An Efficient R-Tree Implementation over Flash-Memory Storage Systems*

Chin-Hsien Wu, Li-Pin Chang, Tei-Wei Kuo
Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan 106, ROC
Fax: +886-2-23628167
{d90003,d6526009,ktw}@csie.ntu.edu.tw

ABSTRACT

For many applications with spatial data management such as Geographic Information Systems (GIS), block-oriented access over flash memory could introduce a significant number of node updates. Such node updates could result in a large number of out-place updates and garbage collection over flash memory and damage its reliability. In this paper, we propose a very different approach which could efficiently handle fine-grained updates due to R-tree index access of spatial data over flash memory. The implementation is done directly over the flash translation layer (FTL) without any modifications to existing application systems. The feasibility of the proposed methodology is demonstrated with significant improvement on system performance, overheads on flash-memory management, and energy dissipation.

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Real-time and embedded systems; H.3.1 [**Content Analysis and Indexing**]: Indexing methods

General Terms

Design, Performance, Algorithm

Keywords

Flash Memory, GIS, R-Tree, Storage Systems, Embedded Systems, Spatial Index Structures

1. INTRODUCTION

Flash memory is now considered as an alternative for hard disks in many applications. The popularity of mobile network also triggers a new wave of mobile applications over hand-held devices. Geographic Information Systems (GIS) is one of the many popular applications over hand-held devices. One major technical question for their storage systems designs is how to efficiently access various geographic information, such as electronic maps, and store information over hand-held devices. The supports for applications with such needs of spatial index structures is highly important for the performance of embedded systems, especially when the capacity of flash memory grows rapidly in recent years.

An R-tree [6, 4] is usually implemented as non-memory-resident index structures for the access of a large collection of spatial data. The concept of R-trees was first proposed by Guttman [6], and later on an R*-tree variant was proposed by Beckman, Kriegel, Schneider, and Seeger [4], where the difference of R-trees and R*-trees is on the overlapping of bounding boxes. Insertions, deletions, and re-balancing over R-trees often cause many sectors being read and written back to the same locations. For disk storage systems, these operations are considered efficient, and R-tree nodes are usually grouped in contiguous sectors on a disk for further efficiency considerations. Implementations over disks could not be applied directly over flash memory. For example, flash memory [8, 7, 11] could not be over-written (updated) unless it is erased first. As a result, out-of-date (or invalid) versions and the latest version of data might coexist over flash memory. Furthermore, an erasable unit of a typical flash memory is relatively large, compared to the unit for reads and writes. After a certain number of sector writes, free space on flash memory would be low. Activities which consist of a series of reads/writes/erases with the intention to reclaim free space would start. The activities are called "garbage collection", which is considered as overheads in flash-memory management. Note that writes and erases over flash memory take more time than reads. Frequent erasing of some particular locations of flash memory could quickly deteriorate the overall lifetime of flash memory, because each erasable unit has a limited cycle count on the erase operation. Any direct application of an R-tree disk implementation over flash memory could result in a severe performance degradation and significantly reduce its reliability. For example, intensive byte-wise operations on R-trees, due to object inserting, object deleting, and R-tree reorganizing, could result in a large number of data copyings.

In this paper, we target an essential problem in the design of a mobile system which needs an intelligent management

*Supported in part by a research grant from the National Science Council under Grant NSC 91-2213-E-002-070 and a research grant from the Academia Sinica.

GIS'03, November 7–8, 2003, New Orleans, Louisiana, USA.

of spatial objects. We explore efficient R-tree implementations for better performance and energy consumption on mobile devices. In the implementation, a reservation buffer and a node translation table are proposed to reduce the number of unnecessary and frequent updates of information over flash-memory storage systems[1]. The implementation is over the flash translation layer (FTL) [3, 2] for the compatibility of applications and systems, where FTL provides block-device emulation. Note that the concept of sectors are provided over FTL. As a result, FTL-based flash-memory storage systems could be easily ported to DOS-like operating systems. When an R-tree node is inserted, deleted, or modified, any newly generated objects would be temporarily held by the reservation buffer, where an object is a entry in an R-tree node. Since the reservation buffer only holds a limited number of objects, these objects should be flushed to flash memory in a timely fashion. We show that the proposed methodology could not only significantly improve the system performance but also reduce the overheads of flash-memory management and energy dissipation.

