An Efficient Buffer Scheme for Flash-based Databases

Liang Huai Yang[†*], Jing Wang[†], Zhifeng Huang[†], Weihua Gong[†], Lijun Chen[‡]
[†]College of Computer Science and Technology, Zhejiang University of Technology, Hangzhou, China
[‡]School of Information Science and Technology, Peking University, Beijing, China
[*]Email: yang.lianghuai@gmail.com

Abstract— **Most embedded database systems are built on a two-level memory hierarchy, a RAM buffer on top of flash memory. Both kinds of memories have limited capacity, thus, how to efficiently utilize them is critical for embedded systems with resource restrictions. Different from magnetic hard disk, flash memory has speed asymmetry in reads and writes, i.e., random reads are over an order of magnitude faster than random writes. μ-tree [9] is the state-of-the-art index supporting efficient random updates on the flash memory. However, we found that the traditional page based buffer scheme(LRU-P for short) under-utilized the RAM space. To improve the utilization on the memory resources and the query performance, we propose a novel buffer scheme called LRU-N – a node-based LRU policy for the RAM buffer. In addition, to increase the utilization of the flash memory, we also present a *k-partitioning* strategy for μ-tree. LRU-N uses a *page address swizzling* scheme to exploit the co-related locality of accessing multiple tree nodes in the μ-tree. The *k-partitioning* strategy determines an optimal k value for space efficiency by allocating more flash memory to the leaf nodes and guarantees sufficient space for tree growth. Our experiments conducted on both simulated data sets of various distributions and real-world data sets show that LRU-N can significantly improve buffer hit rate up to 65% for the best cases over the page-based scheme LRU-P; even with small buffer size, LRU-N can also achieve rather good performance. And our proposed k-partitioning strategy can effectively reduce the index size up to 33% over μ-tree.**

Index Terms— **buffer scheme, LRU, μ-Tree, flash DB**

I. Introduction

Flash memory holds many attractive features such as low power consumption, small size, light weight, and shock resistance. Because of these features, flash memory is widely used in consumer electronics and embedded systems such as mobile phones and sensors. Recently, flash memory has been adopted by personal computers and servers, which might be mounted on the computer motherboard, a DIMM slot, a PCI board, or within a standard disk enclosure as solid-state disk (SSD).

There are two types of flash memory, namely NOR and NAND. NOR flash provides fast read access, slow write/erase time, low density and a random-access interface. The latter property makes NOR flash ideal for application or boot code execution in place (XIP) as it allows direct memory addressing. On the other hand, NAND-flash has faster write/erase times and requires a smaller chip area per cell, thus increasing the overall storage capacity. This also lowers the cost of NAND flash. However, NAND flash does not allow random access to any memory location like NOR flash as it operates with a page-based interface. This character makes NAND flash suitable for external storage.

As a storage medium, NAND flash distinguishes itself from traditional hard disk medium with some special features. The minimum unit of read/write operation in flash is a page. Page sizes vary between 256B to 4KB. Each block consists of a number of pages (with varieties of 32×512B, 64×2KB, 64×4KB and 128×4KB). While reading and programming is performed on a page basis, erasure can only be performed on a block basis. Flash updates need to be preceded by an erase operation. Two consecutive writes to the same physical location must be interleaved by block erasure. Updating a page in its original location (in-place-update) is prohibitively expensive, it requires copying all valid pages from the erase block, then erasing the block, and finally copying the valid pages and updated page back to the block. And data in a block can only be written sequentially, i.e., after page p_i has been written, any page p_j, $1 \leq j < i$ can not be written even if it is empty. The number of times a block can be written or erased is limited (typically between 10,000 to 100,000 times). Whereas typically read and write access on a hard disk are almost identical, in flash it is much faster to read ($\approx 60\mu s$) than to write ($\approx 800\mu s$).

With the increasing popularity of NAND flash memory as data storage for embedded systems and the rapid growth of its capacity, many file systems and database management systems are being built on it. There exist two major approaches to use NAND flash memory as a storage device in embedded systems. One is to employ Flash Translation Layer (FTL) between applications and NAND flash memory. The advantage is that legacy file systems or database systems designed for disks can be used without any modification. However, Most FTL algorithms are designed based on the principle of locality in storage access patterns. FTL may experience degraded performance if the access patterns of the workload do not match those optimized in FTL. For example, the updates to the B+-Tree-like index structure results in random accesses requiring many small flash write operations. In

This work was supported by NSFC Project(61070042), Zhejiang Provincial NSF Project(Y1090096) and China “863” Hi-tech Program(2007AA01Z153). Corresponding author: Liang Huai Yang, Email: yang.lianghuai@gmail.com.

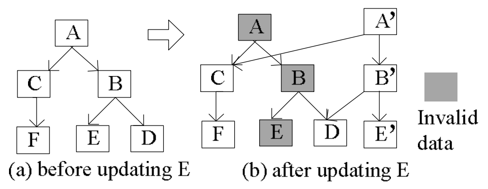


Figure 1. A B+-Tree update example for node E

addition, garbage collection, which reclaims invalid pages by erasing appropriate blocks, may not be efficient due to the limited information available at block level. For instance, SQL Sever Everywhere employs a B+-Tree over FTL and performs poorly. The reason is due to that the out-place-update (write after erase) feature of the flash memory, random updates are not only time-consuming but also consume flash space, which leads to more garbage collection operations.

The second approach is the native approach, directly managing NAND flash memory without an intermediate layer. In this approach, all is under user's control. Designers can take into account of the peculiarities of flash memory, such as speed asymmetry in reads and writes, and out-place-update. This is especially useful in the case of resource-restricted embedded system. As such, lots of efforts [9], [13], [20] are put into the design of an efficient index structure to locate a particular item quickly from database in embedded devices. In this study, we focus on the native approach for flexibility.

Kang et al [9] proposed an index structure μ-tree that operates native flash memories without FTL. μ-tree is the state-of-the-art index supporting efficient random updates on the flash memory. It is similar to B+-Tree and is a balanced tree, which is well fit for query over large volume of data. If B+-Tree is applied to a native flash-based database, an update to a B+-Tree leaf node results in the updates along the path from leaf to root. Taking Fig.1 as an example, Fig.1 (a) shows a B+-tree before updating node E. When updating node E, one feasible approach is to write the update to a new page and discarding the original node page E without incurring the expensive block erasure cost immediately. With E changing to a page, its physical address changes as well. Since E's parent node B is pointing to it, B has to update its pointer, and B has to be moved to a new page as a result due to the out-place-update rule. Such cascading updates continues to propagate upward along the path till to the root. Fig.1 (b) shows the final B+-tree after updating E, where all the invalid nodes are shaded in gray, and its net effect looks like a "*Wandering Trees*" [3]. Unlike B+-Tree, where each node corresponds to a page, μ-tree organizes those nodes to be updated along the path from root to leaf into one page. Thus, μ-tree requires only a single flash write operation whenever any node in the tree is updated.

However, μ-tree has two shortcomings: (1) μ-tree uses read- and write-buffers separately, where the write-buffer is used for buffering those updated pages, which is flushed out to flash memory when full, and read-buffer adopts LRU as its buffer policy. Since multiple nodes along the path in the μ-tree reside in the same page, an update operation on one node can induce invalid node (s) in other μ-tree page (s). This greatly degrades the buffer space utilization. (2) The node size of μ-tree is dissimilar to the traditional index. A node in the μ-tree is only a fraction of the entire page, and decreases as the height of μ-tree increases. The space of μ-tree can be approximately estimated as the number of pages to hold all leaf nodes. Let the fanout be f and the total number of records be n, the space for B+-Tree is around $V_b \approx \lceil \frac{n}{f} \rceil$,and the space for μ-tree is around $V_\mu \approx \lceil \frac{n}{f/2} \rceil$.Given the same n and f, the space needed for μ-tree is about twice larger than the corresponding B+-Tree; It would be even worse, if the number of records are large or multiple indexes are needed in the embedded system. This piece of work aims to overcome these two shortcomings.

