gorithms and access methods will function adequately without any modification. On the other hand, due to a few limitations of flash memory, this approach is not likely to yield the best attainable performance. With flash memory, no data item can be updated in place without erasing a large block of flash memory (called *erase unit*) containing the data item. As is shown in Table 1, writing a sector into a clean (or erased) region of flash memory is much slower than reading a sector. Since overwriting a sector must be preceded by erasing the erase unit containing the sector, the effective write bandwidth of flash memory will be even worse than that. It has been reported that flash memory exhibits poor write performance, particularly when small-to-moderate sized writes are requested in a random order [2], which is quite a common access pattern for database applications such as on-line transaction processing (OLTP). These unique characteristics of flash memory necessitate elaborate flash-aware data structures and algorithms in order to effectively utilize flash memory as data storage media.

In this paper, we present a novel *in-page logging (IPL)* approach toward the new design of flash-based database servers, which overcomes the limitations of and exploit the advantages of flash memory. To avoid the high latency of write and erase operations that would be caused by small random write requests, changes made to a data page are buffered in memory on the *per-page* basis instead of writing the page in its entirety, and then the change logs are written sector by sector to the log area in flash memory for the changes to be eventually merged to the database.

The most common types of flash memory are NOR and NAND. NOR-type flash memory has a fully memory-mapped random access interface with dedicated address and data lines. On the other hand, NAND-type flash memory has no dedicated address lines and is controlled by sending commands and addresses through an indirect IO-like interface, which makes NAND-type flash memory behave similarly to magnetic disk drives it was originally intended to replace [15]. The unit of read and write operations for NAND-type flash memory is a sector of typically 512 bytes, which coincides with the size of a magnetic disk sector. For the reason, the computing platforms we aim at in this paper are assumed to be equipped with NAND-type flash memory instead of magnetic disk drives. Hereinafter, we use the term flash memory to refer to NAND-type flash memory, unless we need to distinguish it from NOR-type flash memory.

The key contributions of this work are summarized as follows.

- A novel storage management strategy called *in-page logging* is proposed to overcome the limitations of and exploit the advantages of flash memory, which is emerging as a replacement storage medium for magnetic disks. For the first time, we expose the opportunities and challenges posed by flash memory for the unique workload characteristics of database applications. Our empirical study demonstrates that the IPL approach can improve the write performance of conventional database servers by up to an order of magnitude or more for the OLTP type applications.
- The IPL design helps achieve the best attainable performance from flash memory while minimizing the changes made to the overall database server architecture. This shows that it is not only feasible but also viable to run a full-fledged database server on a wide spectrum of computing platforms with flash memory replacing magnetic disk drives.
- With a few simple modifications to the basic IPL design, the update logs of the IPL can be used to realize a lean recovery mechanism for transactions such that the overhead during normal processing and the cost of system recovery can be reduced considerably. This will also help minimize the memory foot-print of a database server, which is particularly beneficial to mobile or embedded systems.

The rest of this paper is organized as follows. Section 2 discusses the characteristics of flash memory and their impact on disk-based database servers, and then presents the design objectives of the storage subsystem we propose. Section 3 describes the basic concepts and the design of the in-page logging (IPL) scheme. In Section 4, we analyze the performance of a traditional disk-based database server with the TPC-C benchmark, and demonstrate the potential of the in-page logging for considerable improvement of write performance through a simulation study. Section 5 discusses how the basic IPL design can be extended to support transactional database recovery. Lastly, Section 6 surveys the related work, and Section 7 summarizes the contributions of this paper.

2. DESIGN PRINCIPLES

In this section, we describe the key characteristics of flash memory that distinguish itself from magnetic disk drives, and elaborate on how they would affect the performance of traditional disk-based database servers. We then provide the design principles for new flash-based database servers.

2.1 Characteristics of Flash Memory

2.1.1 No In-Place Update

Most traditional database systems assume magnetic disks as the secondary storage media and take advantage of efficient updates of data items by overwriting them in place. On the other hand, with flash memory, no data item (or a sector containing the data item) can be updated in place just by overwriting it. In order to update an existing data item, a time-consuming erase operation must be performed before overwriting. To make it even worse, the erase operation cannot be performed selectively on the particular data item or sector, but can only be done for an entire block of flash memory called *erase unit* containing the data item, which is much larger than a sector (typically 16 KBytes or 128 KBytes). Since every update request will cause an erase operation followed by a write, the effective update performance may degrade significantly on database servers with a flash-based storage system.

Consequently, in order to overcome the *erase-before-write* limitation of flash memory, it is essential to reconsider the current design of storage subsystems and reduce the requests of an erase operation to the minimum so that the overall performance will not be impaired.

2.1.2 No Mechanical Latency

Flash memory is a purely electronic device and thus has no mechanically moving parts like disk heads in a magnetic disk drive. Therefore, flash memory can provide uniform random access speed. Unlike magnetic disks whose seek and rotational delay often becomes the dominant cost of reading or writing a sector, the time to access data in flash memory is almost linearly proportional to the amount of data irrespective of their physical locations in flash memory. [1]

The ability of flash memory to quickly perform a sector read or a sector (clean) write located anywhere in flash memory is one of the key characteristics we can take advantage of. In fact, this

[1]Even though it takes a rather long time for NAND flash to read out the first data byte compared to NOR flash because of the resistance of the NAND cell array, this time is still much shorter than the seek time for a magnetic disk by several orders of magnitude [15].

brings new opportunities for more efficient design of flash-based database server architectures (*e.g.*, non-sequential logging with no performance penalty). We will show how this property of flash memory can be exploited in our design of in-page logging.

2.1.3 Asymmetric Speed of Read/Write

The read and write speed of flash memory is asymmetric, simply because it takes longer to write (or inject charge into) a cell until reaching a stable status than to read the status from a cell. As is shown in Table 1, the read speed is typically at least twice faster than write speed. On the other hand, most existing software systems with magnetic disks implicitly assume that the speed of read and write operations is almost the same. Not surprisingly, there is little work targeted at the principles and algorithms for storage media with asymmetric speed of read and write operations.

This property of asymmetric speed makes us review many existing techniques for DBMS implementation. We ultimately realize that it is critical to find ways to reduce write operations (and erase operations as a result of that), even though it increases the number of read operations, as long as the overall performance improves.

2.2 Problems with Conventional Designs

Most disk-based database systems rely on a paged I/O mechanism for database update and buffer management and take advantage of sequential accesses given the hardware characteristics of disk storage media composed of sectors, tracks and cylinders. One of the immediate implications is that even an update of a single record will cause an entire page (typically of 4 or 8 KBytes) containing the record to be overwritten. If the access pattern to database records is random and scattered, and the update granularity is small (as is often observed in the OLTP applications), the aggregate amount of data to be overwritten is likely to be much larger than the actual amount of data to be updated. Nonetheless, most disk-based database systems are still capable of dealing with such frequent updates, as is mentioned before, by overwriting them in place.

Suppose all the magnetic disks are replaced by flash memory in the computing platform which a conventional disk-based database system runs on. If the database system insists on updating data items in place, then, due to the erase-before-write limitation of flash memory, each update can only be carried out by erasing an entire erase unit after reading its content to memory followed by writing the updated content back to the erase unit. This will obviously lead to high update latency due to the cost of frequent erase operations as well as an increased amount of data to read from and write to flash memory. Moreover, at the presence of hot data items repeatedly updated, this would shorten the life span of flash memory, because an erase unit can be put through a finite number of erase cycles (typically up to 100,000 times) before becoming statistically unreliable.

