JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 27, 111-127 (2011)

IPL B$^+$-tree for Flash Memory Database Systems*

GAP-JOO NA, BONGKI MOON+ AND SANG-WON LEE**
School of Information and Communication Engineering
Sungkyunkwan University
Suwon, 440-746 Korea
+*Department of Computer Science*
University of Arizona
Tucson, AZ 8572, U.S.A.

Recently, the in-page logging (IPL) scheme has been proposed to improve the overall write performance of flash memory by avoiding costly erase operations that would be caused by small random write requests common in database applications. In this paper, we identify the problems inherent in the existing design of disk-based B$^+$-tree index, and present the design and implementation of the IPL B$^+$-tree. In this paper, in order to prove the concept of IPL to be a viable and effective solution for flash memory, we show the superior performance of the IPL B$^+$-tree index by running it on a real hardware prototype. We then show the IPL B$^+$-tree index outperforms traditional B$^+$-tree index running on top of an FTL by a factor of two to fifteen. In addition, we introduce the concept of FTL dependency: many existing B$^+$ tree schemes for flash memory could not control FTL so that their performance might be heavily dependent on the underlying FTL. In contract, IPL does not suffer from such FTL dependency because it does not assume the underlying FTL.

Keywords: flash memory, database, B$^+$-tree, FTL, index structure

1. INTRODUCTION

NAND flash memory has become the major persistent data storage medium for mobile and embedded devices (*e.g.* cell phones, USB Disk Drive, mp3 player) because of its low access latency, low power consumption, and high shock resistance. On the other hand, due to its erase-before-write limitation, it is also known that flash memory exhibits poor write performance especially when write operations are requested in a random order.

Random write is a fairly common access pattern in many database applications. An index structure is an essential component in database applications to search for or locate data objects from a large database, but it tends to make data access pattern more random. Beside, when data records are inserted, updated or deleted in the database, the size of updates made to the index structure is very small, typically in the range of 10s to 100s bytes. Consequently, the best attainable performance may not be obtained with current disk-based index structures and algorithms over a general purpose flash translation layer (FTL) [1], which emulates the functionality of the hard disks by providing a block device interface. Therefore, in order to make the best use of flash memory, it is crucial to design elaborate flash-aware data structures and algorithms.

In this paper, we propose a novel index structure tailored to the NAND flash mem-

Received July 7, 2009; revised September 15 & December 18, 2009; accepted March 8, 2010.
Communicated by Tei-Wei Kuo.
* This research was partly supported by MKE, Korea under ITRC NIPA-2010-(C1090-1021-0008) and by MEST, Korea under NRF Grants No. 2010-0025649 and 2010-0026511. This work was also sponsored in part by the U.S. National Science Foundation Grant IIS-0848503. The authors assume all responsibility for the contents of the paper.
** Corresponding author.

ory, called In-Page Logging B^+-tree (IPL B^+-tree). While the IPL B^+-tree index is functionally equivalent to the traditional B^+-tree, the IPL B^+-tree index can also reduce the number of overwrites in B^+-tree index by taking advantage of the IPL scheme. The in-page logging (IPL) scheme [2] attempts to improve the overall write performance of flash memory by avoiding the costly 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, 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 previous work on designing flash-aware B^+-Tree indexes [3-5] has been done on top of existing FTLs [6-8]. Therefore, the performance of the flash-aware indexes depends heavily on the characteristics of the chosen FTL. Furthermore, since their design relies on a large RAM buffer space, additional concerns are raised such as cost effectiveness and high capacity battery backup for destaging data upon an accidental power-off. In contrast, the IPL B^+-tree index uses a raw flash device interface instead of a block device interface. Therefore, the IPL B^+-tree index can perform well constantly on any flash memory platform without relying on a particular FTL. In fact, our approach is more exuberant than the previous approaches in utilizing the unique characteristics of flash memory, because the performance of the latter approach is highly dependent on the underlying FTL they assume. Our experimental results show that the IPL B^+-tree index can outperform traditional B^+-tree index and previous flash-based index running on top of FTL by a factor of two to fifteen.

The key contributions of this work are as follows.

- We prove the concept of IPL to be a viable and effective solution for flash memory database systems by implementing IPL based B^+-tree over an embedded board environment.
- We show that the traditional disk-based B^+-tree index is not suitable for the flash storage devices and their performance is heavily relying on their FTL algorithms.
- Also, we show the potential problems of the existing flash-based B^+-tree approaches assuming the underlying FTL, such as block merge operations and garbage collections.
- We demonstrate that the IPL scheme is very suitable for B^+-tree index structure through the experimental result.

The remainder of this paper is organized as follows. Section 2 discusses the background and the motivation of this work. In section 3, we propose the design and implementation of IPL B^+-tree index, and explain some design considerations in applying IPL concept to B^+-tree index. In section 4, we evaluate the performance of the IPL B^+-tree index and show the problems of previous work through three experiments. Finally, we conclude in section 5.

2. BACKGROUND AND MOTIVATION

2.1 S Characteristics of Flash Memory and Flash Translation Layer

A NAND flash memory consists of a set of blocks and a block consist of a set of

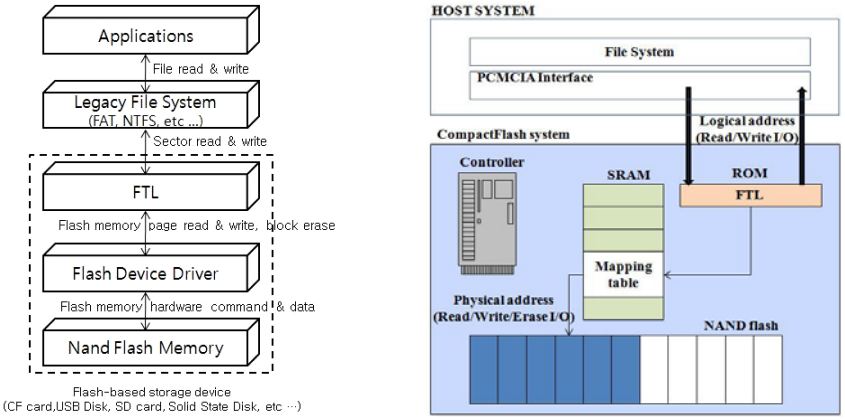
pages [9]. Currently the most popular page size is 2KByte and the block size is 128 KByte consisting of 64 pages. Without loss of generality, we assume that the flash memory means NAND type and the each units of the flash memory have the same size with the size mentioned above throughout this paper.