Our contributions are as follows:

- To tackle the first shortcoming, we propose a node-based LRU policy named LRU-N as the buffer management scheme, which aims at increasing the buffer space utilization. LRU-N uses a *page-address swizzling* scheme to exploit the co-related locality of accessing multiple tree nodes in the μ-tree. Our experimental evaluation shows that our proposed method can improve the performance up to 65% in the best cases. Even with small buffer size, our scheme achieves rather good performance.
- For the second issue, we propose a *k-partitioning* strategy for increasing utilization of the flash memory. The strategy determines an optimal k value for space efficiency by allocating more flash memory to the leaf nodes and guarantees sufficient space for tree growth. This enables the μ-tree to be applied to embedded system with a smaller storage requirement. The experiments show that the space reduction is achieved up to 33%.

For differentiation, we call it a $k\mu$-tree when a μ-tree adopts k value other than 0.5 determined by *k-partitioning* strategy.

The rest of the paper is organized as follows. Section II discusses the related work. Section III presents our node-based LRU placement/replacement policy named LRU-N for the RAM buffer, and details a *page-address swizzling* scheme to keep some useful nodes in buffer. Section IV describes a *k-partitioning* method which aims for high space utilization of $k\mu$-tree. Performance evaluations are given in section V; and finally, we conclude in Section VI.

II. Related Work

As the research on flash-based DBMS is gaining momentum [7], [11], [14], [17], several indexes for flash memory are proposed [1], [2], [9], [13], [18]–[20].

Due to out-of-place update feature in flash memory, an update to a B+-Tree leaf node results in the updates along the path from leaf to root, which is known as "Wandering

Trees". JFFS3 [3] adopts a log-structured design to amortize the update costs in the B+Tree. Insertions are logged in a journal and are applied to the B+Tree in a batch mode. A journal index is maintained in RAM recoverable at the boot time so that a key lookup is first performed on the journal index and then in the B+Tree. Wu et al [19] introduced a B+-Tree layer over FTL, namely BFTL, for flash memory to reduce the update cost. BFTL does not suffer from the out-of-place update problem. However, when a new record is inserted, its key is added to a B+-Tree leaf node, incurring an out-of-place update of the corresponding flash page. To avoid expensive update cost for each node, the changes are sequentially written to logs. The node mapping in the log is maintained by an in-memory data structure called Node Translation Table (NTT). While BFTL reduces the cost of random writes to the flash memory, the read cost is degraded by over a factor of two because many log pages related to the requested node must be accessed. Adopting the similar idea, Wu et al [18] presented an R-tree layer over FTL. Nath et al. [14] proposed a hybrid approach by mixing B+Tree and BFTL, and used a self-tuning scheme via grouping nodes into read-intensive and write-intensive nodes based on the workload characteristics.

Most recently, Chang et al. [2] propose a native index structure for NOR-Flash memory. However, as the authors point out, this index structure is suitable for NOR flash but not for NAND since NOR flash is byte-addressable while NAND flash is page-oriented. Agrawal et al. [1] present a lazy-adaptive tree (LATree) - a novel index structure that exploits flash-resident buffers for batching the update (insert and delete) operations. Meng et al. [21], [22] invent a flash index structure called PBFilter which tackles applications where insertions are more frequent and critical than deletions or updates. The above methods is orthogonal to our approach.

Both log-based and FTL-based indexes do not fit well for resource-restricted embedded systems because the memory footprint of journal is proportional to the number of nodes. The updates to B+-Tree reveal many small writes which are distributed randomly across many blocks, but most FTL algorithms perform poorly for such a workload [8], [10]. The use of log pages does lessen the problem since the log entries in the same node are eventually merged together.

Lin et al. [20] proposed MicroHash indexing structure to speed up lookups on sensor devices. Mani et al. [13] presented an index called TINX for sensor devices based on an unbalanced binary tree structure. Both methods are not scalable to a large number of records. It is also hardly applicable to file systems or DBMSs because it assumes that all stale records written a certain amount of time ago are abandoned in the deletion phase.

Our work is based on μ-tree [9]–the state-of-the-art index supporting efficient random updates on the flash memory. In particular, we address the space efficiency of the μ-tree, while preserving the good search performance. In order to improve the read/write efficiency of flash memory, buffer management scheme (BMS) is extensively studied. Kim and Ahn [6] proposed BPLRU (Block Padding LRU) to improve the random write performance of flash storage over FTL. Park et al. [15], [16] described CFLRU (Clean first LRU) for operating systems over flash storage. The idea is to trade reads for expensive writes by choosing a clean page as a victim rather than a dirty one. Jo et al. [5] presented another BMS–FAB (flash aware buffer policy) for flash memory by selecting a victim based on its page utilization rather than based on the traditional LRU policy. Jiang et al. [4] proposed DULO to exploit both temporal and spatial locality over FTL. Lee et al [12] reduced the power consumption via power-aware buffer cache (PABC). In contrast, we focus on the buffer efficiency of the index, and aim at improving the index performance via identifying the co-related buffering locality in the tree search.

III. Buffer Management Scheme

In this section, we present our node-based LRU replacement policy, namely LRU-N. A good buffer management scheme should capture the locality of data accesses in order to reduce the number of accesses to secondary storage in query processing. Compared with the traditional page-based LRU replacement policy, LRU-N exploits the correlated access locality during the search of the μ-tree.

A. Motivations

μ-tree [8] uses read and write-buffer separately, where write-buffer is used for buffering those updated page and is flushed out to flash memory when the buffer is full. However, there exist two features which lead the LRU strategy does not fit μ-tree well. First, μ-tree stores nodes in the same path into one page, thus, an update operation on one node may results invalid node (s) in other μ-tree page (s) see Fig. 2 for reference. Since the LRU replacement policy manipulates on pages, the invalid nodes may waste a lot of space and degrade the buffer space utilization. For example, in case of updating a μ-tree node E in Fig. 2, it causes node A invalid and wastes the buffer space as a consequence according to page-based LRU policy. In addition, the memory space above node D in page 3 is also wasted.

Second, a μ-tree page stores multiple nodes from different levels. The nodes at different levels have different access probabilities. Due to the tree structure, the nodes at lower level have a lower probability to be accessed. In particular, the probability of a node being accessed at level L is the sum of its child nodes' access probability. Consequently, a page-based LRU policy is not aware of such correlated locality in the tree search.

To improve buffer space utilization and exploit the co-related locality, we propose a node based LRU policy named LRU-N for a better performance on the μ-tree. For differentiation, we denote the traditional page-based LRU as LRU-P.

