhead}$	Query processing overhead
$S_{overhead}$	Space overhead

TABLE I
SYMBOLS DESCRIPTION

Parameter	**Value**
Block size	256 KB
Sector size	4 KB
Data Register Size	4 KB
4KB-Sector Read to Register Time	25 μs
4KB-Sector Write Time from Register	200 μs
Serial Access time to Register (Data bus)	100 μs
Block Erase Time	1500 μs

TABLE III
PARAMETERS VALUES FOR NAND FLASH MEMORY

to the model adopted by the IPL technique [11], when setting $n_l = 1$. **(3) Update Memo Only Approach:** This model has only *update memo*. However, no *log pages* are maintained. There are N_M blocks in the *update memo*. At first, *update logs* are stored in the memo. When memo is full, *update logs* are merged with the old data records. The functionality of this approach is quite similar to the space-efficient FTL design work [10].

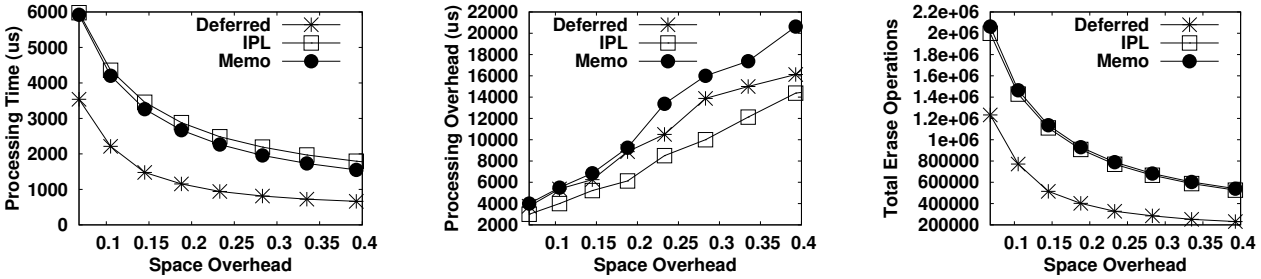
Our measure of analysis are (a) space overhead ($s_{overhead}$), (b) query processing overhead ($q_{overhead}$), and (c) total number of erase operations (N_E). During this analysis, without loss of generality, we assume that the update transactions are uniformly distributed to all database pages and average *update log* size (u_s) is less than a flash sector size (S). To simplify the analysis, we further assume that there is no index over the *update logs* stored in the *update memo* and each overwrite operation in the flash memory incurs an erase operation. Table I introduces various symbols used in this analysis. Table II summarizes the total number of erase operations performed, space requirements, and query processing overhead for different alternative designs using *log pages* and *update memo*. The detailed analysis can be found in the longer version of this paper [6].

V. EXPERIMENTAL RESULTS

We compare *deferred update methodology* with in-page logging (IPL) technique [11] and *Update Memo Only* approach to handle update transactions. IPL is the state-of-the-art technique for handling update transactions for the flash-based storage. It is a special case of *Log Page Only* approach (as described in Section IV) with one *log page* per erase unit. On the other hand, the working principle of *Update Memo Only* approach is very similar to space-efficient log-based FTL design work [10]. We do not consider *No Memo and No Log* approach in the comparison, as it is not suitable for processing update transactions [11] in the flash-based storage.

Simulator and Traces. To evaluate *deferred update methodology*, similar to IPL technique [11], we have implemented a standalone event-driven simulator in the C language on the Linux platform. We use synthetic traces to evaluate *deferred update methodology*. The update transactions in this trace are uniformly distributed over all database pages and there is almost no temporal locality (less than 1%). This trace emulates one of worst writes access patterns for the flash memory. The online transaction processing (OLTP) type applications exhibit quite close behavior to this trace. The size of an *update log* in each transaction lies in between 20-100 bytes. We assume that each database page size is 8 KB.

Performance Calculation Formulas. To calculate the update processing time, we use the following formula: *total number of erase operations * erase time + total sector read operations * (sector read time + page register access time) + total sector write operations * (sector write time + page register access time)*. While for the query processing overhead we use: *the numbers of extra flash sectors read due to query processing* (sector read time + page register access time)*. During query processing, the simulator takes query selectivity and returns *the numbers of extra flash sectors* to process that query. Table III gives the various flash parameters values used to in the experiments. These values are taken from the SSD design project [4].



(a) Average update processing time

(b) Processing overhead for a query having 3% selectivity

(c) Erase count

Fig. 4. Performance trends with varying space overhead in a 100 MB database. Here, 'Memo' stands for *Update Memo Only* approach.