The rest of this paper is organized as follows: Section 2 provides an overview of flash memory. Section 3 provides the problem formulation. Section 4 introduces the R-tree implementation. Section 5 provides performance analysis of the approach. Section 6 shows experimental results. Section 7 is the conclusion.

2. FLASH MEMORY CHARACTERISTICS

A NAND[2] flash memory is organized by many blocks, and each block is of a fixed number of pages. A block is the smallest unit of erase operation, while reads and writes are handled by pages. The typical block size and page size of a NAND flash memory is 16KB and 512B, respectively. Because flash memory is write-once, we do not overwrite data on update. Instead, data are written to free space, and the old versions of data are invalidated (or considered as dead). The update strategy is called "out-place update". In other words, any existing data on flash memory could not be over-written (updated) unless it is erased. The pages store live data and dead data are called "live pages" and "dead pages", respectively. Because out-place update is adopted, we need a dynamic address translation mechanism to map a given LBA (logical block address) to the physical address where the valid data reside. Note that a "logical block" usually denotes a disk sector. To accomplish this objective, a RAM-resident translation table is adopted. The translation table is indexed by LBA's, and each entry of the table contains the physical address of the corresponding LBA. If the system reboots, the translation table could be re-built by scanning the flash memory. Figure 1 illustrates the retrieval of data over flash memory in terms of the translation table. The focus of this paper will be on NAND flash because it is designed mainly for storage systems.

After a certain number of page writes, free space on flash memory would be low. Activities consist of a series of reads,

[1]Similar data structures are adopted for B-tree management in [10]. However, with a different nature of R-trees, compacting of R-tree nodes and packing of index units over flash-memory pages would be very different.

[2]There are two major types of flash memory in the current market: NAND flash and NOR flash. The NAND flash memory is specially designed for data storage, and the NOR flash is for EEPROM replacement.

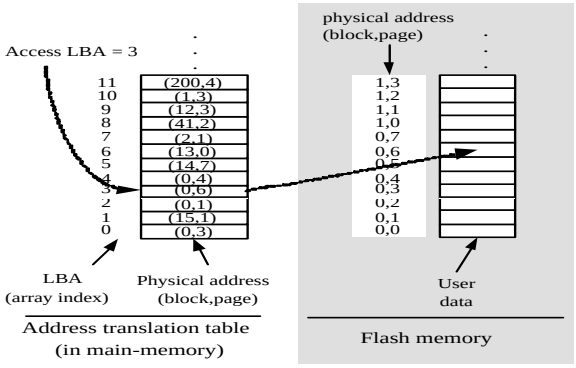


Figure 1: The logical block address "3" is mapped to the physical address "(0,6)" by the translation table.

writes, and erases with the intention to reclaim free spaces would then start. The activities are called "garbage collection" and considered as overheads in flash-memory management. The objective of garbage collection is to recycle dead pages scattered over blocks so that they could become free pages after erasings. How to smartly choose blocks for erasing is the responsibility of a *block-recycling policy*. The block-recycling policy should try to minimize the overheads of garbage collection (caused by live data copyings). Under the current technology, a flash-memory block has a limitation on the erase cycle count. For example, each block of a typical NAND flash memory could be erased for 1 million (10^6) times. A worn-out block could suffer from frequent write errors. The "wear-levelling" policy should try to erase blocks over flash memory evenly so that a longer overall lifetime could be achieved. Note that wear-levelling activities could impose significant overheads over the flash-memory storage systems if the access patterns has a strong locality on updates.

3. PROBLEM FORMULATION

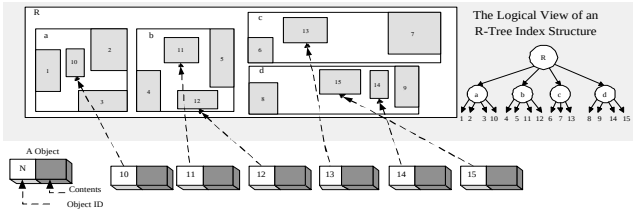


Figure 2: An R-Tree (the max fanout = 5).

Suppose that six spatial objects will be inserted to the R-tree shown in Figure 2. As shown in Figure 2, there is an R-tree with an internal node R and four external nodes a, b, c, and d. The minimal bounding box of node R contains the minimal bounding boxes of nodes a, b, c, and d. The hierarchical structure of the R-tree is shown at the right-hand side of Figure 2.