NAND flash memory has several characteristics compared to a magnetic disk. In contrast with hard disk, there are three basic operations, read, write, and erase, and the latency of each operation is asymmetric, as is shown in Table 1 [2]. The basic unit of read/write operation is page, and the unit of erase operation is block. In principle, the unit of write operation is page, but recently most flash memory allow also sector write (512Byte). On the other hand, since the flash memory does not allow in-place update, a write operation must be preceded by an erase operation, which has the longest access latency among the operations. Moreover erase operation can only be performed in a much larger unit than the write. This means that in order to update even a single byte, a time-consuming erase operation must be performed before overwriting, and additional read and write operations are required for restoring a large amount of data.

Table 1. NAND flash vs. disk.

Medium	Access Time		
	Read	Write	Erase
Magnetic disk	12.7ms (2KB)	13.7ms (2KB)	N/A
NAND flash	80μs (2KB)	200μs (2KB)	1.5ms (128KB)

Disk: Seagate Barracuda 7200.7 ST380011A, average access times including seek and rotational delay
NAND flash: Samsung K9WAG08U1A 16 Gbits SLC NAND



(a) Flash translation layer. (b) Internal organization of a compact flash.

Fig. 1. Flash translation layer.

In order to use NAND flash memory as a storage device, we should redesign current application to be suitable for its particular interface. Otherwise we can use the intermediate software between host application and flash memory, called Flash Translation Layer (FTL), which provides a block device interface to the user by emulating the operations of hard disk drives using flash memory. With FTL, current applications can be used

without any modification. The key role of the FTL is to translate the logical sector address used by the host into the physical flash address used by flash memory. As shown in Fig. 1 (a), FTL is located between file system and flash device driver, and translates the sector operation to the page or block operation. Typical flash storage devices, such as USB disk, CF, SD and SSD, are equipped with the FTL as shown in Fig. 1 (b). The controller of the storage device runs the FTL, which is usually stored in ROM, to operate flash memory.

2.2 Motivation

The database application commonly requires random write, which is not good for flash memory. Furthermore the write unit of the B^+-tree index structure is very small, so that we cannot obtain the best attainable performance of flash memory. If we use the index structure with the naïve FTL, the write performance of the index structure is even worse than that of hard disk. Therefore we need to revisit flash-aware database storage schemes, which manage the flash memory directly, instead of relying on general FTLs.

Recently, a database storage model, named In-Page Logging (IPL) scheme, dedicated to NAND flash memory has been proposed [2]. In IPL scheme, 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.

Since flash memory, in contrast to the hard disk, comes with no mechanical components, there is no considerable performance penalty arising from scattered writes [10], and there is no compelling reason to write log records sequentially either. Therefore, under the in-page logging approach, a data page and its log records are co-located in the same physical location of flash memory, specifically, in the same erase unit. 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 efficiently generated under this approach. Although the amount of data to be read will increase in proportion to the number of log records relevant to the data page, it will still be a sensible tradeoff for the reduced write and erase operations particularly considering that a read operation is typically at least twice faster than a write operation for flash memory. Consequently, the IPL approach can improve the overall write performance considerably.

In the previous work [2], the potential benefits from the IPL scheme in online transactional database applications have only been evaluated via simulations. Besides, the IPL scheme has not been evaluated for indexed accesses. This paper proposes a novel index structure called IPL B^+-tree, based on IPL storage scheme. In this paper, we show the problem of current B^+-tree index system on FTL and demonstrate the superiority of the IPL B^+-tree index, through the experimental evaluation. In addition, this paper lays the foundation work for the implementation of a flash-based database system.

3. IPL B^+-TREE INDEX SYSTEM

3.1 Design of IPL B^+-tree

The key feature of the IPL B^+-tree is to co-locate a tree node and its log records in

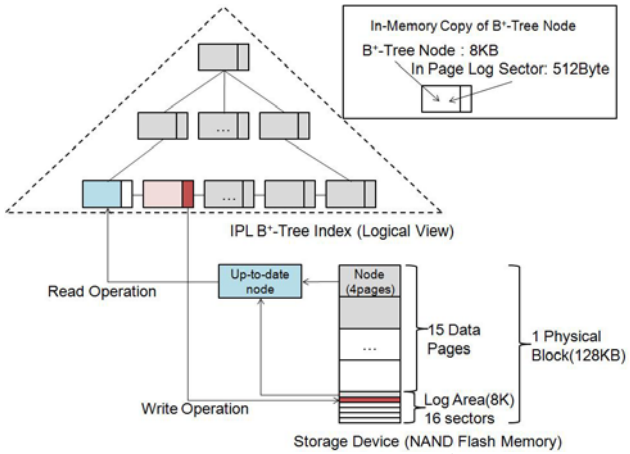


Fig. 2. Overall structure of IPL B+-tree.

the same physical flash block (or erase unit) so that its up-to-date tree node can be re-created efficiently without scanning a large number of log sectors. Fig. 2 illustrates the overall structure of IPL B+-tree, which adopts the in-page logging approach.