Space Overhead	IPL	Memo Only	Deferred Update
6.8%	$n_l = 2$	$N_M = 27$	$n_l = 1, N_M = 14$
10.5%	$n_l = 3$	$N_M = 42$	$n_l = 1, N_M = 29$
14.5%	$n_l = 4$	$N_M = 58$	$n_l = 2, N_M = 31$
18.8%	$n_l = 5$	$N_M = 75$	$n_l = 2, N_M = 48$
23.3%	$n_l = 6$	$N_M = 93$	$n_l = 3, N_M = 51$
28.3%	$n_l = 7$	$N_M = 113$	$n_l = 3, N_M = 71$
33.5%	$n_l = 8$	$N_M = 134$	$n_l = 4, N_M = 76$
39.3%	$n_l = 9$	$N_M = 157$	$n_l = 4, N_M = 99$

TABLE IV
VALUES OF n_l AND N_M FOR SAME SPACE OVERHEAD

A. Parameter Selection

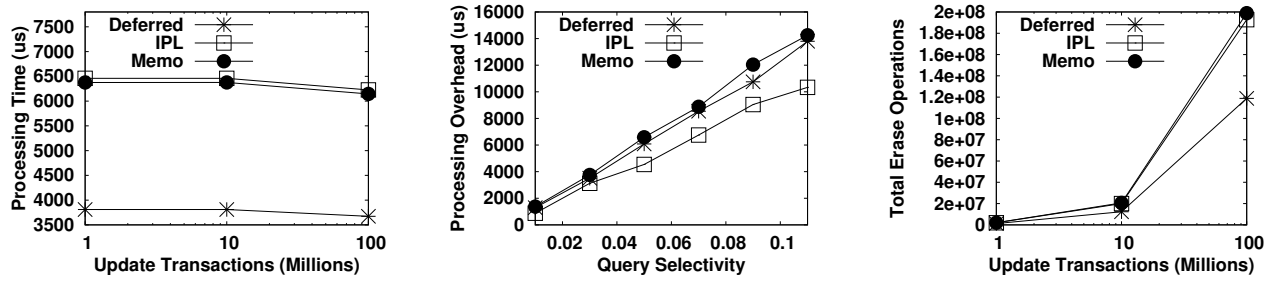
In-page logging (IPL) technique [11] uses *number of log pages per block* (n_l) and *Memo Only* approach uses *number of update memo blocks* (N_M), while *deferred update methodology* uses both n_l and N_M. The space and query processing overhead depend on n_l and N_M. We vary n_l from 1 to 31, and N_M from 1 to 400, for a database of size 100 MB processing one million update transactions. Since, three different approaches use various parameters, to make a fair evaluation, we estimate the average update processing time and query processing time for the same space overhead. Table IV gives different parameters values for which average update processing time is minimum among all configurations. Because IPL technique uses only *log pages* and *deferred update methodology* uses both *log pages* and *update memo*, we set the minimum value of n_l to 2 for IPL.

Figure 4(a) gives the average update processing time for the same space overhead. As expected, with the increase of space overhead, the average update processing time decreases in all three approaches. Compared to IPL, *deferred update methodology* improves average update processing time by 50%-63%, while *Update Memo Only* approach improves update procesing time by 1%-17%. The average update processing time decreases with the increase of the space, as we can buffer more *update logs* and apply them in bulk. This helps to reduce the total number of erase operations. This trend is shown by Figure 4(c). Erase operations are performed in the block-level and before erasing a block we need to move data and write them back. Thus, erase operations incur huge latency. With the decrease in the number of erase operations, this latency overhead decreases, which consequently helps to improve the update processing performance. On the other hand, Figure 4(b) shows that in IPL with the increase of space overhead, the query processing overhead also increases. This happens as with the increase in space, more *update logs* are buffered and we need to process more logs to generate correct query result. Compared to IPL, *deferred update methodology* incurs query processing overhead up to 17%, while *Update Memo Only* approach incurs query processing up to 44%.

Analytical Model Check. The performance trend in Figure 4 is consistent with the analytical model developed in Section IV. Table II shows that with increase of number of log pages (n_l) in IPL (which is a *Log Page only* approach), space overhead increases, erase counts decreases (which helps to improve update processing performance), and query processing overhead increases due to additional processing of the larger number of *update logs* stored in the log pages. Similarly, Table II shows that for the *Update Memo Only* approach, with the increase in number of *update memo* block (N_M), space overhead also increases. However, this additional space helps to hold more *update logs* (n_{ul}), which reduces the total number of erase operations, and consequently improves the update processing performance. In contrast, increasing N_M introduces query processing overhead due to additional processing of the larger number of *update logs* stored in the larger *update memo*. According to Table II, in *deferred update methodology*, increasing both (n_l) and (N_M) contribute to the increased space overhead. However, larger n_l and N_M help to hold larger number of *update logs* (n_{ul}) and process them in bulk, which reduces the total number of erase operations. Thus, it helps to improve update processing time. On the other hand, query processing overhead increases with the increased space overhead due to processing of large number of *update logs*.