The insertions should check up the minimal bounding boxes of nodes in the R-tree. The 1st object is inserted to node a, the 2nd object and the 3rd object are inserted to node b, the 4th object is inserted to nodes c, and the 5th

object and the 6th object are inserted to node d, as shown in Figure 2. Let each R-tree node be stored in one page. Six updates of R-tree nodes (i.e., six page updates) occur. Such modifications of internal or external nodes are considered efficient over disk-like storage media because updates of nodes are usually localized on the corresponding nodes. Because of the characteristics of flash memory, updates of nodes must be done with out-place writings. Even though only a small portion of a node is modified, information stored on the node must be invalidated, and a new page must be found for the update. The out-place updates for R-tree operations could result in the consumption of free pages and, in many cases, quickly trigger garbage collection. Garbage collection might also increase the energy consumption because writes and erases consume much more energy than reads, as shown in Table 1. The maintenance of R-trees over flash memory is complicated, especially when rebalancing (such as splitting or merging) might occur. Note that rebalancing could cause many pointers in nodes being updated. As a result, much more free space is consumed when rebalancing is needed, and garbage collection is more frequent.

These observations motive the research on R-tree implementations over flash memory. The objectives are to not only improve the system performance but also reduce the energy consumption.

Table 1: Performance of a Typical NAND Flash Memory

	Page Read 512 bytes	Page Write 512 bytes	Block Erase 16K bytes
Performance(μs)	50	200	1,881
Energy Consumption	99 ($\mu joule$)	237.6 ($\mu joule$)	422.4 ($\mu joule$)

4. THE R-TREE IMPLEMENTATION OVER FLASH-MEMORY STORAGE SYSTEMS

4.1 Overview

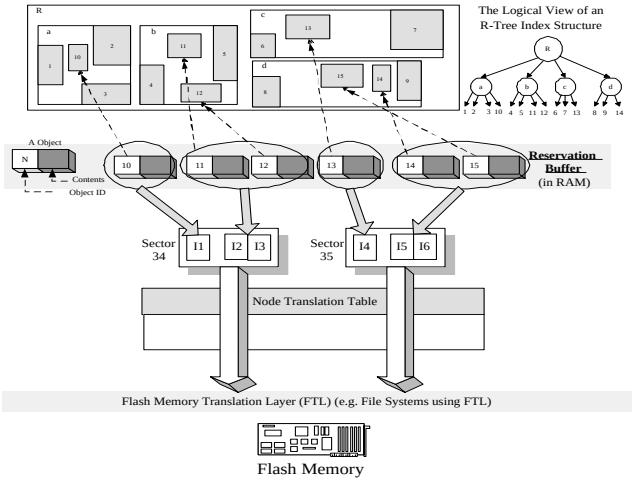


Figure 3: System Architecture

The implementation of R-trees should be independent of the design of FTL and applications, as shown in Figure 3. The proposed R-tree implementation shall consider the characteristics of flash memory. The objective is to provide transparent and efficient accesses over R-tree index structures on flash-memory storage systems to reduce the number of unnecessary writes and to improve the system performance.

When an insertion or deletion request is received from an application, an "object" which contains the corresponding operation and data is created to denote the request. The object will be temporarily held by the reservation buffer. Note that objects for deletions and insertions are created and handled in the same way. The reservation buffer is a write buffer residing in the main memory, as shown in Figure 3. All objects in the reservation buffer will be written to the flash memory in an on-demand fashion (The operation will be presented in Section 4.3.1).

Each object in the reservation buffer has two parts: meta data and data. The meta data of an object contains the minimum bounding box, operation, and pointers for housekeeping. Note that most of the meta data mentioned in the previous statement are traditionally stored in R-tree nodes to form the index structure. A data structure called an *index unit* is used to store the meta data of an object (Please see Section 4.2). When a collection of objects were flushed from the reservation buffer, the corresponding index units will be created and packed into sectors, where each sector is a logical page on flash memory. The proposed R-tree implementation is responsible of packing index units in a small number of sectors and storing them over flash memory through FTL. Because index units in the same sector might be belonging to different R-tree nodes, a node translation table is adopted to maintain the corresponding locations of index units in an R-tree so that each R-tree node could be efficiently reconstructed during data retrieval.