Most flash memory devices or host systems adopt a process called *wear leveling* within the device themselves or in the software layers in order to ensure that erase cycles are evenly distributed across the entire segment of flash memory so that the life span of flash memory is prolonged. For example, a log-based flash file system ELF achieves wear leveling by creating a new sequential log entry for each write operation. Thus, flash memory is used sequentially all the way through, only returning to previously used sectors after all of the sectors have been written to at least once [5]. Since an update is performed by writing the content into a new sector different from the current sector it was read from, an update does not require an erase operation as long as a free (*i.e.*, erased) sector is available. This sequential logging approach, however, optimizes write performance to the detriment of read performance to the extent that the overall performance of transactional database processing may not be actually improved [9]. In addition, this approach tends to consume free sectors quite rapidly, which in turn requires frequent garbage collections to reclaim obsolete sectors to the pool of erased sectors.

Under either of these two approaches, namely *in-place updating* and *sequential logging*, the average latency of an update operation increases due to frequent execution of costly erase operations. This may become the major bottleneck in the overall performance of a database server particularly for write-intensive workload.

2.2.1 Disk-Based Server Performance

To make the points raised above more acute, we ran a commercial database server on two computer systems that were identical except that one was equipped with a magnetic disk drive and the other with a flash memory storage device instead of the disk drive. In each case of the experiment, we executed an SQL query that accessed the same base table differently - sequentially or randomly, and measured the response time of the query in the wall clock time. Table 2 summarizes the read and write performance of the magnetic disk drive and the flash memory device, respectively, in terms of the random-to-sequential performance ratio.

Media	Random-to-Sequential Ratio	
	Read workload	Write workload
Magnetic Disk†	4.3 ∼ 12.3	4.5 ∼ 10.0
NAND Flash‡	1.1 ∼ 1.2	2.4 ∼ 14.2

†Disk: Seagate Barracuda 7200.7 ST380011A
‡NAND Flash: Samsung K9WAG08U1A 16 Gbits SLC NAND

Table 2: DBMS Performance: Sequential vs. Random

In the case of a magnetic disk drive, the random-to-sequential ratio was fairly high for both read and write queries. This result should not be surprising, given the high seek and rotational latency of magnetic disks. In the case of a flash memory device, the result was mixed and indeed quite surprising. The performance of a read query was insensitive to access patterns, which can be perfectly explained by the no-mechanical-latency property of flash memory. In contrast, the performance of a write query was even more sensitive to access patterns than the case of disk. This is because, with a random access pattern, each update request is very likely to cause an erase unit containing the data page being updated to be copied elsewhere and erased. This clearly demonstrates that database servers would potentially suffer serious update performance degradation if they ran on a computing platform equipped with flash memory instead of magnetic disks. See Section 4.1 for more detail of this experiment.

2.3 Design Manifesto

When designing a storage subsystem for flash-based database servers, we assume that the memory hierarchy of target computing platforms consists of two levels: volatile system RAM and non-volatile NAND flash memory replacing magnetic disks. Guided by the unique characteristics of flash memory described in this section, the design objectives of the storage subsystem are stated as follows.

- Take advantage of new features of flash memory such as uniform random access speed and asymmetric read/write speed. The fact that there is no substantial penalty for scattered random accesses allows us more freedom in locating data ob-

jects and log records across the flash-based storage space. In other word, log records can be scattered all over flash memory and need not be written sequentially.

- Overcome the erase-before-write limitation of flash memory. In order to run a database server efficiently on the target computing platforms, it is critical to minimize the number of write and/or erase requests to flash memory. Since the read bandwidth of flash memory is much faster than that of write, we may need to find ways to avoid write and erase operations even at the expense of more read operations. This strategy can also be justified by an observation that the fraction of writes among all IO operations increases, as the memory capacity of database servers grows larger [9].

- Minimize the changes made to the overall DBMS architectures. Due to the modular design of contemporary DBMS architectures, the design changes we propose to make will be limited to the buffer manager and storage manager.

3. IN-PAGE LOGGING APPROACH

In this section, we present the basic concepts of the *In-Page Logging (IPL)* approach that we propose to overcome the problems of the conventional designs for disk-based database servers. We then present the architectural design and the core operations of the In-Page Logging. In Section 5, we will show how the basic design of IPL can be extended to provide transactional database recovery.

3.1 Basic Concepts

As described in Section 2.2, due to the erase-before-write limitation of flash memory, updating even a single record in a page results in invalidating the current page containing the record, and writing a new version of the page into an already-erased space in flash memory, which leads to frequent write and erase operations. In order to avoid this, we propose that only the changes made to a page are written (or *logged*) to the database on the *per-page* basis, instead of writing the page in its entirety.

Like conventional *sequential logging* approaches (*e.g.*, log-structured file system [23]), all the log records might be written sequentially to a storage medium regardless of the locations of changes in order to minimize the seek latency, if the storage medium were a disk. One serious concern with this style of logging, however, is that whenever a data page is to be read from database, the current version of the page has to be re-created by applying the changes stored in the log to the previous version of the page. Since log records belonging to the data page may be scattered and can be found only by scanning the log, it may be very costly to re-create the current page from the database.[2]

In contrast, since flash memory comes with no mechanical component, there is no considerable performance penalty arising from scattered writes [8], and there is no compelling reason to write log records sequentially either. Therefore, we *co-locate* a data page and its log records in the same physical location of flash memory, specifically, in the same erase unit. (Hence we call it *In-Page* logging.) Since we only need to access the previous data page and its log records stored in the same erase unit, the current version of the page can be re-created efficiently under the IPL approach. Although the amount of data to read will increase by the number of log records belonging to the data page, it will still be a sensible trade-off for the reduced write and erase operations particularly considering the fact that read is typically at least twice faster than write for flash memory. Consequently, the IPL approach can improve the overall write performance considerably.

[2]LGeDBMS, recently developed for embedded systems with flash memory, adopted the sequential logging approach for updating data pages [17].

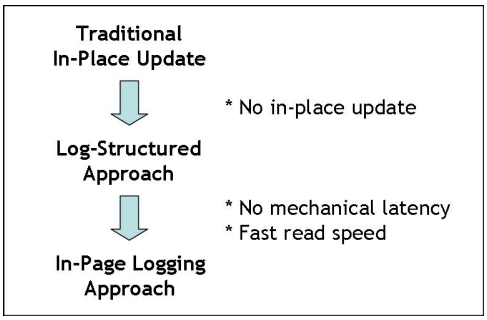


Figure 1: From Update-In-Place to In-Page Logging

Figure 1 illustrates how the IPL approach is evolved from the traditional update-in-place and log-structured approaches. While logging is a consequential decision due to the erase-before-write (or no update-in-place) limitation of flash memory, *in-page* logging is to take advantage of the desirable properties (*i.e.*, no mechanical latency and fast reads) of flash memory.

3.2 The Design of IPL

As is mentioned in the design manifesto (Section 2.3), the main design changes we propose to make to the overall DBMS architecture are limited to the buffer manager and storage manager. In order to realize the basic concepts of the in-page logging with the minimal cost, logging needs to be done by the buffer manager as well as the storage manager. See Figure 2 for the illustration of the IPL design.