In the IPL B+-tree, an in-memory copy of each tree node can be associated with a small in-memory log sector. When an insertion or a deletion operation is performed on a tree node, the in-memory copy of the tree node is updated just as done by traditional B+-tree indexes. In addition, a physiological log record is added to the in-memory log sector associated with the tree node. An in-memory log sector is allocated on demand when a tree node becomes dirty, and released when the log records are written to a log sector in flash memory.

The log records in an in-memory log sector are written to flash memory either when the in-memory log sector becomes full or when a dirty tree node is evicted from the buffer pool. When a dirty tree node is evicted, it is not necessary to write the content of the dirty tree node back to flash memory, because all of its updates are already saved in the form of log records in flash memory. Thus, the previous version of the tree node remains intact in flash memory, but is just augmented with update log sectors.

When an in-memory log sector is flushed to flash memory, its content is written to a flash log sector in the erase unit which its corresponding tree node belongs to. In order to support this operation, each erase unit of flash memory is divided into two segments – one for tree nodes and the other for log sectors.

In the current implementation of IPL B+-tree, as is shown at the bottom of Fig. 2, an erase unit of 128 KBytes is divided into 15 data pages of 8 KBytes each and 16 log sectors of 512 bytes each. When an erase unit runs out of free log sectors, the tree nodes and log sectors in the erase unit are merged into a new erase unit.

3.2 IPL B+-tree Engine

As is illustrated in Fig. 3, the IPL B+-tree engine consists of three components: (1) B+-tree operator module, (2) page operator module, and (3) IPL manager. The B+-tree operator module processes traditional B+-tree operations such as insertion, deletion and

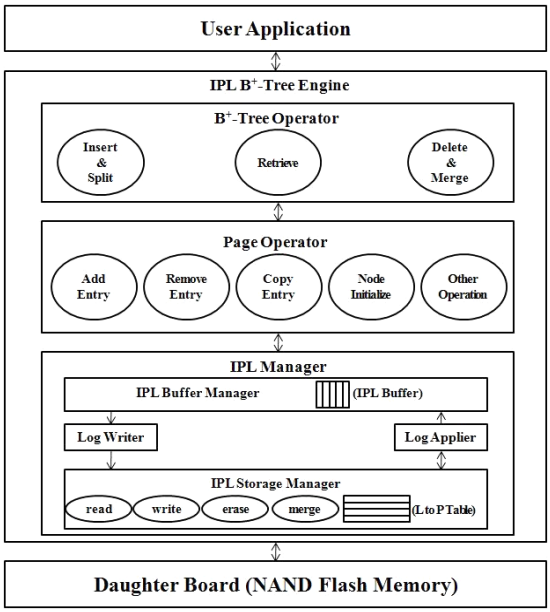


Fig. 3. IPL B^+-tree engine.

search. If a single operation involves more than one tree node (*i.e.* node split), this is divided into multiple single node requests, each of which is processed by the page operator module. For each single node request, the page operator module adds its physiological log record to the corresponding in-memory log sector.

The IPL manager, which is the most important component of the IPL B^+-tree engine, provides the in-page logging mechanism for B^+-tree indexes. The IPL manager consists of four internal components: (1) IPL buffer manager, (2) IPL storage manager, (3) Log writer, and (4) Log applier. IPL buffer manager maintains the in-memory tree nodes and their in-memory log sectors in an LRU buffer pool. When an in-memory log sector becomes full and needs to be flushed to flash memory, the IPL buffer manager determines whether the log area of the corresponding erase unit in flash memory can accommodate the in-memory log sector. If it does, the in-memory log sector is written to flash memory. Otherwise, a block merge operation request is sent to the IPL storage manager.

When an in-memory tree node is to be evicted from the buffer pool, the IPL buffer manager invokes the Log writer to write its log sector to flash memory through the IPL storage manager and release the in-memory log sector. When a tree node is to be read from flash memory, the IPL storage manager returns an in-flash copy of the tree node along with its log records. Then, the Log applier creates the up-to-date tree node by applying its log records to the tree node. Finally, the up-to-date version of the tree node is added to the IPL buffer pool.

If a block merge operation is requested, the IPL storage manager allocates a new erase unit, and creates the up-to-date version of each tree nodes in the data area of the old erase unit by applying the corresponding log records of each node through the Log applier. And then, the up-to-date tree nodes are stored in the newly allocated erase unit. Since the entire tree nodes in the merged erase unit are relocated to a physically different region in

flash memory, the logical-to-physical map is updated by the IPL storage manager, when a merge operation is complete. Therefore, along with the basic operations (read/write/erase), the IPL storage manager maintains the logical-to-physical address mapping information.

3.3 Operations of IPL B+-tree Index

As shown in Fig. 2, the logical view of the IPL B+-tree index is the same as that of the traditional disk-based B+-tree index. Therefore the IPL B+-tree index has a same hierarchical node structure and the same logical operations of the disk-based B+-tree, such as node split and node merge. The only difference from the traditional disk-based B+-tree index is the IPL storage manager which encapsulates all the complex logics behind the IPL and thus provides simple disk-like interface to the upper module. Consequently, the IPL manager is responsible for two types of operations required for flash memory. The first one is a flash read operation, which is requested when a page operator asks for an index node from flash memory. The other type is a flash update operation, which is performed to write log sectors into flash memory. Flash update operations are commonly caused by insert or delete operations from host applications.

3.3.1 Flash read operation

Algorithm 1 shows a pseudo-code of the node retrieval procedure for B+-tree index. When node retrieval is requested for a page *P* by the page operator, the IPL manager finds a tree node (*N*) with page number *P*. If the node is found in the buffer pool, the node is returned. Otherwise, the IPL manager loads the node as well as its log data from flash memory. To make the node to be up-to-date, the IPL manager applies the log to the node using the applyLog() function. Finally, the IPL manager allocates the node into the buffer frame and clears the log sector of the buffer frame.