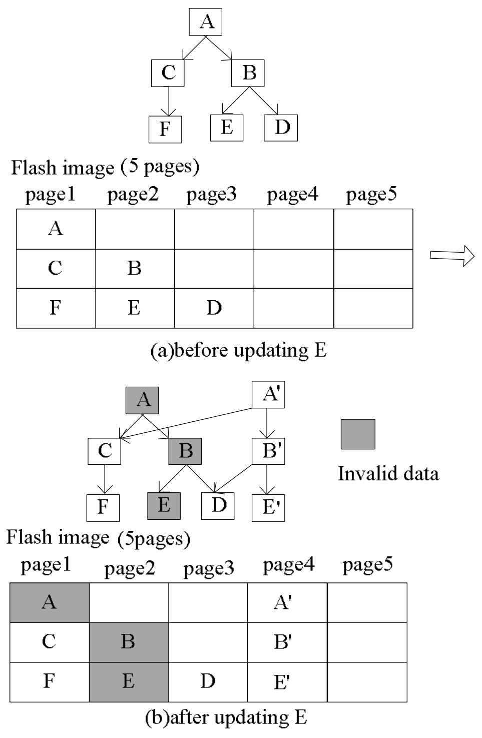


Figure 2. A μ-tree update example for node E

B. LRU-N

In view of the peculiarities of μ-tree, a new buffer management scheme should take into account the fact that higher level nodes have higher probability to be accessed than lower level ones while the nodes at higher level are much fewer than those at lower level, and try to keep as more higher level nodes in-memory as possible to fully utilize those wasted memory space. With these aspects in mind, we propose a novel replacement scheme called LRU-N, where, instead of applying LRU to the whole flash index page, LRU is applied level-wise to nodes of each level in μ-tree index page. To be specific, for index pages in the buffer pool, nodes of each level are logically organized into a list, where least recently used nodes are placed at the list head while recent frequently used nodes are put at the tail. Newly fetched valid node is also appended to the tail. When buffer pool is full, the head node is selected as the victim to be replaced. However, since the number of nodes in the higher level is fewer than that of the lower level in buffer pages, there exists some unused space which can retain in memory the valid node of the victim buffer page at the same level. As such, LRU-N takes advantage of this fact. When a valid node is moved to and retained in another memory page at some level, it will be annotated with its flash page address (hereafter FPA for short).

Since root is always kept in write-buffer [9], there is no need for read-buffer to manage it. A buffer pool and its corresponding LRU lists are illustrated in Fig. 3, where the leaf node number shown in the buffer page (here 3, 4, 5, 6) stands for FPA, and other node number of upper

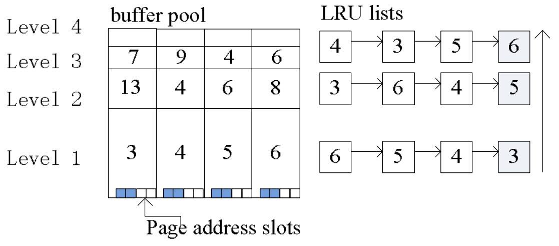


Figure 3. Buffer pool and its corresponding LRU lists

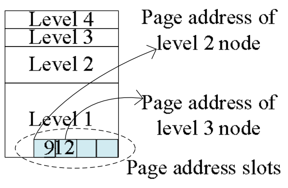


Figure 4. Buffer page layout for "*page address swizzling*" scheme

hierarchy means the node's FPA. For example, the FPAs in level 3 are 7, 9, 4, 6, where flash pages of 7 and 9 only have nodes of level 3 in buffer while their child nodes of the lower levels were evicted from buffer, and similarly, flash pages of 4 and 6 are kept in buffer but their nodes of different levels are kept in different buffer pages after certain LRU-N placement/replacement operations. Note that, for issue of presentation and implementation, the node number of each node shown in LRU lists uses the leaf node number of the buffer page where this node resides. The actual FPA for an upper level node is set by "*page address swizzling*" scheme described below.

Since nodes from different levels of μ-tree are organized into independent LRU lists, when buffer pool is full, the victim nodes (VNs for short) from different LRU lists may be from different flash pages. However, if DBMS requests a non-resident flash page, it needs a whole buffer page to adopt this flash page. To solve this problem, we contrive a "*page address swizzling*" scheme wherein some FPA slots are reserved at the end of each buffer page. An example buffer page layout is shown in Fig. 4, where the FPA of level 2 node is kept in the left most slot and FPA of level 3 node in the next slot. Note there is no need to keep the FPAs in the slots for leaves and root level node because, as stated before, root node is not managed in the read-buffer, while leaf node's FPA is always the original FPA of the corresponding flash page. When a node access request arrives, LRU-N works as follows:

1) If the node is found in buffer pool, the node is returned.
2) Otherwise, see if the buffer pool is full.
    a) If the buffer pool is not full, we fetch the node into the buffer and the node is returned.
    b) Otherwise, this event triggers LRU replacement and chooses a victim for each LRU list. Let VPA be the victim page address of the victim leaf node and VBP the victim buffer page, in order to keep as many upper level tree nodes in-memory as possible and not

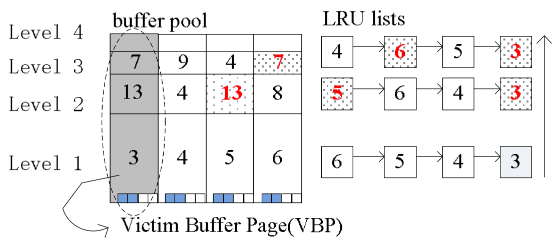


Figure 5. An execution example of LRU-N to make room for a new flash page

```
Input: L-level of the node to be found in the tree
          D-page address
Output: N-the node
1:  for each buffer page i= 1, ..., $M$
/** check the corresponding slot for the page address
*   of the level L node of buffer page i;
**/
2:      if found $slot_L$=$D$ in the page $D_i$ then
3:          return N←GetNodeFromPage($D_i$, L);
4:      end if
5:  end for
6:  if page $D$ not found in the buffer pool then
7:      for each buffer page i= 1, ..., $M$
8:          if $D_i$ = $D$ then
9:              return N←GetNodeFromPage($D$, $L$);
10:         end if
11:     end for
12: end if
13: fetch page $D$ into buffer from flash memory;
14: return N←GetNodeFromPage($D$, $L$);
```

Figure 6. Retrieving a node from a buffer page

waste unused space, the scheme performs the following actions:

i) Check if there are any in-memory nodes of the flash page to be adopted in the LRU lists. If there is any, move this node to the head of lists as the victim since these nodes will be brought in again and should make room for other nodes;

ii) Check the page address FPA of each victim node VN_i of each LRU list from bottom up (except leaf and root) for level i=2, $\cdots$, h-1, if this victim's page address is the same as VPA, repeat this step until it checks all LRU lists; otherwise, since the content of VN_i will be invalidated, and VN_i is not part of the VBP which is about to be replaced, so a natural idea is to retain the valid node in VBP by migrating it to VN_i. Hence, copy the level i node content (if valid) of VBP to node VN_i, set its corresponding buffer page's slot to VPA, and exchange the positions of VN_i and VBP's level i node in the LRU list where VN_i lies. Repeat this step until it checks all LRU lists;

iii) Finally, the normal LRU placing/replacing is performed: removing all victims from the LRU lists, reading the flash page into the buffer page, and adjusting each LRU list of each level by placing new valid nodes at the list's tail or placing the empty (invalid) node at the head.

Incidently, though we illustrate the buffer pool pages by including the top level nodes, in actual implementation, there is no need to allocate buffer space for the top level nodes.

Example: Assume the DBMS need to adopt a new page from the NAND flash block and the read-buffer is full as shown in Fig. 3. According to the LRU-N strategy, the buffer page indicated by the leaf node in LRU lists is selected to be the victim buffer page VBP and its VPA is 3. Next, LRU-N will check the upper level LRU lists' victim nodes. Since VN_2's buffer page address 5 is different from VPA, thus the content of level 2 node of VBP 3 is migrated to VN_2, i.e., the level 2 node 6 of the third buffer page, and its corresponding physical address slot is set appropriately (here 13); similarly, LRU-N will check VN_3=6, it differs from VPA again, so the content of level 3 node of VBP is transferred to VN_3 and the slot is set as well (here 7). By this time, it's ready for adopting the new flash page. The resulting example is shown in Fig. 5.

C. Node Retrieval in LRU-N buffer page

Since the logical structure of $k\mu$-tree and μ-tree are identical, the retrieval process of $k\mu$-tree is essentially the same as that of μ-tree. However, when retrieving a node in a buffer page from $k\mu$-tree, it needs some extra checking because of our "*page address swizzling*" scheme used in buffer management. Similar to μ-tree, GetNodeFromPage() is used iteratively during the retrieval process to locate the correct node. We will present its detailed algorithm on the *"k-partitioning"* method. Fig. 6 outlines pseudo code for retrieving the level L node from page D.