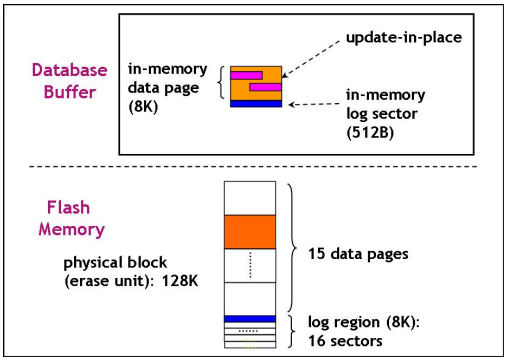


Figure 2: The Design of In-Page Logging

Whenever an update is performed on a data page, the in-memory copy of the data page is updated just as done by traditional database servers. In addition, the IPL buffer manager adds a physiological log record on the per-page basis to the *in-memory* log sector associated with the in-memory copy of the data page. An in-memory log sector can be allocated on demand when a data page becomes dirty, and can be released when the log records are written to a log sector on the flash memory. The log records in an in-memory log sector are written to flash memory when the in-memory log sector becomes full or when a dirty data page is evicted from the buffer pool. The effect of the in-memory logging is similar to that of write caching [22], so that multiple log records can be written together

at once and consequently frequent erase operations can be avoided. When a dirty page is evicted, it is not necessary to write the content of the dirty page back to flash memory, because all of its updates are saved in the form of log records in flash memory. Thus, the previous version of the data page remains intact in flash memory, but is just augmented with the update log records.

When an in-memory log sector is to be flushed to flash memory, its content is written to a flash log sector in the erase unit which its corresponding data page belongs to, so that the data page and its log records are physically co-located in the same erase unit. To do this, the IPL storage manager divides each erase unit of flash memory into two segments – one for data pages and the other for log sectors. For example, as shown in Figure 2, an erase unit of 128 KBytes (commonly known as large block NAND flash) can be divided into 15 data pages of 8 KBytes each and 16 log sectors of 512 bytes each. (Obviously, the size of an in-memory log sector must be the same as that of a flash log sector.) When an erase unit runs out of free log sectors, the IPL storage manager merges the data pages and log sectors in the erase unit into a new erase unit. This new *merge* operation proposed as an internal function of IPL will be presented in Section 3.3 in more detail.

This new logic for update requires the redefinition of read operation as well. When a data page is to be read from flash memory due to a page fault, the current version of the page has to be computed on the fly by applying its log records to the previous version of the data page fetched from flash memory. This new logic for read operation clearly incurs additional overhead for both IO cost (to fetch a log sector from flash memory) and computational cost (to compute the current version of a data page). As is pointed out in Section 3.1, however, this in-page logging approach will eventually improve the overall performance of the buffer and storage managers considerably, because write and erase operations will be requested less frequently.

The memory overhead is another factor to be examined for the design of IPL. In the worst case in which all the pages in the buffer pool are dirty, an in-memory log sector has to be allocated for each buffer page. In the real-world applications, however, an update to a base item is likely to be quickly followed by updates to the same or related items (known as update locality) [1], and the average ratio of dirty pages in buffer is about 5 to 20 percent [14]. With such a low ratio of dirty pages, if a data page is 8 KBytes and an in-memory log sector is 512 bytes, then the additional memory requirement will be no more than 1.3% of the size of the buffer pool. Refer to Section 4.2.2 for the update pattern of the TPC-C benchmark.

3.3 Merge Operation

An in-memory log sector can store only a finite number of log records, and the content is flushed into a flash log sector when it becomes full. Since there are only a small number of log sectors available in each erase unit of flash memory, if data pages fetched from the same erase unit get updated often, the erase unit may run out of free log sectors. It is when merging data pages and their log sectors is triggered by the IPL storage manager. If there is no free log sector left in an erase unit, the IPL storage manager allocates a free erase unit, computes the new version of the data pages by applying the log records to the previous version, writes the new version into the free erase unit, and then erases and frees the old erase unit. The algorithmic description of the merge operation is given in Algorithm 1.

The cost of a merge operation is clearly much higher than that of a basic read or write operation. Specifically, the cost of a merge operation c_{merge} will amount to $(k_d + k_l) \times c_{read} + k_d \times c_{write} + c_{erase}$ for IO plus the computation required for applying log records to data pages. Here, k_d and k_l denote the number of data sectors and log sectors in an erase unit, respectively. Note that a merge operation is requested only when all the log sectors are consumed on an erase unit. This actually helps avoid frequent write and erase operations that would be requested by in-place updating or sequential logging method of traditional database servers.

Algorithm 1: Merge Operation

Input: B_o: an old erase unit to merge
Output: B: a new erase unit with merged content

procedure $Merge(B_o, B)$
1: allocate a free erase unit B
2: **for** *each data page* p *in* B_o **do**
3: **if** *any log record for* p *exists* **then**
4: $p' \leftarrow$ apply the log record(s) to p
5: write p' to B
else
6: write p to B
endif
endfor
7: erase and free B_o

When a merge operation is completed for the data pages stored in a particular erase unit, the content of the erase unit (*i.e.*, the merged data pages) is physically relocated to another erase unit in flash memory. Therefore, the logical-to-physical mapping of the data pages should be updated as well. Most flash memory devices store the mapping information persistently in flash memory, which is maintained as meta-data by the flash translation layer (FTL) [16, 18]. Note again that the mapping information needs to be updated only when a merge operation is performed, and the performance impact will be even less under the IPL design than traditional database servers that require updating the mapping information more frequently for all write operations.

4. PERFORMANCE EVALUATION

In this section, we analyze the performance characteristics of a conventional disk-based database server to expose the opportunities and challenges posed by flash memory as a replacement medium for magnetic disk. We also carry out a simulation study with the TPC-C benchmark to demonstrate the potential of the IPL approach for considerable improvement of write performance.

4.1 Disk-Based Server Performance

In this section, we analyze the performance characteristics of a conventional disk-based database server with respect to different types of storage media, namely, magnetic disk and flash memory.

4.1.1 Setup for Experiment

We ran a commercial database server on two Linux systems, each with a 2.0 GHz Intel Pentium IV processor and 1 GB RAM. The computer systems were identical except that one was equipped with a magnetic disk drive and the other with a flash memory storage device instead of the disk drive. The model of the disk drive was Seagate Barracuda 7200.7 ST380011A, and the model of the flash memory device was M-Tron MSD-P35 [11] (shown in Figure 3), which internally deploys Samsung K9WAG08U1A 16 Gbits SLC NAND flash. Both storage devices were connected to the computer systems via an IDE/ATA interface.

In order to minimize the interference by data caching and log-

Read Queries	Query processing time (sec)		Write Queries	Query processing time (sec)	
	Disk	Flash		Disk	Flash
Sequential (Q_1)	14.04	11.02	Sequential (Q_4)	34.03	26.01
Random (Q_2)	61.07	12.05	Random (Q_5)	151.92	61.76
Random (Q_3)	172.01	13.05	Random (Q_6)	340.72	369.88

Table 3: Read and Write Query Performance of a Commercial Database Server

Figure 3: MSD-P35 NAND Flash-based Solid State Disk

ging, the commercial database server was set to access both types of storage as a raw device, and no logging option was chosen so that most of IO activities were confined to data pages of a base table and index nodes. The size of a buffer pool was limited to 20 MBytes, and the page size was 8 KBytes by default for the database server.