Algorithm 1: Node Retrieval

Input: pageNo *P*
Output: node *N*
1. **for** each buffer frame *B* in IPL Buffer **do**
2. **if** *b*.pageNo = *P* **then**
3. *N* ← *b*.nodeData
4. **return** *N*
5. **end if**
6. **end for**
7. *N* ← readFromFlash(getPhysicalNodeAddress(*P*))
8. allocate a New Log Page *L* (Log page size is 8KB)
9. *L* ← readFromFlash(getPhysicalLogAddress(*P*))
10. **for** each log sector *S* in *L* **do**
11. **if** *S*.pageNo = *P* **then**
12. applyLog(*N*, *S*)
13. **end if**
14. **end for**
15. replaceBufferFrame(*B*, *N*) // if the buffer frame is full, then execute replacement
16. **return** *N*

Algorithm 2: Node Update

```
Input: pageNo P, new Log N
1. for each buffer frame B in IPL Buffer do
2.   if B.pageNo = P then
3.     if isFull(B.logSector) then
4.       writeLog(B)
5.       initiate(b.logSector)
6.     else
7.       appendLog(b.logSector, N)
8.     end if
9.   end if
10. end for
```

Algorithm 3: WriteLog Function

```
Input: BufferFrame V
1. M ← getPhysicalLogAddress(V.pageNo)
2. if isFull(M) then
3.  M ← blockMerge(V.nodeData))
4. end if
5. writeToFlash(M, V.logSector, LogSectorSize) // LogSectorSize is 512Byte
```

Algorithm 4: blockMerge

```
Input: pageNo N, nodeData D
Output: physicalAddress P
1. M ← getPhysicalBlockAddress(N)
2. G ← getPhysicalNodeAddress(N)
3. allocate a New Log Page L (Log page size is 8KB)
4. L ← readFromFlash(getPhysicalLogAddress(N))
5. allocate a New Physical Block Address P
6. K ← 0
7. while K ⇐ DataAreaSize do // DataAreaSize is 120KB
8.   if G = M + K then
9.     writeToFlash(P + K, D, NodeSize) // NodeSize is 8KB
10.  else
11.    Q ← readFromFlash(M + K)
12.    applyLog(Q, L)
13.    writeToFlash(P + K, Q, NodeSize)
14.  end if
15.  K ← K + NodeSize
16. end while
17. update Logical-to-Physical Mapping Table M ← P
18. eraseFlashBlock(M) // optional operation, which depends on recovery policy
19. return P
```

3.3.2 Flash update operation

As shown in Algorithm 2, the node update operation can be done to append the log

records into the in-memory log sector of its tree node. However, when an in-memory log sector becomes full in buffer frame and needs to be flushed, the IPL manager executes the writeLog() function as shown in line 4 of Algorithm 2. Another case of executing a write-Log() function is when there is no more buffer frames available in the LRU buffer pool. The writeLog() function determines whether the log area of its erase unit can accommodate the in-memory log sector. If it does, the log sector in buffer frame is written to flash memory. Otherwise, the IPL manager calls the blockMerge() function as shown in line 3 of Algorithm 3, and then the in-memory log sector is written to flash memory.

Algorithm 4 shows the pseudo-code of the block merging process. The IPL manager allocates a new physical block and creates the up-to-date version of each node stored in the data area of an old physical block by applying the corresponding log of the node as shown in lines 7-16. The up-to-date nodes are stored in the newly allocated physical block. Finally the IPL manager updates the logical-to-physical mapping table, and then the old physical block can be erased.

4. PERFORMANCE EVALUATION

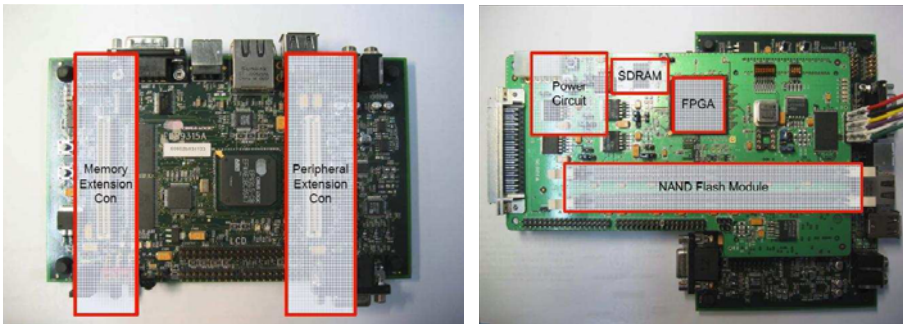
In this section, we perform three experiments for the performance evaluation of IPL B^+-tree index. In the first experiment, we prove the concept of IPL scheme by implementing IPL B^+-tree over an embedded board and show that the IPL B^+-tree is suitable for the flash memory. From the second experiment, we demonstrate the FTL dependency of the traditional B^+-tree index. And from the third experiment we point out the problems of previous flash-aware B^+-trees and compare their performance with that of IPL B^+-tree.

4.1 Implementation IPL B^+-tree on an Embedded Environment

One of the technical challenges in implementing the IPL B^+-tree is that the logical-to-physical mapping for flash memory blocks needs to be under the control of the IPL storage manager. However, all of the flash storage devices, like SSD (solid state disk), come with an on-device controller that runs a FTL to deal with address mapping as well as command interpretation [11]. Consequently, the logical-to-physical mapping is completely hidden from outside the flash memory devices such as SSD. This leads us to choose an EDB9315 processor board from Cirrus Logic [7] as a hardware test bed instead of a computing platform equipped with flash storage devices such as SSD drives.