IV. *k-Partitioning* MODEL

The particular way of organizing μ-tree results in that its total used flash memory space depends on its allocated leaf node size. To obtain a small size $k\mu$-tree, we should allocate to the leaf node a portion of a page as large as possible. For the sake of addressing convenience, a fixed

M:	the number of buffer pages
i:	level i (leaf nodes are in level 1, root is in level h)
D_i:	the i^{th} page number
f_i:	maximum fanout in level i
p:	flash page size
$J(i)$:	nodes total in level i
F:	maximum fanout of nodes
S:	node size
n:	records total
k:	partitioning rate (0<k<1)

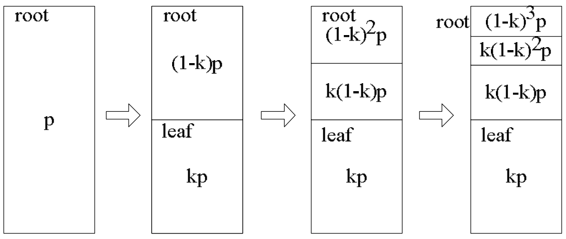


Figure 7. Page layout evolution of a $k\mu$-tree

allocation proportion k is used and can be applied to the top level nodes for space partitioning as necessary.

For example, let the flash page size be p. Initially the entire page is used for the root; as more records are inserted, the root node splits and the height of $k\mu$-tree increases by one level as a result. By then, the leaf nodes has the size of kp while the root node gets $(1-k)p$ in size. This process proceeds as necessary and applies to the root node in the end. The page layout evolution of a $k\mu$-tree is illustrated in Fig. 7, where node of each level is annotated with its node size.

The symbols used in our model are listed below:

The number of pages accessed during record retrieval depends on the height of $k\mu$-tree. Since the fanouts of nodes in the same level may not be same, it is difficult to work out the exact height of $k\mu$-tree. Thus, we estimate the maximum height of the tree as its height.

Firstly, let's figure out the max node fanout for each level of $k\mu$-tree. According to proposed *k-partitioning* model, the node size of the lower level is always allocated by k ratio of the space to be partitioned. As a result, we arrive at the formula for maximum node fanout at each level:

$$f_i = \begin{cases} k(1-k)^{i-1}F, & \text{i<h;} \\ (1-k)^{h-1}F, & \text{i=h.} \end{cases}$$

Next, we will work out the max height h of the $k\mu$-tree with n index entries. Assume that there exists another T' with the minimum number of index entries n' among all those $k\mu$-tree with height h. It is not difficult to know that T' has single index entry; otherwise, we can find another tree with less index entries and this is a contradiction. Similarly, for other nodes of T', they have the minimum number of index entries. Similar to B+-tree, $k\mu$-tree is a balanced tree and the number of index entries for each node is no less than half of the max fanout of this node. Thus, the number of nodes in the i-th level of T' can be computed as:

$$J(i) = \begin{cases} 1, & \text{i=h} \\ 2, & \text{i=h-1} \\ \prod_{j=i}^{h-2} \left\lceil \frac{k(1-k)^j F}{2} \right\rceil, & \text{i<h-1} \end{cases},$$

hence,

$$n' = \prod_{i=1}^{h-2} \left\lceil \frac{k(1-k)^i F}{2} \right\rceil \left\lceil \frac{kF}{2} - 1 \right\rceil$$

By some simplification, we have:

$$n' = \left(\frac{kF}{2}\right)^{h-2} (1-k)^{\left(\frac{(h-1)(h-2)}{2}\right)} \left(\frac{kF}{2} - 1\right)$$

Take the logarithms of both sides of the equation above:

$$\lg(1-k)\, h^2 + \left(2\lg\frac{kF}{2} - 3\lg(1-k)\right) h - \\ 4\lg\frac{kF}{2} + 2\lg(1-k) - 2\lg\frac{2n'}{kF-2} = 0$$

By solving the quadratic equation, we have:

$$h = \frac{3\lg(1-k) - 2\lg\frac{kF}{2} + \sqrt{\begin{matrix}\left(lg(1-k)+2\lg\frac{kF}{2}\right)^2 + \\ 8\lg(1-k)\lg\frac{2n'}{kF-2}\end{matrix}}}{2\lg(1-k)} \quad (1)$$

Let $h(k, n', F) \triangleq h$, it holds: $h(k, n', F) \leq h(k, n, F)$.

From formula (1) and given n records, we know that the maximum height of $k\mu$-tree depends on k value and max fanout F, while height h and value k is in proportion. The larger the k value is, the bigger the space will be allocated to the lower level tree nodes, especially the leaf nodes. In $k\mu$-tree, one flash page may hold all the nodes of the path from root to leaf, consequently the space (pages) of $k\mu$-tree, denoted as $S_{k\mu-Tree}$ is equal to the number of leaves. Since the size of a leaf node is k fraction of the flash page, and the leaf node's fanout is kF, therefore the space (pages) of $k\mu$-tree can be approximated as $S_{k\mu-Tree} \approx \left\lceil \frac{n}{kF} \right\rceil$. That is, $S_{k\mu-Tree}$ is inversely proportional to k, the larger the k value the smaller the $k\mu$-tree size. On the other hand, if k is too large, then the upper level nodes do not have enough space to organize the $k\mu$-tree.

In reality, applications usually have the knowledge of upper limit of records or the system has limited secondary storage, therefore n can be pre-defined. The index entry size depends on the application, so does the max fan-out. By feeding these two parameters to the index system, the max k value according to (1) can be found easily.

We demonstrate the relationships among k, n, h, and F in Figs 8 and 9. Fig. 8 shows the relationship between n, k, h, and F=512. Fig. 9 (a) illustrates the relationships between the optimal value k and the number of records by fixing the max fanout, while Fig. 9 (b) shows the relationships between the optimal value k and index entry size when n is fixed. As the number of records increases or the index entry size increases, the suitable k value decreases.

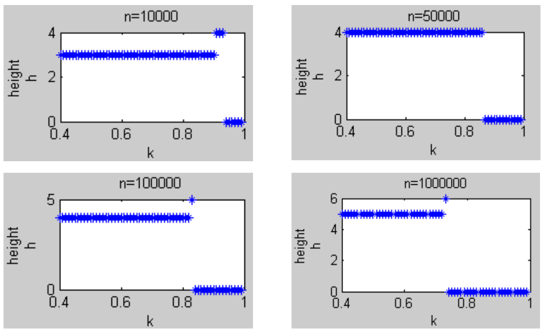
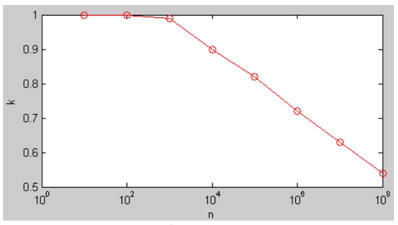
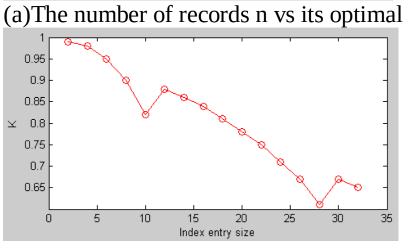


Figure 8. The relationship between k and h (h=0 is caused by too big k to have real h solution).



(a)The number of records n vs its optimal *k*



(b)The index entry size vs the optimal value k

Figure 9. the optimal value k vs the number of records, the index entry size.

Due to the *k-partitioning* strategy, the node of $k\mu$-tree will not take an entire page. The algorithm for getting a node within a page for $k\mu$-tree is shown in Fig. 10. Lines 1 to 7 computes the offset for locating the node and the size for the node.

V. Evaluation

In this section, we introduce the results of the experiments, with which we have implemented the μ-tree with our enhanced buffer scheme on a NAND flash simulator [23] to evaluate the effectiveness of our proposed buffer scheme and the effect of the *k-partitioning* method.