A sample table was created on each of the storage devices, and then populated with 640,000 records of 650 bytes each. Since each data page (of 8 KBytes) stored up to 10 records, the table was composed of 64,000 pages. In case of the flash memory device, this table was spanned over 4,000 erase units, because each 128 KByte erase unit had sixteen 8 KByte data pages in it. The domain of the first two columns of the table was integer, and the values of the first two columns were given by the following formulas

$$\begin{aligned} col_1 &= \lfloor record\_id/160 \rfloor \\ col_2 &= record\_id \pmod{160} \end{aligned}$$

so that we could fully control data access patterns through B$^+$-tree indices created on the first two columns.

4.1.2 Read Performance

To compare the read performance of magnetic disk and flash memory, we ran the following queries Q_1, Q_2 and Q_3 on each of the two computer systems. Q_1 scans the table sequentially; Q_2 and Q_3 read the table randomly. The detailed description of the queries is given below.

- Q_1: scan the entire table of 64,000 pages sequentially.
- Q_2: pick 16 consecutive pages randomly and read them together at once; repeat this until each and every page of the table is read only once.
- Q_3: read a page each time such that two pages read in sequence are apart by 16 pages in the table. The id's of pages read by this query are in the following order: $0, 16, 32, \ldots, 63984, 1, 17, 33, \ldots$.

The response times of the queries measured in the wall clock time are presented in Table 3. In the case of disk, the response time increased as the access pattern changed from sequential (Q_1) to quasi-random (Q_2 and Q_3). Given the high seek and rotational latency of magnetic disks, this result was not surprising, because the more random the access pattern is, the more frequently the disk arm has to move. On the other hand, in the case of flash memory, the amount of increase in the response times was almost negligible. This result was also quite predictable, because flash memory has no mechanically moving parts nor mechanical latency.

4.1.3 Write Performance

To compare the write performance of magnetic disk and flash memory, we ran the following queries Q_4, Q_5 and Q_6 on each of the two computer systems. Q_4 updates all the pages in the table sequentially; Q_5 and Q_6 update all the pages in the table in a random order but differently. The detailed description of the queries is given below.

- Q_4: update each page in the entire table sequentially.
- Q_5: update a page each time such that two pages updated in sequence are apart by 16 pages in the table. The id's of pages updated by this query are in the following order: $0, 16, 32, \ldots, 63984, 1, 17, 33, \ldots$.
- Q_6: update a page each time such that two pages updated in sequence are apart by 128 pages in the table. The id's of pages updated by this query are in the following order: $0, 128, 256, \ldots, 63872, 1, 129, 257, \ldots$.

The response times of the queries measured in the wall clock time are presented in Table 3. In the case of disk, the trend in the write performance was similar to that observed in the read performance. As the access pattern became more random, the query response time became longer due to prolonged seek and rotational latency.

In the case of flash memory, however, a striking contrast was observed between the read performance and the write performance. As the access pattern changed from sequential to random, the update query response time became longer, and it was actually worse than that of disk for Q_6. As is discussed previously, the dominant factor of write performance for flash memory is how often an erase operation has to be performed.

In principle, each and every update operation can cause an erase unit to be copied elsewhere and erased. In practice, however, most of flash memory products are augmented with a DRAM buffer to avoid as many erase operations as possible. The MSD-P35 NAND flash solid state disk comes with a 16 MByte DRAM buffer, each one MByte segment of which can store eight contiguous erase units. If data pages are updated sequentially, they can be buffered in a DRAM buffer segment and written to the same erase unit at once. This was precisely what happened to Q_4. Due to the sequential order of updates, each of the 4000 erase units of the table was copied and erased only once during the processing of Q_4.

On the other hand, Q_5 and Q_6 updated data pages in a random order but differently. Each pair of pages updated by Q_5 in sequence were apart by 16 pages, which is equivalent to an erase unit. Since a total of eight consecutive erase units are mapped to a DRAM buffer segment of one MByte, an erase operation was requested every eight page updates (*i.e.*, a total of $64000/8 = 8000$ erases).

The pages updated by Q_6 were apart from the previous and the following pages by 128 pages, which is equivalent to a DRAM buffer segment of one MByte. Consequently, each page updated by Q_6 caused an erase operation (*i.e.*, a total of 64000 erases). This is the reason why Q_6 took considerably more time than Q_5, which in turn took more than Q_4.

4.2 Simulation with TPC-C Benchmark

In this section, we examine the write performance of the IPL approach and compare it with that of a disk-based database server for more realistic workload. We used a reference stream of the TPC-C benchmark, which is a representative workload for on-line transaction processing, and estimated the performance of a server managing database stored in flash memory, with and without the IPL features. As pointed out in Section 3, the IPL read operation may incur additional overhead to fetch log sectors from flash memory. However, due to its superior read performance of flash memory irrespective of access patterns, as shown in Table 3, we expect that the benefit from the improved write performance will outweigh the increased cost of read operations.

4.2.1 TPC-C Trace Generation

To generate a reference stream of the TPC-C benchmark, we used a workload generation tool called *Hammerora* [7]. Hammerora is an open source tool written in Tcl/Tk. It supports TPC-C version 5.1, and allows us to create database schemas, populate database tables of different cardinality, and run queries from a different number of simulated users.

We ran the Hammerora tool with the commercial database server on a Linux platform under a few different configurations, which is a combination of the size of database, the number of simulated users, and the size of a system buffer pool. When the database server ran under each configuration, it generated (physiological) log records during the course of query processing. In our experiments, we used the following traces to simulate the update behaviors of a database server with and without the IPL feature.

`100M.20M.10u`:	100 MByte database, 20 MByte buffer pool, 10 simulated users
`1G.20M.100u`:	1 GByte database, 20 MByte buffer pool, 100 simulated users
`1G.40M.100u`:	1 GByte database, 40 MByte buffer pool, 100 simulated users

Note that each of the traces contained update log records only, because the database server did not produce any log record for read operations. Nonetheless, these traces provided us with enough information, as our empirical study was focused on analyzing the update behaviors of a database server with flash memory.

4.2.2 Update Pattern of the TPC-C Benchmark

First, we examined the lengths of log records. Since the number of log records kept in memory by the IPL buffer manager is determined by the average length of log records and the size of a flash sector, which is typically 512 bytes, the average length of log records is an indicator suggesting how quickly an in-memory log sector becomes full and gets written to flash memory. Table 4 shows the average length of log records by the types of operations for the `1G.20M.100u` trace. Since the average length is no longer than 50 bytes, a log sector of 512 bytes can store up to 10 log records on average. This implies that an in-memory log sector will not be flushed to flash memory until its associated data page gets updated 10 times on average, unless the data page is evicted by a buffer replacement mechanism. In addition to the three types of physiological log records, the traces contain log records of physical page writes. For example, the `1G.20M.100u` trace contained 625527 log records of physical page writes in addition to the 784278 physiological log records.