The EDB9315 processor board (shown in Fig. 4 (a)) comes with a 200 MHz ARM-920T processor and 32 MBytes of SDRAM, and runs Linux 2.6.8.1 kernel. Since the EDB9315 processor board is not equipped with a storage device, it is augmented with an additional circuit board with Xilinx Spartan XC3S5000 FPGA controller and one GBytes of Samsung K9F2G08U0M-PCB0 NAND flash chips (shown in Fig. 4 (b)). The IPL B^+-tree runs on the processor board and accesses flash memory blocks directly without any intermediary such as a flash translation layer.

To evaluate the performance of IPL B^+-tree index, we measure the elapsed time of insertion and selection operations of IPL B^+-tree (or traditional B^+-tree) index on different storage devices. In following the performance evaluation process will be described. First we insert a set of 100,000 new entries randomly into an initially empty IPL B^+-tree



(a) Cirrus EDB 9315A development board. (b) NAND flash memory daughter board.

Fig. 4. Development platform for IPL B$^+$-tree.

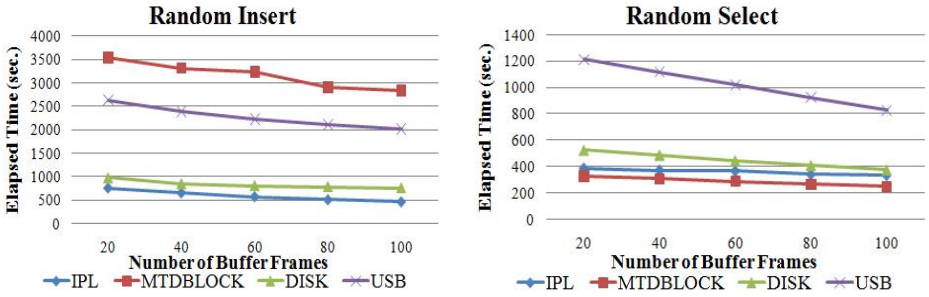
Table 2. Various devices for experiment.

Type of Index & Storage Device	Interface	Type of Device under Linux Kernel	Features of device
B$^+$-tree Index (Hard Disk)	IDE	Block device (ext2 file system)	250Gbyte, 7200 rpm
B$^+$-tree Index (1GB USB Flash Disk)	USB 2.0	Block device (ext2 file system)	Read: 22MB/sec Write: 12MB/sec unknown FTL
B$^+$-tree Index (MTD-BLOCK)	Memory connector & Peripheral connector	Mtd-block device (with naïve FTL)	Daughter board (Fig. 4 (b))
IPL B$^+$-tree Index	Memory connector & Peripheral connector	Mtd-character device	Daughter board (Fig. 4 (b))

(or traditional B$^+$-tree) index. After the insertion process has finished, we execute a set of 100,000 random selections against the IPL B$^+$-tree (or traditional B$^+$-tree) index. The node size of the B$^+$-tree index is 8KB, and an entry, whose size is 16byte, consists of a key and a value.

Table 2 shows each storage device in detail. As illustrated in Table 2, a hard disk, a USB flash disk and a mtd-block are all block devices so that we have to implement traditional B$^+$-tree index since we cannot skip FTL on each device. IPL B$^+$-tree index is only implemented on mtd-character device. Memory Technology Device (MTD) subsystem is a generic subsystem for handling memory technology devices under Linux. MTD subsystem can also handle the raw flash memory chip so that we can access flash memory directly without FTL. We can use the flash memory as block device (mtd-block) or character (mtd-character) device through MTD. Since mtd-block device has only block device IO interface with naïve FTL, we mount flash memory as mtd-character device to implement IPL B$^+$-tree, which access flash memory directly. Mtd device is usually used to raw flash memory chip, so that the daughter board, shown in Fig. 4 (b), behaves as if it was a raw flash memory chip unlike other commercial flash storage device.

Fig. 5 shows the performance of IPL B$^+$-tree and the traditional B$^+$-tree index. The y-axis represents the elapsed time and the x-axis represents the number of buffer frames of each B$^+$-tree index. As shown in Fig. 5 (a), the IPL B$^+$-tree index outperformed the traditional B$^+$-tree index for random insertions, especially in case of flash device. The



(a) Random insertion. (b) Random selection.

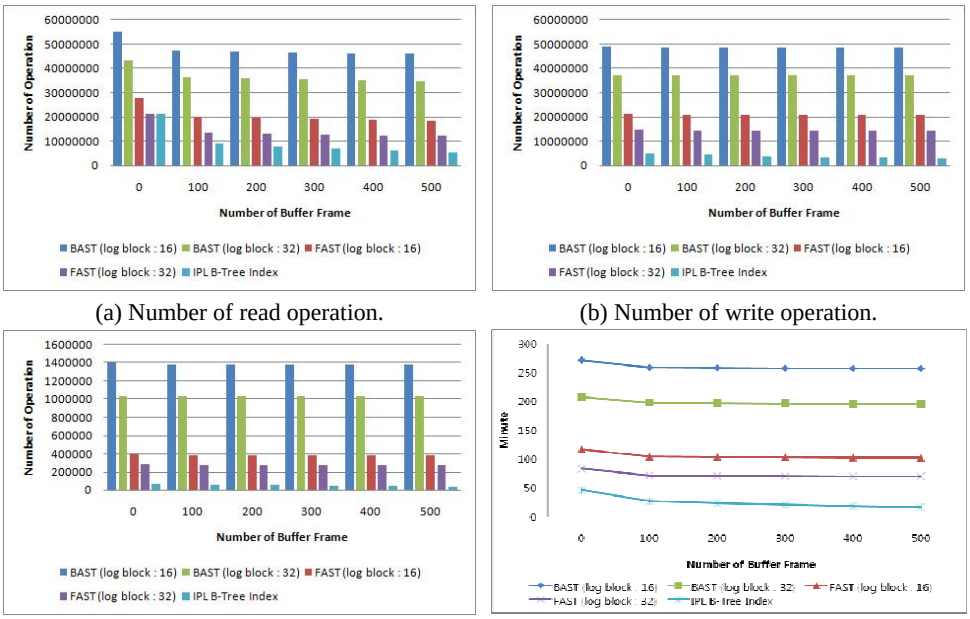
Fig. 5. Elapsed time (IPL vs. traditional B+-tree).