4.2 Data Structures

Three data structures for the proposed R-tree implementation are presented in this section:

- **The Reservation Buffer**: The reservation buffer is a write buffer residing in the main memory. When an R-tree node is inserted, deleted, or modified, any newly generated objects would be temporarily held by the reservation buffer, where an object is a entry in an R-tree node. Objects in the reservation buffer represent operations which have not yet been applied to an R-tree.

- **Index Units**: The physical representation of an R-tree node consists of index units. Index units are created when objects in the reservation buffer are flushed to an R-tree index structure. An index unit contains necessary meta data of an object to denote any modifications to the corresponding R-tree node. An index unit consists of the following components: $data\_ptr$, $parent\_node$, $next\_node$, id, $minimal\_bounding\_box$, and op_flag. $data\_ptr$, $parent\_node$, $next\_node$, and $minimal\_bounding\_box$ are a pointer to the data, a pointer to the parent R-tree node, a pointer to the child R-tree node, and the minimal bounding box, respectively. id denotes the R-tree node to which the index unit is belonging. op_flag represents the corresponding operation, i.e., an insertion ($op\_flag = i$), a deletion ($op\_flag = d$), or an update ($op\_flag = u$).

Note that modifications to a node usually modifies only a small portion of the contents of a node. Index units are packed into few sectors by the proposed R-tree implementation for storing over flash memory to prevent the R-tree index structure from frequent updating (due to minor modifications to nodes). However, due to the packing of index units into sectors, the index units of an R-tree node might be scattered over different sectors of flash memory.

- **The Node Translation Table**: A node translation table is adopted to maintain the mapping of index units and the corresponding R-tree nodes. Since the index units of an R-tree node might be scattered over flash memory due to the proposed R-tree implementation, a node translation table is adopted to maintain the mapping of index units and the corresponding R-tree nodes so that each R-tree node could be efficiently reconstructed. The node translation table is an array of lists, where each entry of the array, i.e., a list, denotes an R-tree node. The entry of the array contains a list of the LBA's of the sectors which contain index units of the corresponding R-tree node.

4.3 Manipulations of Data Structures

4.3.1 Packing of Index Units

The proposed R-tree implementation will pack index units into sectors and store them on flash memory through FTL. The R-tree implementation should minimize the number of written sectors, where sectors are logical storage units provided by FTL.

We shall use an example to illustrates the idea: Suppose that the reservation buffer could hold up to six objects, as shown in Figure 3. Objects that correspond to the insertions of items with ID's equal to 10, 11, 12, 13, 14, and 15 are left in the reservation buffer. Since the buffer is full, the proposed R-tree implementation first transforms the 6 objects into 6 index units {I1, I2, I3, I4, I5, I6}. According to the minimal bounding boxes of the objects and the minimal bounding boxes of leaf nodes a, b, c, d, the six index units should be partitioned into four disjoint sets to avoid the scattering of the index units of an R-tree node widely over sectors: {I1} $\in a$, {I2, I3} $\in b$, {I4} $\in c$, {I5, I6} $\in d$. Suppose that each sector could contain up to 3 index units. {I1} and {I2, I3} are stored in the first sector (i.e., sector number 34). {I4} and {I5, I6} are stored in the second sector (i.e., sector number 35). The two sectors are then written to flash memory by the proposed R-tree implementation. With the traditional approach in R-tree maintenance, up to six updates of R-tree nodes (i.e., six sector writes) would be needed. The packing problem can be defined as follows:

DEFINITION 1. *The packing problem of index units:*
Given a collection T of disjoint sets of index units, a capacity constraint C of sectors, and a positive integer I, the packing problem is to find a partition of T into groups such that the total number of index units in each group is no more than C, and the number of groups in the partition is no more than I.

THEOREM 1. *The packing problem of index units is NP-Hard.*[3]

[3]A similar but simplified proof on the packing of information could also be found in [10].

Proof. The intractability of the problem could be shown by a reduction from the Bin-Packing [5] problem. The Bin-Packing problem can be defined as follows: Let S be a collection of items to be packed into bins, where the size of a bin is B, and each item x_i is of size $size(x_i)$. Given an integer J, a Bin-Packing problem instance is to partition S into groups and store each group in a bin such that the total size of a group is no larger than B, and the number of groups in the partition is no more than J.