Operation	occurrences	avg. length
Insert	86902 (11.08%)	43.5
Delete	284 (0.06%)	20.0
Update	697092 (88.88%)	49.4
Total	784278 (100.00%)	48.7

Table 4: Update Log Statistics of the `1G.20M.100u` Trace

Second, we examined the log reference locality by counting the number of log records that updated individual data pages. As shown in Figure 4(a), the distribution of update frequencies was highly skewed. We also examined the distribution of references in terms of how frequently individual data pages were physically written to the database. Note that the frequency distribution of physical page writes is expected to be correlated to the log reference locality above, but may be slightly different, because a data page cached in the buffer pool can be modified multiple times (generating multiple log records), until it is evicted and written to the database. We obtained the page write count for each data page from the traces, which contained the information of individual data pages being written to database, in addition to the update log records. Figure 4(b) shows the distribution of the frequency of physical page writes for the 2000 most frequently updated pages in the `1G.20M.100u` trace. The distribution of the write frequencies is also clearly skewed. In this case, the 2000 most frequently written pages (1.6% of a total of 128K pages in the database) were written 29% of the times (637823 updates).[3] We also derived the physical erase frequencies by mapping each page to its corresponding erase unit. Figure 4(c) shows the erase frequencies of erase units for the same `1G.20M.100u` trace.

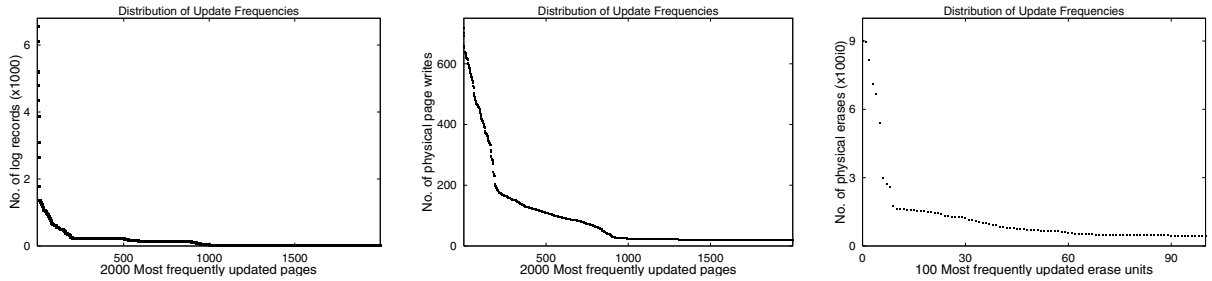
Third, we examined the temporal locality of data page updates by running a sliding window of length 16 through each trace of physical write operations. We counted the number of distinct pages within individual windows, and averaged them across the entire span of the trace. For the `1G.20M.100u` trace, the probability that 16 consecutive physical writes would be done for 16 distinct data pages was 99.9%. We can derive the similar analysis for erase units. The probability that 16 consecutive physical writes would be done for 16 distinct erase units was 93.1% (*i.e.*, on average 14.89 out of 16). Due to this remarkable lack of temporal locality, the update patterns of the TPC-C benchmark are expected to cause a large number of erase operations with flash memory.

4.2.3 Write Performance Estimation

In order to evaluate the impact of the IPL design on the performance of flash-based database servers, we implemented an event-driven simulator modeling the IPL buffer and storage managers as described in Section 3. This simulator reads log records from the TPC-C traces described in Section 4.2.1, and mimics the operations that would be carried out by the IPL managers according to the types of log records. The simulator was implemented in the C language, and its pseudo-code is shown in Algorithm 2.

There are four types of log records in the traces: three types of physiological log records (namely, insert, delete, and update) plus log records for physical writes of data pages. When a physiological

[3]This trend of skewedness appears to coincide with the TPC-C characterization of DB2 UDB traces [4]. The DB2 UDB traces contained both read and write references.



(a) Log reference locality by pages (b) Locality of physical page writes (c) Locality of physical erases

Figure 4: TPC-C Update Locality in terms of Physical Writes (`1G.20M.100u` Trace)

Algorithm 2: Pseudo-code of the IPL Simulator

Input: $\{L_i\}$: a trace from the TPC-C benchmark

procedure $Simulate(\{L_i\})$
1: **for** *each log record* L_i **do**
2: **if** L_i.*opcode* $\in$ {*insert, delete, update*} **then**
3: **if** *log_count(*L_i*.pageid)* $\geq \tau_s$ **then**
4: generate a sector-write event
5: log_count(L_i.pageid) $\leftarrow 0$
endif
6: log_count(L_i.pageid)++
else
// L_i is a log record of physical page write
7: generate a sector-write event
8: log_count(L_i.pageid) $\leftarrow 0$
endif
endfor
event_handler $SectorWrite(\{L_i\})$
9: eid $\leftarrow$ erase unit id of the sector
10: **if** *logsector_count(eid)* $\geq \tau_e$ **then**
11: global_merge_count++
12: logsector_count(eid) $\leftarrow 0$
endif
13: logsector_count(eid)++
14: global_sector_write_count++

log record is accepted as input, the simulator mimics adding the log record into the corresponding in-memory log sector by incrementing the log record counter of the particular sector.[4] If the counter of the in-memory log sector has already reached the limit (denoted by τ_s in the pseudo-code), then the counter is reset, and an internal *sector-write* event is created to mimic flushing the in-memory log sector to a flash log sector (Lines 3 to 5).

When a log record of physical page write is accepted as input, the simulator mimics flushing the in-memory log sector of the page being written to the database by creating an internal sector-write event (Lines 7 to 8). Note that the in-memory log sector is flushed even when it is not full, because the log record indicates that the corresponding data page is being evicted from the buffer pool.

Whenever an internal sector-write event is generated, the simulator increments the number of consumed log sectors in the corresponding erase unit by one. If the counter of the consumed log sectors has already reached the limit (denoted by τ_e in the pseudo-code), then the counter is reset, and the simulator increments the global counter of merges by one to mimic the execution of a merge operation and to keep track of total number of merges (Lines 10 to 13). The simulator also increments the global counter of sector writes by one to keep track of total number of sector writes.

Trace	No of update logs	No of sector writes
`100M.20M.10u`	79136	46893
`1G.40M.100u`	784278	594694
`1G.20M.100u`	785535	559391

Table 5: Statistics of Log Records and Sector Writes

When we ran the IPL simulator through each of the traces, we increased the size of the log region in each erase unit from 8 KBytes to 64 KBytes by 8 KBytes to observe the impact of the log region size on the write performance. The IPL simulator returns two counters at the completion of analyzing a trace, namely, the total number of sector writes and the total number of erase unit merges. The number of sector writes is determined by the update reference pattern in a given trace and the buffer replacement by the database server, independently of the size of a log region in the erase units. Table 5 summarizes the total number of references in each trace and the number of sector writes reported by the IPL simulator for each trace. Note that, even with the relatively small sizes chosen for the buffer pool, which causes in-memory log sectors to be flushed prematurely, the number of sector writes was reduced by a non-trivial margin compared with the number of update references.