poor performance of the traditional B+-tree can be attributed to the amount of write operation of each device. The unit of the write operation of the traditional B+-tree index, whose device is block device like in a USB, HDD and mtd-block, is equal to the size of the node (8KB) and the unit of the write operation of the IPL B+-tree index is generally a sector (512Byte). The basic unit of write operation in flash memory is a page (2KB), so that traditional B+-tree index requires four write operations of flash memory. The IPL B+-tree, however, requires only one write operation. Consequently, in the flash storage device, the traditional B+-tree index incurred more frequent block merge operations. Moreover, random insertions caused internally flash random write so that FTL also required block merge operation more frequently.

Fig. 5 (b) shows the result of random selection of the IPL B+-tree and traditional B+-tree index. As shown in Fig. 5 (b), the traditional B+-tree with mtd-block device outperformed the others because of excellent read performance of the flash memory. One unexpected, but the interesting point is that the performance of the traditional B+-tree of HDD device outperformed other block devices with FTL. This peculiar result is caused that most HDD devices have their own write cache on device and the linux kernel also has a system buffer for block device, so the index in case of HDD can be positively affected the additional effect for the performance. Another point is the performance of USB disk. The most commercial flash storage device manufactures do not open their FTL algorithms so that we cannot analyze the poor result of USB device. However, we can find the performance of the tree on flash devices with poor FTL is inferior to HDD.

4.2 The FTL Dependency

In order to be clear about the IO performance of IPL B+-tree index and to reveal the weakness of the FTL dependency in the existing FTL-based index, we perform another experiment. In this experiment, we deliberately remove any side effect of write caching, buffer cache, translation bus, *etc*. on IO performance by comparing the number of IO operations. At this time, we insert a set of 1,000,000 new entries randomly into an initially empty IPL B+-tree index on mtd-character device and traditional B+-tree index on the FTL. In case of IPL B+-tree index, we can get the number of flash operations (read/write/erase). However since the FTL code is stored in the ROM and we cannot access the FTL of flash storage devices, there is no way to get the number of flash operations of FTL. Consequently, to get the number of flash operations of traditional B+-tree on the



(a) Number of read operation. (b) Number of write operation.

(c) Number of erase operation. (d) Simulate IO time.

Fig. 6. The number of flash operations of various B^+-tree.

FTL, we use the FTL simulator, which can simulate the BAST and FAST FTL. The FTL simulator is implemented by JDK 1.4 (Java Development Kit) and can configure the number of log blocks of each FTL. The BAST and FAST FTL perform more efficiently as the number of log blocks is getting larger but the space utilization is getting worse. In this experiment we set the number of log blocks is 16 (11%) and 32 (22%).

Fig. 6 shows the result of the experiment. In Figs. 6 (a)-(c), we can see the performance of each flash operation of the index. In all cases, the IPL B^+-tree index outperformed the traditional B^+-tree index regardless of the buffer frame. As explained in the previous experiment, the main reasons of this result are the size of the write unit and the randomness of the access range of the flash memory. Consequently, in case of the B^+-tree index, BAST FTL, which is suitable for sequential write, was the worst, and the FAST FTL was more efficient than the BAST FTL as shown in Fig. 6. On the other hand, the IPL Scheme, which is suitable for random write, had the best performance. Fig. 6 (d) shows the simulated elapsed IO time on the assumption that the latency of each operation is like Table 1. As it might be expected, the IPL B^+-tree index outperformed the others at least twice and the maximum performance was more than 15 times.

These results means that the performance of the B^+-tree index heavily depends on the characteristics of FTLs such as the logical-to-physical addressing algorithms and the number of log blocks in case of block mapping FTLs. Therefore we have to consider internal FTL algorithms before implementing an index on top of the FTL. However, it is almost impossible to know the internal FTL algorithms.

By the way, through the above two experiments, we can find out that the main problem of the traditional B^+-tree index under the FTL is their amount of write operations and their randomness, not the size of buffer frame. Consequently, we conclude that since most

FTL algorithms are not good for B+-tree index, we have to redesign the B+-tree index for flash memory, and that the IPL scheme is the best choice of the basic storage scheme for B+-tree index under the flash memory.

4.2 Performance Comparison with Previous Work

In this section, we analyze the problems of previous work and compare the performance of our scheme with that of the existing schemes. The existing flash-aware B+-tree schemes can be divided into two types: FTL-optimized and NAND-optimized. An FTL-optimized B+-tree scheme (*e.g.* BFTL [3], FlashDB [4], and FD-tree [5]) assumes the FTL module provided by the flash storage device while a NAND-optimized B+-tree scheme (*e.g.* μ-Tree [12], our IPL B+-tree) does not assume FTL and the scheme itself plays the role of FTL and thus can control the internals of flash storage.

In this section, we compare our IPL B+-tree index with BFTL and μ-Tree. We choose BFTL for comparison because it is the first and one of the most well-known FTL-optimized B+-tree. Besides, BFTL is so intuitive that we can easily explain the limitations of FTL-optimized B+-tree schemes using it. Meanwhile, as far as we know, μ-Tree is the first NAND-optimized B+-tree index. Before explaining the experimental result, let us briefly overview BFTL and μ-Tree.