The reduction from a Bin-Packing problem instance into a packing problem instance for index units. Let the capacity constraint C of sectors be B, and the collection of disjoint sets of index units T be S, where each item x_i denotes a set of index units, and the number of index units in a set is equivalent to the size of the corresponding item. The constraint I on the number of sectors for the packing problem of index units is equal to J. If there exists a solution for the packing problem of index units, then the solution can be directly applied to the Bin-Packing problem instance. As described above, the reduction can be done in a polynomial time. Since the Bin-Packing problem is NP-Hard [5], the packing problem of index units is NP-Hard. □

Note that the well-known FIRST-FIT approximation algorithm [9] could have an approximation bound no more than twice of the optimal solution. Insertions and deletions of an R-tree node might not only update the corresponding nodes but also result in rebalancing of the R-tree index structure. In the proposed approach, deletions are handled by having "invalidation objects" in the reservation buffer. In other words, deletions could be considered as insertions because deletions are turned into invalidation objects for the modifications of the R-tree.

4.3.2 Updating of the Node Translation Table

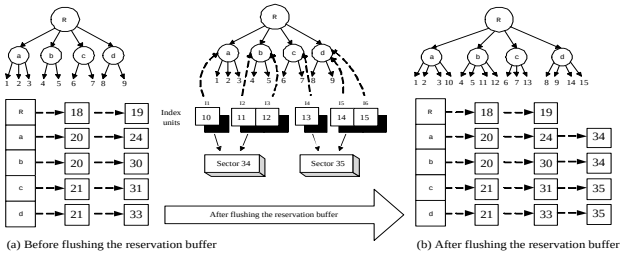


Figure 4: The situation after the flushing of the reservation buffer.

A node translation table is adopted to maintain the mapping of index units and the corresponding R-tree nodes so that each R-tree node could be efficiently reconstructed. Figure 4.(a) shows an R-tree with five nodes (one internal node and four external nodes) and its corresponding node translation table. Figure 4.(b) shows an R-tree and its node translation table after the reservation buffer in Figure 3 is flushed. When an R-tree node is visited, we collect all of the index units belonging to the visited node by scanning the sectors whose LBA's (logical block addresses) are stored in the list. For example, index units in sectors with LBA's 20, 24, and 34 must be accessed to reconstruct an R-tree node a, as shown in Figure 4.(b). On the other hand, a sector, e.g., that with LBA 20, might contain index units for more than one node, e.g., R-tree nodes a and b.

The node translation table is an array of lists, where each entry of the array, i.e., a list, denotes an R-tree node. The entry of the array contains a list of the LBA's of the sectors which contain index units of the corresponding R-tree node. The number of items in a list could result in the degradation of the system performance and the increasing of the space overheads. A system parameter ω is used to restrict the maximum number of items in lists of the node translation table. Once the number of items in a list grows over ω, the list must be compacted. The compaction of items in a list can be done by reading all sectors that contain the index units of the corresponding R-tree node and then write back to other available sectors. Note that since deletions are handled as variants of insertions, compaction also help in eliminating redundant index units. We shall use the following example to illustrate the idea and the compaction process:

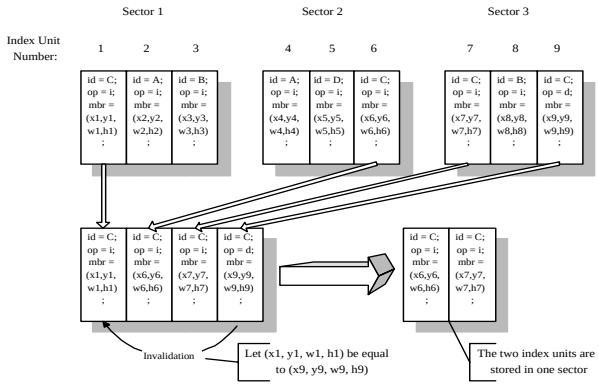


Figure 5: Compacting of an R-Tree node C.