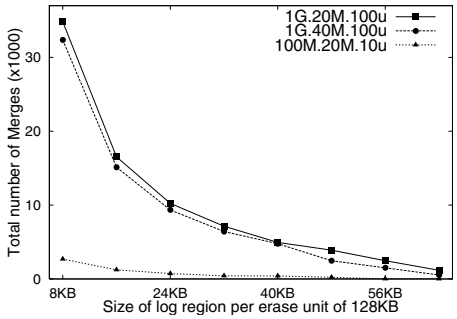
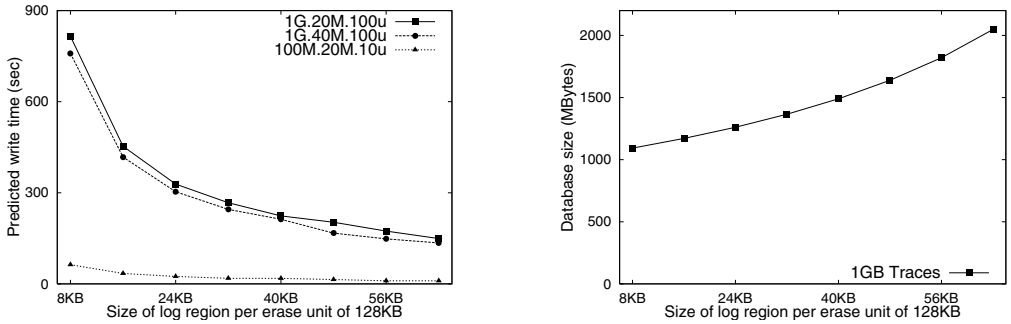


Figure 5: Simulated Merge Counts

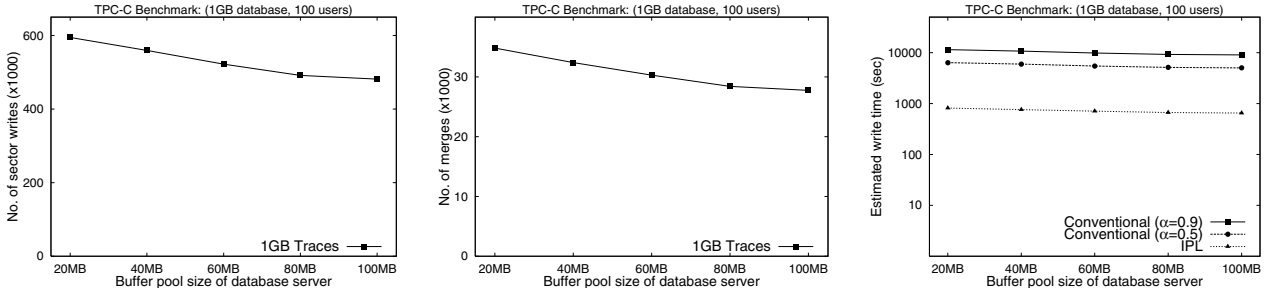
On the other hand, the number of merges is affected by the size of a log region. Figure 5 shows the number of merges for each of the three traces, `100M.20M.10u`, `1G.20M.100u` and `1G.40M.100u`, with a varying size of the log region. As the size of the log region increased, the number of merge operations decreased dramatically

[4] Since the traces do not include any record of physical page reads, we can not tell when the page is fetched from the database, but it is inferred from the log record that the page referenced by the log record must have been fetched before the reference.



(a) Estimated write time (b) Storage usage for database

Figure 6: Estimated Write Performance and Space Overhead



(a) Total number of sectors written (b) Total number of merges performed (c) Estimated write time

Figure 7: Performance Trend with Varying Buffer Sizes (1GB DB and 100 users; 8KB log sectors)

at the cost of increased storage space for database. As we observed in Figure 4(a), the distribution of update frequencies was so skewed that a small fraction of data pages were updated much more frequently than the rest, before they were evicted from the buffer pool. This implies that the erase units containing the hot pages exhausted their log regions rapidly, and became prone to merge operations very often. For the reason, the more flash log sectors were added to an erase unit, the less frequently the erase unit was merged (*i.e.*, copied and erased), because more updates were absorbed in the log sectors of the erase unit.

To understand the performance impact of the reduced merge operations more realistically, we used the following formula to estimate the time that an IPL-enabled database server would spend on performing the insert, delete and update operations in the TPC-C traces.

$$t_{IPL} = (\#\text{ of sector writes}) \times 200\mu s + (\#\text{ of merges}) \times 20ms$$

The average time (200μs) spent on writing a flash sector is from Table 1.[5] The average time (20 ms) spent on merging an erase unit can be calculated from Table 1 by adding the average times taken for reading, writing and erasing an erase unit.[6]

Figure 6(a) shows the write time estimated by the t_{IPL} formula for each of the three traces. The performance benefit from the increased number of log sectors was evident, but the space overhead, as shown in Figure 6(b), was not trivial. As the price of flash memory is expected to drop steadily, however, we expect that the increase throughput will outweigh the increased cost for storage.

[5]Writing a 512-byte sector takes the same amount of time as writing a 2-KByte block on large block NAND flash devices.

[6]This average time of merge coincides with the measurement given by Birrel *et al.* [2].

We also examined how the performance of a database server with the IPL features was affected by the capacity of the system buffer pool. To do this, we generated three additional traces called `1G.60M.100u`, `1G.80M.100u` and `1G.100M.100u` by running the database server with a buffer pool of different capacities, namely, 60 MB, 80 MB and 100 MB. Figures 7(a) and 7 (b) show the total number of sector writes and the total number of merges with a varying capacity of the buffer pool. Not surprisingly, as the buffer capacity increased, the total number of pages replaced by the buffer manager decreased. Consequently, Figure 7(c) shows the similar trend in the estimated write time, as the buffer capacity increases.

In addition, Figure 7(c) shows the expected write time that a conventional database server would spend without the IPL features. The write time of this case was estimated by the following formula,

$$t_{Conv} = (\alpha \times \#\text{ of page writes}) \times 20ms$$

where the parameter α denotes the probability that a page write will cause the container erase unit to be copied and erased. In Figure 7(c), the value of α was set to 90% and 50%. (Recall that in Section 4.2.2 the probability of 16 consecutive physical writes being done to 16 distinct erase units was 93.1%.) Even when the value of α was arbitrarily adjusted from 90% to 50%, the write performance of the IPL server was an order of magnitude better that that of the conventional server. Note that the y-axis of Figure 7(c) is in the logarithmic scale.

4.3 Summary

High density flash memory has been successfully adopted by personal media players, because flash memory yields excellent read and write performance for sequential access patterns. However, as shown in Table 3, the write performance of flash memory dete-

riorates drastically, as the access pattern becomes random, which is quite common for the OLTP-type applications. The simulation study reported in this section demonstrates that the IPL strategy can help database servers overcome the limitations of flash memory and achieve the best attainable performance.

5. SUPPORT FOR RECOVERY

In this section, we discuss how the basic IPL design can be augmented to support transactional database recovery. The IPL buffer and storage managers, as described in Section 3, rely on logging updates temporarily in main memory and persistently in flash memory in order to overcome the erase-before-write limitation of flash memory. The update logs of IPL can also be used to realize a lean recovery mechanism for transactions with the minimal overhead during the normal processing such that the cost of system recovery can be reduced considerably. This will help minimize the memory foot-print of database servers particularly for mobile or embedded systems.

5.1 Additional Logging and Data Structure

For the support of transactional database recovery, it is necessary to adopt the conventional system-wide logging maintained typically in a separate storage, for keeping track of the start and end (*i.e.*, commit or abort) of transactions. Like the *transaction log* of the Postgres No-Overwrite Storage [25], the only purpose of this system-wide logging is to determine the status of transactions at the time of system failure during the recovery. Since no additional log records (other than those by the in-page logging) are created for updates, the overall space and processing overhead is no worse than that of conventional recovery systems.

In addition to the transaction log, *a list of dirty pages* in the buffer pool can be maintained in memory during the normal processing, so that a committing transaction or an aborting transaction (not by the system failure) can quickly locate the in-memory log sectors containing the log records added by the transaction.