In order to alleviate the poor random write performance in flash storage device, BFTL represents insert/update/delete operations against a node in B+-tree as a log at the record level, stores the log data sequentially in RAM area, called reservation buffer, and flushes the log data to flash memory when a node-sized buffer slot is full. It should be noted that because a page can be shared by the log data of two or more nodes in BFTL, the change logs of a node might be stored in several physical pages which are scattered in flash memory. That is, each logical node in BFTL does not have its dedicated physical storage unit, instead its records are scattered in many different physical pages in flash memory. For this reason, BFTL need to maintain a complex node translation table (NTT) which maps each logical node in B+-tree to the pages containing any log data for the node. Consequently, when a node is accessed, the node should be constructed on the fly by reading all the pages containing its log data from the flash memory. In contrast to BFTL, the log records in IPL B+-tree are co-located in the same physical block so that we just read the original node and the corresponding log records from a flash block. Thus, the additional IO overhead in accessing a node in IPL B+-tree does not exceed a log area size (8KB) at most, which is same as one node.

In order to reduce the read overhead in BFTL, the compaction operation is called for a node when the number of pages which contains any log data of the node exceeds the predefined threshold value. For the compaction operation, BFTL reads the pages containing the log data of the node, builds the node in RAM, and then re-write the node into a series of pages. Consequently, as more compaction operations are called in BFTL, less page read operations are performed but more write and erase operations are necessary. In addition, BFTL does not suggest any kind of garbage collection for the invalid pages at the logical file system level, after the compaction operation. They simply mention the internal garbage collection inside the underlying FTL, which is not controllable by upper tree module. However, to reclaim the invalid pages at the file system level, a garbage collection should be carried out at the file system level by tree module itself and

this burden would be not trivial.

μ-Tree is a novel variant of B^+-tree tailored to the characteristics of NAND flash memory. μ-Tree use the NAND flash interface instead of block device interface and thus μ-Tree, like IPL B^+-tree, does not use any FTLs. In the naïve implementation of B^+-tree without FTL on NAND flash memory, if an update occurs on a record in the leaf node, the updated node should be written at a new empty page due to the erase-before-write limitation of NAND flash memory. Since the physical address has been changed, the pointer of its parent node also should be changed. Consequently, since the all nodes from the leaf node to root node have to be changed, naïve implementation of B^+-tree cause as many write operations as the height of the tree. In μ-Tree index, since all nodes along the path from the root to the leaf node are located together in a single flash page, μ-Tree can minimize the number of write operations when a leaf node is updated.

However, since the μ-Tree uses only one single page in order to store all nodes along the path, its node size cannot be configured by user. In addition, as the record size or the number of records gets larger, the height of the tree should drastically increase. Moreover, since the root node is always only one in the B^+-tree, except for one flash page with current root node, every page have useless space of the old path. Therefore the space utilization of μ-Tree could be low, and thus the required storage space of μ-Tree would be much larger than naïve implemented B^+-tree or IPL B^+-tree. In addition, one serious concern in μ-Tree is that it does not have the chaining pointers of leaf nodes. As a result, the range scan query could not be effectively supported, which is a common usage of B^+-tree.

Table 3. Insertion performance.

Index (node)	Number of operations			Elapsed time	Used blocks	Total blocks
	Read	Write	Erase		Data + Log	
BFTL (2KB)	8762892	1650295	25341	17.82 min	1472 + 442	1914
BFTL (2KB)	10128909	3037995	47342	24.82 min	1472 + 148	1620
BFTL (8KB)	46437542	33541087	525485	187 min	606 + 182	788
μ-Tree	7669149	1262588	19547	14.92 min	182	256
IPL B^+-tree	7444079	3544772	47667	22.93 min	90	100

Now let us proceed with the experiment result. Table 3 shows the number of IO operation and the elapsed time of three flash-aware B^+-trees, BFTL, μ-Tree and IPL B^+-tree after inserting 1,000,000 records and a record consists of a key and a pointer. The data type of keys and pointers is integer and the key ranges between 1 and 1,000,000 and the keys were inserted in a full random way.

In case of the BFTL, we configure the node size to be 2KB and 8KB and simulate it on the FAST FTL [8]. As mentioned before, the performance of BFTL varies much depending on FTLs. Since our workload is fully random and the FAST FTL is known to be good in random workload, we choose the FAST FTL for fair comparison. The maximum number of pages in NTT for the compaction in BFTL is set as 4 when the node size is 2KB (1 page) and 32 when the node size is 8KB (4 pages) according to [3]. In Table 3, the Log in 'Used Blocks' column means the number of log blocks of FAST. FAST can

improve the performance of random writes using the log blocks, and thus the FAST FTL can perform better as the number of log blocks increases. We set the log blocks 30% and 10% of the data blocks. Meanwhile, the μ-Tree and IPL B$^+$-tree do not need log blocks.

As shown in Table 3, μ-Tree, as we expected, outperformed other B$^+$-trees. However, as mentioned before, because of inefficient space utilization, μ-Tree required twice used blocks than IPL B$^+$-tree. And, as shown in 'Total Blocks' column in Table 3, it had an additional cost for garbage collection when more records were inserted. Consequently, μ-Tree showed the best performance, but it has two limitations: no range scan support and storage space inefficiency.

On the other hand, BFTL outperformed IPL B$^+$-tree by far in the write and erase operations when the node size was 2KB. However, when the log blocks were 10% of data blocks, it underperformed IPL B$^+$-tree. This implies that the performance of BFTL is determined by the characteristics of FTL in use. When the node size was 2KB and the log blocks were 30%, BFTL outperformed IPL B$^+$-tree. However, the used blocks of BFTL were more than 2100% of IPL B$^+$-tree. Since every entry in BFTL includes the meta-data such as node ID, parent node, and chaining node, the additional space is always required. Moreover, the compaction is the key tuning knob for read performance and thus there should be many inevitable invalid pages after compactions. Therefore, BFTL requires excessive flash memory space for reasonable performance. To reduce the excessive space requirement, we set the node size to be 8KB. However, as shown in Table 3, the performance of BFTL has drastically degraded and the total blocks are still quite large.