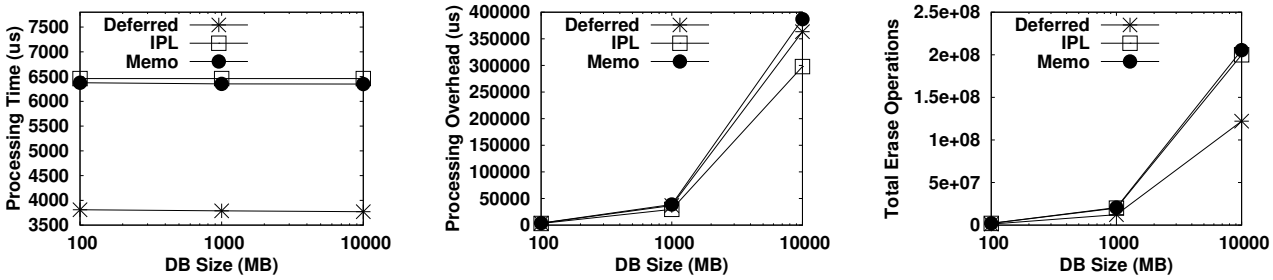
B. Scalability

We demonstrate the scalability of the *deferred update methodology* with the increase in the number of update transactions and size of a database. We keep the space overhead same (i.e., 6.8%) for all three approaches. First, we vary the number of update transactions from one million to 100 millions as shown in Figure 5. Next, we vary database size as well as in proportion vary the number of update transactions



(a) Average update processing time (b) Query processing overhead (c) Erase count

Fig. 5. Scalability with varying the number of update transactions in a 100 MB database. Here, 'Memo' stands for *Update Memo Only* approach.



(a) Average update processing time (b) Processing overhead for a query having 3% selectivity (c) Erase count

Fig. 6. Scalability with varying database size. Here, 'Memo' stands for *Update Memo Only* approach.

as shown in Figure 6. Both figures show that the *deferred update methodology* scales very well with the increase in update transactions as well as data size.

VI. Conclusion

In this paper, we have proposed a flash-friendly hierarchical update processing technique, named as *deferred update methodology*. Our main goal is to improve update processing overhead and increase lifetime of flash memory by reducing the total number of erase operations preformed in order to process update transactions. Our experimental results show the *deferred update methodology* outperforms state-of-the-art update processing techniques.

Acknowledgment

This work was supported in part by National Science Foundation grant no. CCF-0621462, the University of Minnesota Digital Technology Center Intelligent Storage Consortium, and the Minnesota Supercomputing Institute.

References

[1] MacBook Air: SSD Media Capacity. http://support.apple.com/kb/HT2734.

[2] White Paper: MySpace Uses Fusion Powered I/O to Drive Greener and Better Data Centers. http://www.fusionio.com/PDFs/myspace-case-study.pdf.

[3] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. In *VLDB*, 2009.

[4] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX*, 2008.

[5] L. Bouganim, B. Jonsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR*, 2009.

[6] B. Debnath, M. F. Mokbel, D. J. Lilja, and D. Du. Deferred Updates for Flash-Based Storage. *UMN Laboratory for Advanced Research in Computing Technology and Compilers Technical Report, ARCTiC 09-02*, Dec 2009.

[7] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. In *ACM Computing Surveys*, volume 37, 2005.

[8] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *ASPLOS*, 2009.

[9] H. Kim and S. Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *FAST*, 2008.

[10] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. A Space-Efficient Flash Translation Layer for CompactFlash Systems. In *IEEE Transactions on Consumer Electronic*, volume 48, 2002.

[11] S. Lee and B. Moon. Design of Flash-based DBMS: an In-page Logging Approach. In *SIGMOD*, 2007.

[12] S. Lee, B. Moon, C. Park, J. Kim, and S. Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *SIGMOD*, 2008.

[13] Y. Li, B. Hey, Q. Luo, and K. Yi. Tree Indexing on Flash Disks. In *ICDE*, 2009.

[14] M. Moshayedi and P. Wilkison. Enterprise SSDs. *ACM Queue*, 6(4), 2008.

[15] S. Nath and P. Gibbons. Online Maintenance of Very Large Random Samples on Flash Storage. In *VLDB*, 2008.

[16] S. Nath and A. Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *IPSN*, 2007.

[17] J. Shin, Z. Xia, N. Xu, R. Gao, X. Cai, S. Maeng, and F. Hsu. FTL Design Exploration in Reconfigurable High-Performance SSD for Server Applications. In *ICS*, 2009.

[18] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. Najjar. MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices. In *FAST*, 2005.