A. Experimental Setup

All the experiments of this work were conducted on a PC with an Intel dual-core processor, 2GB memory, and Linux with kernel 2.6.28.

Simulator. NAND simulator (nandsim) [23] is an extremely useful debugging and development tool which simulates NAND flashes in RAM or a file. We used NAND simulator to simulate a 1GB flash memory with 2KB page size by specifying the simulated flash type with parameters: first_id_byte=0xec second_id_byte=0xd3 third_id_byte=0x51 fourth_id_byte=0x95. Compared with

```
GetNodeFromPage ()
Input: Page address D, Level L
Output: N
1: if L=h then
2:     S← $\lceil (1-k)^{h-1} p \rceil$    /*S-the node size*/
3:     O←0                                  /*O-page offset*/
4: else
5:     S← $\lceil k(1-k)^{L-1} p \rceil$
6:     O← $\lceil (1-k)^{L} p \rceil$
7: end if
8: N←read at page D from offset O with size S
9: return N
```

Figure 10. The algorithm for getting a node within a page in $k\mu$-tree.

the access time of a flash page, the access time of in-memory page is negligible. Thus, we only present the number of buffer misses m, i.e., the number of flash memory accesses, during the key search.

In our experiments, we investigate the performance of our proposed techniques by varying the number of query requests, the buffer size, and query access patterns generated according to random (uniform), normal and zipf distributions of the search key values; we'll also look into the effect of tree height, buffer size, and the partition value k on the performance of the buffer scheme. To see the effect of the partition value k on μ-tree space, we'll demonstrate the space performance of μ-tree by varying the index entry size. By going one step further, we validate the results by using the real-world data from the UMass Trace Repository[1].

Datasets. Since we focus on the index search performance, the query sequences (access patterns) were generated by varying the distributions on the key values, including random (uniform) distribution, normal distribution (μ = maximalKeyValue/2, σ = 500), and Zipf distribution (skew=0.5). The length of query sequences ranges from 10^4, 10^5, 2×10^5, 5×10^5, 10^6. μ-trees with different tree heights (hence different number of index entries accordingly) also are used to show what effect of tree height will have on our proposed buffer scheme. The tree height ranges from 3 to 5.

To validate the simulation result, we use the storage data sets of the UMass Trace Repository as our data sets, where there are two I/O traces from OLTP applications running at two large financial institutions. Since we have limited nand flash storage, here we map 10,000 logical I/O block address of the trace access patterns as our real query access patterns for μ-trees index while we filter out those blocks falling out of this scope. The data sets and their requests are shown in Table I.

Metrics. We used the number of index page accesses to evaluate the effectiveness of different buffer schemes

[1]http://traces.cs.umass.edu/. It provides network, storage, and other traces to the research community for analysis.

TABLE I.
NUMBER OF REQUESTS.

data sets	requests
Financial 1	49406
Financial 2	78800
Websearch 1	27984
Websearch 2	26934
Websearch 3	26772

Query reqs (*10^4) / Query Pattern	1	10	20	50	100
Random	15590	155910	312038	780047	1559907
Normal (μ=N/2)	8440	84557	168774	421593	843000
Zipf (skew=0.5)	6323	62601	125389	313557	627857

(a)LRU-N

Query reqs (*10^4) / Query Pattern	1	10	20	50	100
Random	17055	170927	342033	854324	1708855
Normal (μ=N/2)	11204	112373	224522	560877	1122580
Zipf (skew=0.5)	8789	87335	174897	438076	877579

(b)LRU-P

Figure 11. Comparisons between LRU-N and LRU-P under various distributions (k=0.5,buffer=4P,H=3,N=10000).

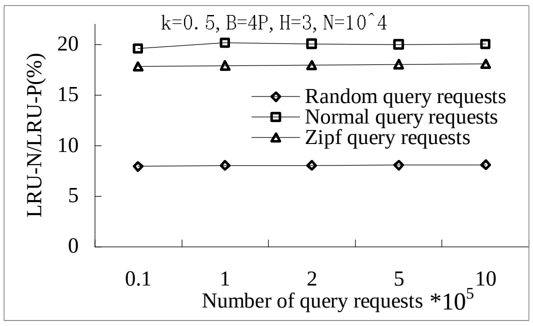
(i.e.,the conventional page based LRU-P vs our proposed node based LRU-N), the occupied index space size in flash storage to compare the space efficiency of the k-partitioning method, the estimated elapsed time to compare the overall performance. Here, we define the improvement brought by our proposed technique as $\frac{|v_{LRU-P}-v_{LRU-N}|}{v_{LRU-P}} \times 100\%$ where v_{LRU-N} and v_{LRU-P} are the metric value (the number of index page accesses, the index space size, and the elapsed time) with and without our techniques, respectively.

For simplicity, we use the following notations in all the illustrated diagrams: k for k-partition value, H for tree height, B for buffer size, P for page, N for the number of index entries in the index. For the convenience of statement, when we say a query requests conforming to random distribution, we'll use *Random queries* for short; similarly, we'll use *Normal queries* and *Zipf queries* for the query requests of normal distribution and Zipf distribution respectively.

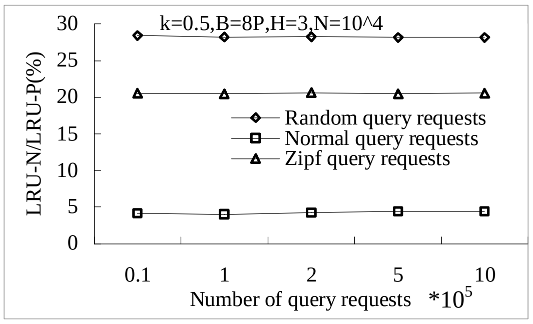
B. Results on LRU-N

For the first set of experiments, we set the page size of the flash memory to be 2048 bytes. The key value is eight bytes, consisting of a 4-byte key and a four-byte pointer. The buffer size is set to 4 pages (8KB).

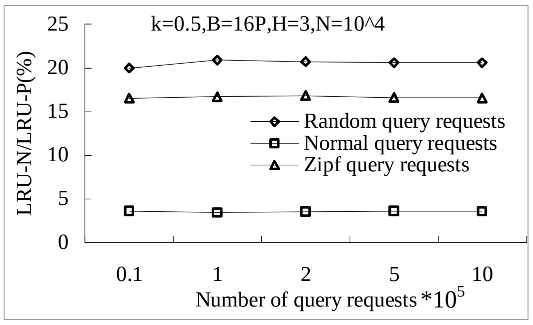
Fig. 11 (a) - (c) report the number of page accesses to the flash memory of μ-tree with the LRU-P and the LRU-N replacement policies. By using query requests of different data distributions on the search key values, we varied the length of query requests and fixed partition value k to 0.5, buffer size B to 4 pages, tree height H to 3,



(a) B=4P



(b) B=8P



(c) B=16P

Figure 12. Percentage improvement of LRU-N over LRU-P under various buffer size (k=0.5, H=3, N=10000).

the number of index entries N to 10000. From the results, we can observe that overall the LRU-N replacement policy improves the buffer effectiveness, reducing the number of page accesses from 8% to 20%. We further analyze the results on various distributions:

- ◇ **Random distribution:** the improvement of LRU-N over LRU-P is about 9%. This trend keeps stable for different lengths of query requests. Since we only use a small buffer size (4P), achieving this improvement is not easy.
- ◇ **Zipf distribution:** Given the same data scale, the number of page accesses for *Zipf queries* is smaller than that for *Random queries*. This is because the search key values in *Zipf queries* is more skew than that in *Random queries*, especially for the key values with larger weight, which results in more buffer reuse. The improvement of LRU-N over LRU-P is about 18%.
- ◇ **Normal distribution:** the improvement of LRU-N

for *Normal queries* achieves more than 20%.