On the other hand, IPL B$^+$-tree requires 100 blocks to store 1,000,000 records and it does not suffer from any kind of garbage collection overhead because IPL B$^+$-tree performs the block merge operation (*i.e.* a kind of foreground garbage collection) when necessary. In addition, IPL B$^+$-tree can improve the performance by delaying the block merge operation. Consequently, IPL B$^+$-tree requires the smallest space among the schemes to be compared and does not suffer from additional costly operation such as garbage collection.

Table 4. Read overhead.

	Total Read Cnt.	Node Read Cnt.	Additional Page Read Cnt.
BFTL (2KB)	8762892	2019972	6742920 (including compaction)
μ-Tree	7669149	7669149	Not Available
IPL B$^+$-tree	7444079	2902100	4541979

In order to show the additional read overhead of BFTL and IPL B$^+$-tree, we broke down the number of read operations in 'Read' column in Table 3 into the number of read operations for pages for normal data nodes and the number of read operations for log pages. Table 4 shows the additional read overhead in each of three index schemes while inserting 1,000,000 records. In fact, μ-Tree does not require any additional read operation for log data. However, as shown in Table 4, total read count of μ-Tree was comparatively similar to others. This imply that although μ-Tree does not need to read additional log or pages, since it has to maintain the physical addressing pointer and its path from the root node to leaf node, additional node read operations are required [12]. On the

other hand, BFTL and IPL B^+-tree must read additional pages to read a node. As shown in Table 4, the node read count of BFTL was smaller than that of IPL B^+-tree. However, the number of additional page read operations of BFTL was quite larger than that of IPL B^+-tree. Consequently, in spite of the compaction operation of BFTL, its read overhead is still considerable.

5. CONCLUSION AND FUTURE WORK

In this paper, we proposed a novel index structure called IPL B^+-tree that can reduce write and erase operations efficiently on NAND flash memory by using In-page Logging scheme. Our evaluation results showed that the IPL B^+-tree index outperforms the traditional B^+-tree index with FTL at least twice and the maximum performance is more than 15 times. We also explained the FTL dependency when we use the flash memory as a storage medium for B^+-tree index. Besides we described the potential problems of the previous flash-aware B^+-tree indexes. Our future work is that we will design the recovery policy of the IPL B^+-tree index without additional log like undo and redo log. We also plan to investigate the concurrency control policy of the IPL B^+-tree index.

REFERENCES

1. Intel Corporation, "Understanding the flash translation layer (FTL) specification," Application Note AP-684, Intel Corporation, 1998.
2. S. W. Lee and B. Moon, "Design of flash-based DBMS: An in-page logging approach," in *Proceedings of the ACM SIGMOD*, 2007, pp. 55-66.
3. C. H. Wu, T. W. Kuo, and L. P. Chang "An efficient B-tree layer implementation for flash-memory storage systems," *ACM Transactions on Embedded Computing Systems*, Vol. 6, 2007, Article 19.
4. S. Nath and A. Kansal, "FlashDB: Dynamic self-tuning database for NAND flash," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, 2007, pp. 410-419.
5. Y. Li, B. He, Q. Luo, and K. Yi, "Tree indexing on flash disks," in *Proceedings of IEEE International Conference on Data Engineering*, 2009, pp. 1303-1306.
6. A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proceedings of USENIX Winter Technical Conference*, 1995, pp. 155-164.
7. J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Transactions on Consumer Electronics*, Vol. 48, 2002, pp. 366-375.
8. S. W. Lee, D. J. Park, T. S. Chung, D. H. Lee, S. Park, and H. J. Song, "A log buffer-based flash translation layer using fully associative sector translation," *ACM Transactions on Embedded Computing Systems*, Vol. 6, 2007, pp. 436-453.
9. Samsung Electronics, "NAND flash memory data sheets," http://www.samsung.com/.
10. E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Computing Surveys*, Vol. 37, 2005, pp. 138-163.
11. M. Wu and W. Zwaenepoel, "eNVy: a non-volatile, main memory storage system," in *Proceedings of the 6th International Conference on Architectural Support for*

Programming Languages and Operating Systems, 1994, pp. 86-97.
12. D. Kang, D. Jung, J. U. Kang, and J. S. Kim, "μ-tree: An ordered index structure for NAND flash memory," in *Proceedings of ACM Conference on Embedded Systems Software*, 2007, pp. 144-153.

Gap-Joo Na received the B.S. degree in Avionics Engineering from Korea Aerospace University and the M.S. degree in Computer Engineering from Sungkyunkwan University. He is presently a Ph.D. candidate at the School of Information and Communication Engineering, Sungkyunkwan University. His current research focuses on flash-based database technology.

Bongki Moon is Professor of Computer Science at the University of Arizona. His current areas of research are flash memory database systems, XML indexing and query processing, and information streaming and dissemination. He has served on program committees and review panels for numerous conferences, workshops and the National Science Foundation. He currently serves on the editorial board of the IEEE Transactions on Knowledge and Data Engineering (TKDE) as an associate editor. He received an NSF CAREER Award in 1999 for his work on distributed cooperative web server design. He received his Ph.D. degree in Computer Science from University of Maryland, College Park in 1996, and his M.S. and B.S. degrees in Computer Engineering from Seoul National University, Korea, in 1985 and 1983, respectively. He was on the research staff for Samsung Electronics and Samsung Advanced Institute of Technology, Korea, from 1985 to 1990.

Sang-Won Lee is an Associate Professor with the School of Information and Communication Engineering at Sungkyunkwan University, Suwon, Korea. Before that, he was a Research Professor at Ewha Womans University and a technical staff at Oracle, Korea. He received a Ph.D. degree from the Computer Science Department of Seoul National University in 1999. His research interest is in flash-based database technology.