5.2 Transaction Commit

Most disk-based database systems adopt a *no-force* buffer management policy for performance reasons [13]. With a force policy, all the data pages modified by a committing transaction would have to be forced out to disk, which might often lead to random disk accesses for an increased volume of data rather than just flushing the log tail sequentially to a stable storage. With a no-force policy, only the log tail is forced to a stable storage. Consequently, however, data pages resident on disks may not be current. Thus, when a system failure occurs, REDO recovery actions should be performed for committed transactions at the system restart.

To adopt a no-force policy for the IPL design, the corresponding in-memory log sectors need to be written to flash memory when a transaction commits. Note that, in the basic IPL design as described in Section 3.2, an in-memory log sector is written to flash memory when it becomes full or its associated data page is evicted from the buffer pool. In addition to that, for the sake of transactional recovery, the IPL buffer manager has to force out an in-memory log sector to flash memory, if it contains at least one log record of a committing transaction.

Unlike a log tail sequentially written to a stable storage by disk-based database systems, the IPL in-memory log sectors are written to non-consecutive locations of flash memory, because they must be co-located with their corresponding data pages. The access pattern is thus expected to be random. With no mechanical latency of flash memory, however, the cost of writing the in-memory log sectors to flash memory will be just about the same as the cost of writing them sequentially, and this process will cause no substantial performance degradation at commit time.

We claim that, even with the no-force policy, the IPL design does not require REDO actions explicitly for committed transactions at the system restart. Rather, any necessary REDO action will be performed implicitly as part of normal database processing, because the redefined IPL read applies log records on the fly to data pages being fetched from flash memory, and all the changes made by a committed transaction are available in the log records in flash memory. In other words, under the IPL design, the materialized database [13] consists not only of data pages but also of their corresponding log records.

5.3 Transaction Abort

When an individual transaction T aborts (not by a system failure), T's log records that still remain in the in-memory log sectors can be located via the list of dirty pages maintained in memory, removed from the in-memory log sectors, and de-applied to the corresponding data pages in the buffer pool. Some of T's log records, however, may have already been written to flash log sectors by the IPL buffer manager. To make the matter even more complicated, the IPL merge operation described in Section 3.3 creates a new version of data pages in an erase unit by applying the log records to the previous version, and frees the old erase unit in flash memory. Since, when a merge is completed, the log records that were stored in the old erase unit are abandoned, it would be impossible to rollback the changes made by an aborting transaction without providing a separate logging mechanism for UNDO actions.

To cope with this issue, we propose a *selective merge* operation instead of the regular *merge* so that we can take advantage of the in-page logging and simplify the UNDO recovery for aborted transactions or incomplete transactions at the time of system crash. The idea of the selective merge is simply to keep log records from being applied to data pages if the corresponding transactions are still active when a merge is invoked. With this selective merge, we can always rollback the changes made by uncommitted transactions just by discarding their log records, because no changes by those transactions are applied to any data page in flash memory until they commit.

When a selective merge is invoked for a particular erase unit, the IPL storage manager inspects each log record stored in the erase unit, and performs a different action according to the status of the transaction responsible for the log record. If the transaction is committed, then the log record is applied to a relevant data page. If the transaction is aborted, then the log record is simply ignored. If the transaction is still active, the log record is moved to the log sector in a newly allocated erase unit. Obviously, when multiple log records are moved to a new erase unit, they are compacted into the fewest number of log sectors.

There is a concern, however, that may be raised when too many log records need to be moved to a new erase unit. For example, if all the transactions associated with the log records are still active, then none of the log records will be dropped when they are moved to a new erase unit. The problem in such a case is that the newly merged erase unit is prone to another merge in the near future due to the lack of free slots in the log sectors, which will cause additional write and erase operations.

One way of avoiding such a thrashing behavior of the selective merge is to allow an erase unit being merged to have overflow log sectors allocated in a separate erase unit. When a selective merge is triggered by an in-memory log sector being flushed to an erase unit (say E), the IPL storage manager estimates what fraction of log records would be carried over to a new erase unit. If the frac-

Algorithm 3: Selective Merge Operation

Input: B_o: an old erase unit to merge
Output: B: a new erase unit with merged content

procedure $Merge(B_o, B)$
if *carry-over-fraction* $> \tau$ **then**
    // the log sector is added to an overflow log area
    return B_o as B
endif
allocate a free erase unit B
for *each data page* p *in* B_o **do**
    if *any committed log record for* p *exists* **then**
        $p' \leftarrow$ apply the $committed$ log record(s) to p
        write p' to B
    else
        write p to B
    endif
endfor
compact and write all $active$ log records to B
erase and free B_o

tion is over a certain threshold value τ, the erase unit E remains intact, but instead the in-memory log sector is written to a flash log sector in a separate erase unit allocated as an overflow area. The algorithmic description of the selective merge operation is presented in Algorithm 3.

With the selective merge replacing the regular merge, it is not necessary to explicitly perform UNDO actions for aborted or incomplete transactions. Rather, any necessary UNDO action will be performed implicitly as part of normal database processing by preventing any change made by aborted or incomplete transactions from being merged to data pages. Note that the log records by aborted or incomplete transactions are not explicitly invalidated by the IPL storage manager in order not to incur any unnecessary IO, but instead dropped by selective merge operations during the normal processing, and eventually garbage-collected and erased.

5.4 System Restart

As described above, the IPL storage manager maintains data pages and their log records in such a way that a consistent database state with respect to all the committed transactions can always be derived from data pages and log records. Consequently, both the REDO and UNDO actions can be performed implicitly, just as they are done during the normal processing.

When the database server is recovered from a system failure, the transaction log (described in Section 5.1) is examined to determine the status of transactions at the time of the failure. For a transaction whose commit or abort record appears in the transaction log, no recovery action needs to be performed. For a transaction that was active at the time of failure, an abort record should be added to the transaction log, so that any change made by this transaction can be rolled back by the subsequent processing of the IPL storage manager.

6. RELATED WORK

Most commercial database systems rely on the in-place updates and the no-force buffer replacement. Without the no-force policy, the commit-time overhead may be high, because scattered random writes are required to propagate all the changes made by each and every committing transaction. With the no-force policy, on the other hand, whenever a transaction commits, the log tail has to be written to a stable storage in order to ensure the durability of transactions. Since the log tail is always written in the sequential manner, the commit-time overhead will be minimal even with magnetic disk drives with high mechanical latency. Under the recovery mechanism supported by the IPL scheme (Section 5), when a transaction commits, its log records still cached in the buffer pool are written to corresponding log sectors in flash memory in the scattered fashion. Due to no mechanical latency of flash memory, however, small random writes can be processed efficiently as long as costly erase operations are not involved [8]. Since the IPL scheme can keep the the number of merge (*i.e.*, copy and erase) operations to the minimum, the commit-time overhead is likely to be still low.

As large and cheap magnetic disks were available in the mid 1990s, the concept of "no-overwrite storage manager" was proposed for Postgres [25]. The main idea was, instead of overwriting data in disk, to store the historical delta records of updates in addition to the original contents of data. By taking the no-overwrite policy, it can travel the history of changes for a data item, and more importantly, recover from database failures very quickly. In this respect, it is similar to our in-page logging. However, the no-overwrite storage manager of Postgres must force to disk at commit time all pages written by a transaction by random IO. Therefore, it was retrospected that the no-overwrite storage would become a viable storage option only with stable main memory (*e.g.*, FeRAM) [26].