To see the above trends clear, we redraw the above results in Fig. 12 (a), where we show the percentage improvement of LRU-N over LRU-P for each kind of query requests.

We performed next set of experiments by varying the buffer size. Fig. 12 (a) through (c) shows the effect of buffer size on the performance of LRU-N. As buffer size increases from 4 pages to 16 pages, we witnessed an interesting performance changes of LRU-N:

◇ **Random distribution:** the improvement increases from 8% for 4P, 28% for 8P and 20% for 16P. Note that both performance of LRU-N and LRU-P increases as the larger buffer size results in more index pages can be buffered.

◇ **Zipf distribution:** For query requests of this distribution, the improvement of LRU-N over LRU-P varies from 18% for 4P, 21% for 8P and 17% for 16P.

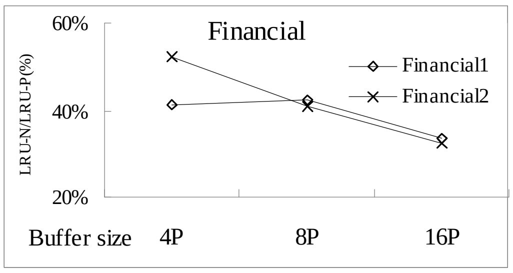
◇ **Normal distribution:** the improvement of LRU-N in the normal distribution witnessed an interesting variation from 20% for 4P, about 4% for 8P and 16P. For this distribution, both performance of LRU-N and LRU-P increases as buffer size increases but their ratio is not a simple linear relationship.

In the mean time, we also validate our new buffer scheme on the UMass Trace and the results are shown in Fig. 13 and Fig. 14. By varying the buffer size from 4 pages to 16 pages, LRU-N shows an improved hit rates over LRU-P about 20% to 65%. The reason is that LRU-N makes use of the memory space wasted by invalid index nodes and considers both the temporal and spacial locality of the access patterns while its counterpart LRU-P fails to do so. Again in this real-world trace, we noticed the gradual drop-down of the LRU-N/LRU-P ratio as the buffer size increases. LRU-N can perform excellently even with small buffer size. In whatever case, LRU-N always gives better hit rate.

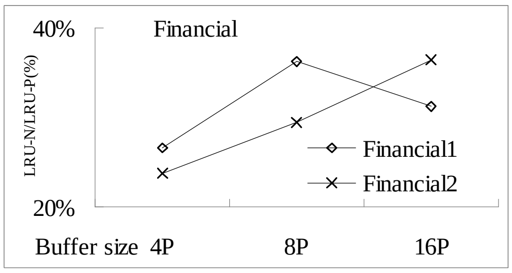
To understand the effect of tree height on the performance improvement of LRU-N, we carried out another set of experiments by varying the number of index entries and the buffer size. Fig. 15 (a) - (c) gives the results. We can observe the general trends. For small buffer size 4P, LRU-N outperforms LRU-P by 20% at H=3, 52% at H=4 and 43% at H=5 for *Normal queries*; For larger buffer size 16P, LRU-N outperforms LRU-P by 4% at H=3, 10% at H=4, and 25% at H=5. An outstanding fact is that for all access patterns LRU-N outperforms LRU-P by more than 21% at H=5 and buffer size 16P. Similar results are also observed for *Random queries* (8% to 28%)and *Zipf queries* (10% to 20%). In this set of experiments, we found that even with 4 pages small buffer size, LRU-N outperforms LRU-P by 9 - 16% for *Random queries*, 20 - 52% for *Normal queries*, and 10 - 17% for *Zipf queries*.

C. Impact of k-Partitioning on the Buffer Scheme

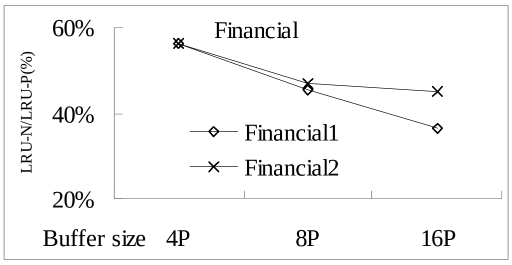
Now let's look into the performance impact of the various partition value k. We performed experiments on



(a) K=0.3



(b) K=0.5



(c) K=0.7

Figure 13. Percentage improvement of LRU-N over LRU-P under various buffer size and k values (Financial, N=10000).

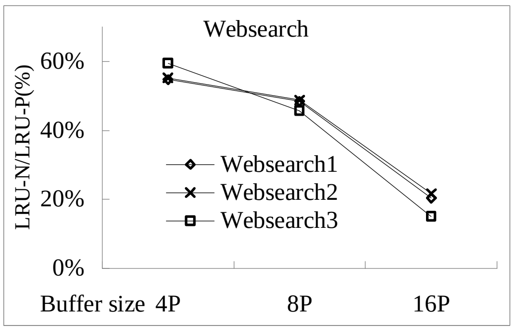
k=0.3, k=0.5 and k=0.7. The experimental results are shown in Fig. 16 (a) - (c). As stated above, for small buffer size 4P, the performance of *Normal queries* or *Zipf queries* is better than that of *Random queries*. On the other hand, for larger buffer size, the performance of *Random queries* or *Zipf queries* is better than that of *Normal queries*. It reaches up to 24%.

With respect to real-world data sets, it can be inferred from Fig. 17 and Fig.18 that, as k increases, the percentage of improvement tends to increase as well. The improvement of LRU-N over LRU-P ranges from 15% to 65%. For small buffer size, LRU-N performs significantly better than LRU-P. Such a feature is definitely desired for resource restricted embedded system.

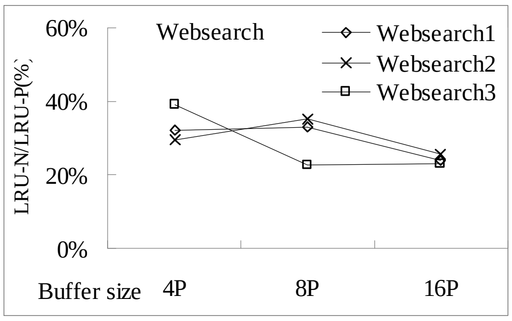
D. Space efficiency of K-Partitioning

In this section, we'll see the effect of k value on the space performance of μ-tree. For small flash page size, a big k will result in limited number of records indexed. The relationship between k and tree height has been detailed in Section IV and Fig. 8.

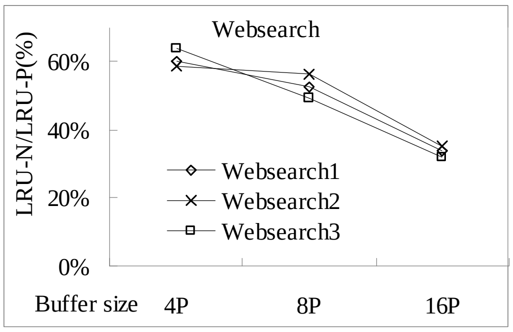
In this set of experiments, the flash page size is set to 2KB as before, k values take 0.7 and 0.5, and the numbers of indexed entries varies from 5000, 10000, 20000 and 30000. Fig. 16 (a) - (b) shows the space usage of μ-tree



(a) K=0.3

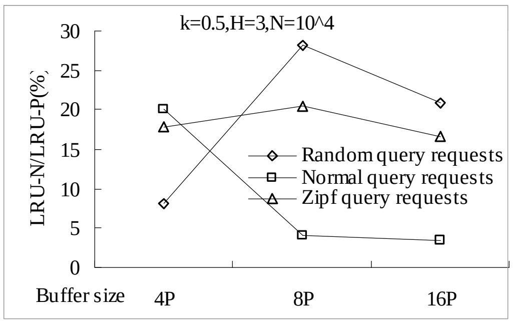


(b) K=0.5

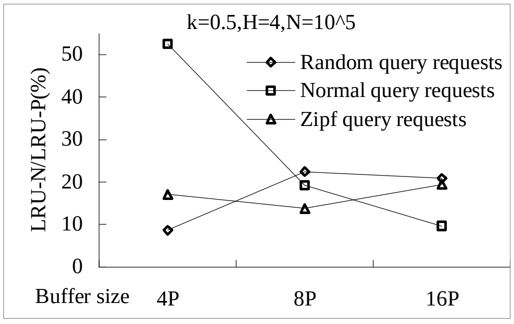


(c) K=0.7

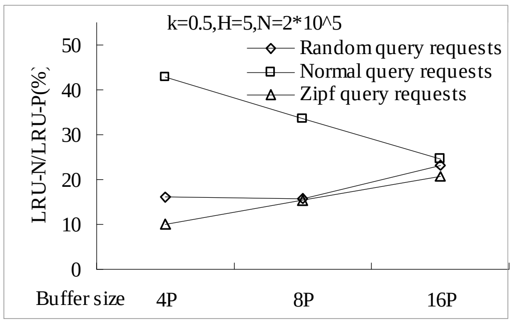
Figure 14. Percentage improvement of LRU-N over LRU-P under various buffer size and k values (WebSearch, N=10000).



(a)H=3



(b)H=4



(c) H=5

Figure 15. Percentage improvement of LRU-N over LRU-P under various tree height (k=0.5).

and $k\mu$-tree by varying the number of indexed entries and the index entry size respectively. As the number of indexed entries varies, the improvement in the index size of $k\mu$-tree over μ-tree ranges from 25% to 35%; Similarly, for the index entry sizes, the space improvement is around 20% to 33%.

VI. CONCLUSION

μ-tree is the state-of-the-art flash-aware tree index for flash-based embedded database systems. In this paper, we investigated the novel buffer scheme to enhance the μ-tree query performance. We proposed a node-based buffer replacement policy LRU-N and a *k-partitioning* method to improve the buffer effectiveness and the space efficiency for μ-tree, respectively. We evaluated our techniques using workloads under various scales and distributions. The experimental results demonstrate that our node-based buffer scheme is very effective and has good performance. It reduces the number of page accesses up to 52% in the best cases compared with that of the LRU-P scheme. It is worth to note that, even with 4 pages small buffer size, LRU-N outperforms LRU-P by 9 - 16% for *Random queries*, 20 - 52% for *Normal queries*, and 10 - 17% for *Zipf queries*; with regards to real-world data sets, LRU-N outperforms LRU-P by 26 - 56% for Financial data set and 29 - 65% for WebSearch data set. This characteristics is very useful for embedded applications with memory restrictions. In addition, the *k-partitioning* method reduces the index size up to 33%.

ACKNOWLEDGEMENTS

Here, we would like to give thanks to Dr. Bingsheng He (NTU) and Prof. Lei Chen (HKUST) for their insightful discussions. And we are grateful to the anonymous referees for their valuable comments on the presentation of the paper.

REFERENCES

[1] Agrawal D, Ganesan D, Sitaraman R, Diao Y. Lazy-Adaptive Tree: An Optimized Index Structure for Flash

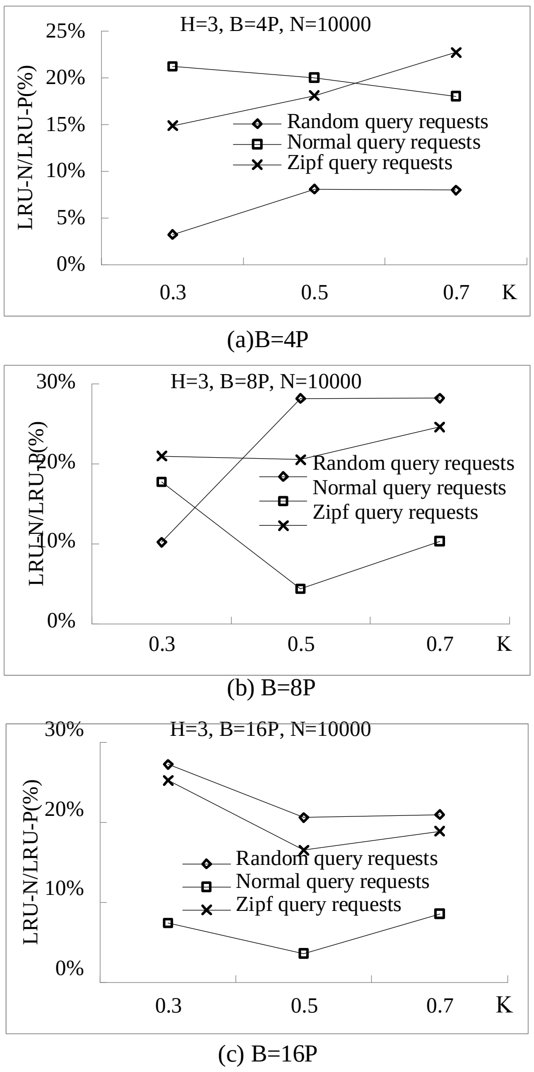


Figure 16. Percentage improvement of LRU-N over LRU-P under various k values and buffer sizes (H=3,N=10000).

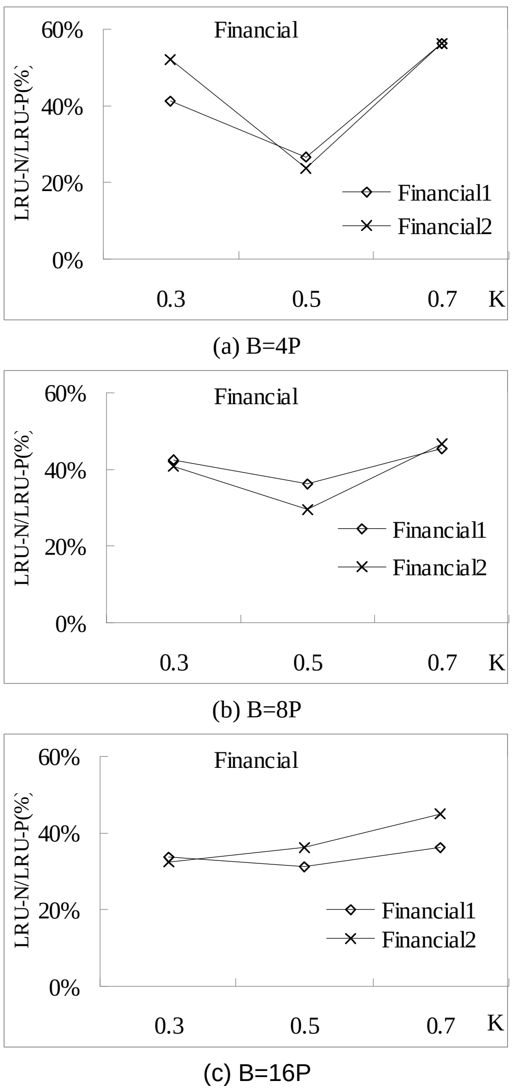
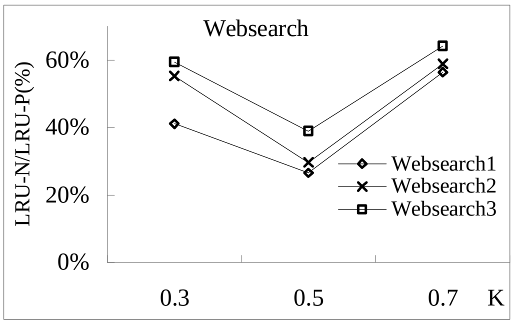
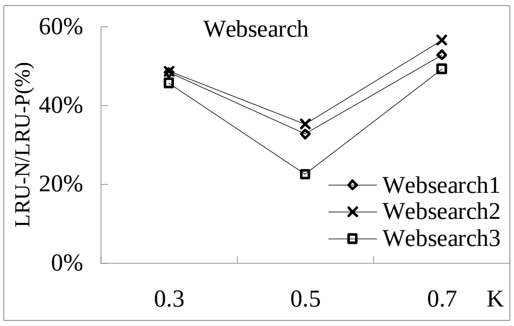


Figure 17. Percentage improvement of LRU-N over LRU-P under various k values and buffer sizes (Financial,N=10000).