In the late 90s, PicoDBMS [3] was developed for EEPROM, which was then the major storage media for Smartcard in the late 1990s. The main bottleneck of EEPROM is its write performance. While the read time per word is about 60 $\sim$ 250 ns, the write time per word is about 1 $\sim$ 5 ms. Since EEPROM allows overwrite unlike flash memory, PicoDBMS was built on the update-in-place approach. If flash memory is used instead of EEPROM, PicoDBMS will suffer the same performance degradation as the traditional disk-based database servers (see Table 3). Besides, PicoDBMS frequently uses pointer-based data accesses in order to minimize the size of database, and to take advantage of the fast read accesses of EEPROM. However, the read speed of NAND flash memory is not as fast, compared with EEPROM. Therefore, the intensive use of pointer-based data access will be another performance bottleneck for flash-based systems.

Finally, we would like to discuss the limitations of flash translation layers (FTL) for database workloads. The main goal of FTL's is to minimize erase operations even for small random updates. The pattern of random writes typically dealt with by a file system is quite different from that of database workload. Specifically, in a file system, most random writes are required for meta-data such as FAT (file allocation table) and I-node map, and the writes are scattered over only a very limited address space (typically less than several megabytes) [18]. Therefore, this type of random writes can be efficiently handled by the LSF-like techniques adopted by most FTL's. In contrast, the write patterns typical in database workloads are scattered randomly over a large address space (usually more than several gigabytes). Consequently, most existing FTL's are not well suited for processing database workload.

Table 6 summarizes the representatives of the previous work related to the in-page logging approach proposed in this paper. This table classifies database storage techniques with respect to the data update policy (*i.e.*, in-place vs. no in-place) and the data access latency (*i.e.*, with or without mechanical latency).

7. CONCLUSIONS

The evidence that high-density flash memory can replace magnetic disks for a wide spectrum of computing platforms is clear and

	In-place update	No in-place update
Mechanical Latency	Traditional DB Storage and recovery [10, 20] (Disk)	Postgres Storage [25] (Disk)
No mechanical latency	PicoDBMS [3] (EEPROM)	In-page logging (Flash Memory)

Table 6: Classification of Database Storage Techniques

present. While multimedia applications tend to access large audio or video files sequentially, database applications tend to read and write data in small pieces in the scattered and random fashion. Due to the erase-before-write limitation of flash memory, the traditional database servers designed for disk-based storage systems will suffer seriously deteriorated write performance.

To the best of our knowledge, it is the first time we expose the opportunities and challenges posed by flash memory for the unique workload characteristics of database applications, by running a commercial database server on a flash-based storage system. The in-page logging (IPL) proposed in this paper has demonstrated its potential for considerable improvement of write performance for OLTP-type applications by exploiting the advantages of flash memory such as no mechanical latency and high read bandwidth. Besides, the IPL design can be extended to realize a lean recovery mechanism for transactions.

Acknowledgment

The authors thank Dr. Young-Hyun Bae of M-Tron Corp. for providing the company's SSD product prototype and its technical information, and Mr. Jae-Myung Kim for assisting with the experiments.

8. REFERENCES

[1] Brad Adelberg, Ben Kao, and Hector Garcia-Molina. Database Support for Efficiently Maintaining Derived Data. In *the 5th International Conference on Extending Database Technology*, pages 223–240, Avignon, France, March 1996.

[2] Andrew Birrel, Michael Isard, Chuck Thacker, and Ted Wobber. A Design for High-Performance Flash Disks. Technical Report MSR-TR-2005-176, Microsoft Research, December 2005.

[3] Christophe Bobineau, Luc Bouganim, Philippe Pucheral, and Patrick Valduriez. PicoDBMS: Scaling Down Database Techniques for the Smartcard. In *Proceedings of the 26th VLDB Conference*, pages 11–20, Cairo, Egypt, September 2000.

[4] R. Bonilla-Lucas et al. Characterization of the Data Access Behavior for TPC-C Traces. In *Performance Analysis of Systems and Software*, pages 115–122, 2004.

[5] Hui Dai, Michael Neufeld, and Richard Han. ELF: An Efficient Log-Structured Flash File System for Micro Sensor Nodes. In *The Second International Conference on Embedded Networked Sensor Systems (SenSys'03)*, pages 176–187, Baltimore, MD, USA, November 2004.

[6] Fred Douglis, Ramon Caceres, Frans Kaashoek, Kai Li, Brian Marsh, and Joshua A. Tauber. Storage Alternatives for Mobile Computers. In *Proceedings of the USENIX 1st Symposium on Operating Systems Design and Implementation (OSDI-94)*, Monterey, CA, USA, November 1994.

[7] Julian Dyke and Steve Shaw. *Pro Oracle Database 10g RAC on Linux: Installation, Administration, and Performance*. Apress, 2006.

[8] Eran Gal and Sivan Toledo. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys*, 37(2):138–163, June 2005.

[9] Goetz Graefe. Write-Optimized B-Trees. In *Proceedings of the 30th VLDB Conference*, pages 672–683, Toronto, Canada, September 2004.

[10] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[11] MTRON Media Experts Group. MSD-P Series Production Specification. Technical Report Version 0.7 sv, MTRON Co. Ltd., October 2006.

[12] Mark Hachman. New Samsung Notebook Replaces Hard Drive With Flash. http://www.extremetech.com, May 2006.

[13] Theo Härder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Survey*, 15(4):287–317, 1983.

[14] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. I/O Reference Behavior of Production Database Workloads and the TPC Benchmarks - An Analysis at the Logical Level. *ACM Transactions on Database System*, 26(1):96–143, 2001.

[15] Atsushi Inoue and Doug Wong. NAND Flash Applications Design Guide. Technical Report Revision 2.0, Toshiba America Electronic Components, Inc., March 2004.

[16] Intel. Understanding the Flash Translation Layer (FTL) Specification. Application Note AP-684, Intel Corporation, December 1998.

[17] Gye-Jeong Kim, Seung-Cheon Baek, Hyun-Sook Lee, Han-Deok Lee, and Moon Jeung Joe. LGeDBMS: A Small DBMS for Embedded System with Flash Memory. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 1255–1258. ACM, 2006.

[18] Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, and Yookun Cho. A Space-Efficient Flash Translation Layer for CompactFlash Systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

[19] Katsutaka Kimura and Takashi Kobayashi. Trends in High-Density Flash Memory Technologies. In *IEEE Conference on Electron Devices and Solid-State Circuits*, pages 45–50, Hong Kong, December 2003.

[20] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[21] Linda Dailey Paulson. Will Hard Drives Finally Stop Shrinking? *IEEE Computer*, 38(5):14–16, May 2005.

[22] Mendel Rosenblum. *The Design and Implementation of a Log-Structured File System.* PhD thesis, UC Berkeley, 1991.

[23] Mendel RosenBlum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *the 13th Symposium on Operating System Principles*, pages 1–15, Pacific Grove, CA, September 1991.

[24] Rajkumar Sen and Krithi Ramamritham. Efficient Data Management on Lightweight Computing Devices. In *Proceedings of the 21st Inter. Conference on Data Engineering*, Tokyo, Japan, April 2005.

[25] Michael Stonebraker. The Design of the Postgres Storage System. In *Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 289–300. Morgan Kaufmann, 1987.

[26] Michael Stonebraker and Greg Kemnitz. The Postgres Next Generation Database Management System. *Communications of the ACM*, 34(10):78–92, Oct 1991.