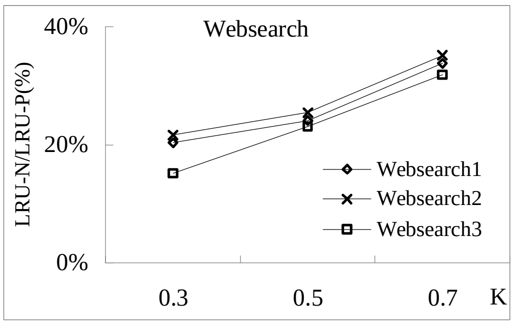
Devices. Int’l Conf. on Very Large Data Bases, 2009, pp. 361-372.
[2] Chang L P, Hsu C H. Soft lists: a native index structure for NOR-flash-based embedded devices. Asia and South Pacific Design Automation Conf., 2009, pp. 799-804.
[3] Bityutskiy A B. JFFS3 design issues.
[4] Jiang S, Ding X, Chen F, Tan E, and Zhang X. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. USENIX Conf. on File and Storage Technologies, 2005, pp. 8-8.
[5] Jo H, Kang J, Park S, Kim J, and Lee J. FAB: Flash-Aware Buffer Management Policy for Portable Media Players. IEEE Transactions on Consumer Electronics, 2006, 52(2):485-493.
[6] Kim H and Ahn S. BPLRU: a buffer management scheme for improving random writes in flash storage. USENIX Conf. on File and Storage Technologies, 2008, pp. 239-252.
[7] Kim G J, Baek S C, Lee H S, Lee H D, and Joe M. LGeDBMS: A Small DBMS for Embedded System with Flash Memory. Int’l Conf. on Very Large Data Bases, 2006, pp. 1255-1258.
[8] Kang J U, Jo H, Kim J S, Lee J. A Superblock-based Flash Translation Layer for NAND Flash Memory. Int’l Conf. on Embedded Systems Software, 2006, pp. 161-170.
[9] Kang D, Jung D, Kang J, Kim J. -Tree: An Ordered Index Structure for NAND Flash Memory. Int’l Conf. on Embedded Software, 2007, pp. 144-153.
[10] Kawaguchi A, Nishioka S, and Motoda H. A Flash-Memory Based File System. USENIX Conf. on File and Storage Technologies, 1995, pp. 155-164.
[11] Lee S W, and Moon B. Design of Flash-Based DBMS: An In-Page Logging Approach. Int’l Conf. on Management of Data, 2007, pp. 55-66.
[12] Lee M, Seo E, Lee J, and Kim J. PABC: Power-Aware Buffer Cache Management for Low Power Consumption. IEEE Transactions on Computers, 2007, 56(4):488-501.
[13] Mani A, Rajashekhar M B, and Levis P. TINX - A Tiny Index Design for Flash Memory on Wireless Sensor Devices. ACM Conf. on Embedded Networked Sensor Systems(SenSys), 2006, pp. 425-426.
[14] S. Nath, and A. Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. Int’l Conf. on Information Processing in Sensor Networks, 2007, pp. 410-419.
[15] Park S, Jung D, Kang J, Kim J, and Lee J. CFLRU: A Replacement Algorithm for Flash Memory. Int’l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems, 2006, pp. 234-241.
[16] Park C, Kang J, Park S, and Kim J. Energy-Aware De-



(a) B=4P

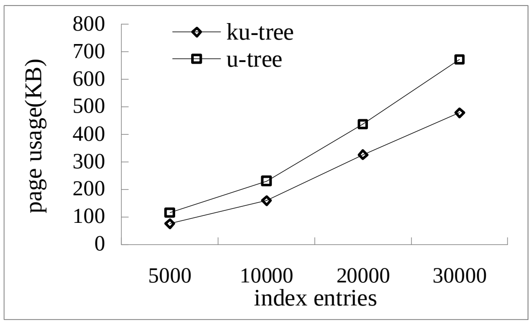


(b) B=8P

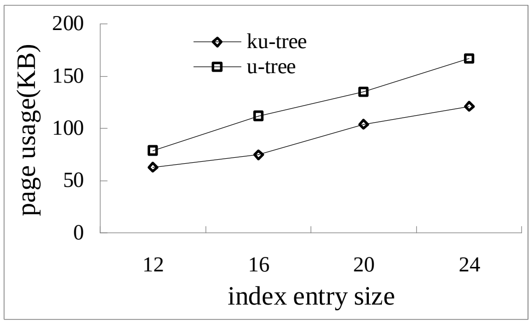


(c) B=16P

Figure 18. Percentage improvement of LRU-N over LRU-P under various k values and buffer sizes (WebSearch,N=10000).



(a)varying the index entries



(b)varying the index entry size

Figure 19. Space comparison between μ-tree and $k\mu$-tree.

mand Paging on NAND Flash-based Embedded Storages. IEEE/ACM Int'l Symposium on Low Power Electronics and Design, 2004, pp. 338-343.

[17] Sen R and Ramamritham K. Efficient Data Management on Lightweight Computing Device. Int'l Conf. on Data Engineering, 2005, pp. 419-420.

[18] Wu C, Chang L, Kuo T. An efficient R-tree implementation over flash-memory storage systems. Int'l Symposium on Advances in Geographic Information Systems, 2003, pp. 17-24.

[19] Wu C, Chang L, and Kuo T. An Efficient B-Tree Layer for Flash-Memory Storage Systems. ACM Transactions on Embedded Computing Systems, 2007, 6(3):1-23.

[20] Yazti D Z, Lin S, Kalogeraki V, Gunopulos D, and Najjar W. MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices. USENIX Conf. on File and Storage Technologies, 2005, pp. 31-44.

[21] Yin S, Pucheral P, Meng X. PBFilter: Indexing Flash-Resident Data through Partitioned Summaries. Int'l Conf. on Information and Knowledge Management, 2008, pp. 1333-1334.

[22] Yin S, Pucheral P, Meng X. A sequential indexing scheme for flash-based embedded systems. Int'l Conf. on Extending Database Technology, 2009, pp. 588-599.

[23] http://www.linux-mtd.infradead.org

Liang Huai Yang received the BSc. degree in Information Science (Department of Mathematics) in 1989 and the PhD degree in Computer Science from Peking University in 2001. He assumed a research fellow position at National University of Singapore during 2001 and 2005. He is now a professor at Zhejiang University of Technology. He has published about 40 papers in major conferences and journals in the database field. He has served on the program committee of some database conferences, and as reviewers of some journals such as Information Sciences, Information Systems, International Journal of Electronics and Computers, etc.

Jing Wang is currently a graduate student in Zhejiang University of Technology. His research focuses on energy-efficient multi-core computing.

Zhifeng Huang received his B.S.degree from Zhejiang University of Technology. His research interests include access methods in DBMS, 3G-4G wireless mobile communication. He is currently holding the engineer position at Nokia Siemens Networks.

Weihua Gong, Ph.D., associate-professor at Zhejiang University of Technology. His research interests include data mining, database system and p2p computing, etc.

Li Jun Chen received the BSc. degree in Application Mathematics in 1991 and the PhD degree in Information Science from Peking University in 1998. He is now a Associate Professor at School of Electronics Engineering and Computer Science, Peking University. He has published about 20 papers in major conferences and journals in the database field. His research interest includes Database management system implement technology and Data stream processing.