In Figure 5, some index units of R-tree node C are scattered in three sectors, i.e., Sectors 1, 2, and 3. Let the capacity of each sector be enough for three index units. Suppose that *id*, *op*, and *mbr* denote the identifer, the op_flag, and the minimal bounding box of an index unit, respectively, where other meta data are not shown in the figure for the simplicity of explanation. Suppose there are 9 index units in the three sectors. They are numbered from 1 to 9. Let (x_i, y_i) and (w_i, h_i) denote the coordinate of the lower-left corner and the dimension (i.e., the width and the height) of the minimal bounding box of the corresponding index unit (for i=1 to 9), respectively. The compacting of the list that corresponds to R-tree node C involves the reading of the three sectors and the writing back of the index units. During the compaction process, the system find that two index units, i.e., the first and ninth index units, have the same minimal bounding box, where one is an insertion, and the other is a deletion. As a result, the deletion causes the removing of the insertions. Since only two left index units, i.e., the sixth and seventh index units, are belonging to R-tree node C, they are written into an available sector. The node translation table is updated accordingly. In Section 5, we would provide further analysis of the compaction overheads.

5. SYSTEM ANALYSIS

The purpose of this section is to provide the analysis of the number of sector accesses for insertions under the proposed R-tree implementation, compared to those under the original one. The overheads of compaction will be then explored.

5.1 Analysis of the R-Tree Implementation

Suppose that n spatial objects (with different minimal bounding boxes) are to be inserted. Consider an R-tree index with a height equal to H on flash memory, where each R-tree node can be stored in a flash-memory sector. H is bounded by $O(log_{\text{fan-out}}(m+n))$, where m is the number of objects before the n insertions occur. We shall first explore the number of sector accesses for the insertions of n spatial objects under the original R-tree implementation:

Suppose that the number of node splittings is N_{split} for the insertions of n spatial objects. The numbers of reads and writes for the insertions are $R_R = O(n * H)$ and $W_R = O(n + 3 * N_{split})$, respectively, where R_R and W_R denote the numbers of reads and writes under the original R-tree implementation. R_R is bounded by $O(n * H)$ because of the locating of the leaf node for each insertion of a spatial object. W_R is bounded by $O(n+3*N_{split})$ because n writes are needed to insert the n spatial objects into the proper leaf nodes, and $3 * N_{split}$ comes from N_{split} splittings in which each splitting consists of two writes on the split nodes and one write on the parent node.

The number of sectors read for the insertions of n spatial objects under the proposed R-tree implementation could be derived as follows: Given ω as the bound on the length of lists in the node translation table, the number of reads for the insertions are $R_{PR} = O(n*H*\omega)$, where R_{PR} and W_{PR} denote the numbers of reads and writes under the proposed R-tree implementation. R_{PR} is $O(n * H * \omega)$ because each visiting of an R-tree node might involve the traversing of a list in the node translation table. R_{PR}, compared to R_R, shows that the the proposed R-tree implementation might read more sectors in handling the insertions. In fact, the proposed R-tree trades the number of reads for the number of writes.

The number of sectors written under the proposed R-tree implementation could be derived as follows: Let the capacity of the reservation buffer be of b spatial objects. As a result, the reservation buffer would be flushed at least $\lceil n/b \rceil$ times for the insertion of n spatial objects. Let N^i_{split} denote the number of nodes splittings to handle the i-th flushing of the reservation buffer. Obviously, $\sum_{i=1}^{\lceil n/b \rceil} N^i_{split} = N_{split}$ because the R-tree index structures under the proposed R-tree implementation and the original R-tree implementation are logically identical. For each single step of the reservation buffer flushing, we have $(b+N^i_{split}*(fanout-1)+N^i_{split}*2)$ index units to commit, where $fanout$ is the maximum fanout of the R-tree. Note that the multiplication of N^i_{split} and $(fanout-1)$ in the formula denotes that each splitting will result in 2 new nodes, and the number of index units in the 2 new nodes is $(fanout-1)$. Furthermore, the splitting might result in the twice updates of the parent node, because the minimal bounding boxes of the two new nodes for the splitting have not been reflected in the parent node. Therefore, $N^i_{split} * 2$ index units are needed in the worst case. Suppose that an R-tree node could fit in a sector. That is, a sector could hold up to (fanout-1) index units. The number of sectors written by the i-th committing of the reservation buffer could be $(\frac{b}{\Lambda} + N^i_{split} + \frac{N^i_{split}*2}{\Lambda})$, where $\Lambda = (fanout-1)$. In order to flush out the reservation buffer for the insertions of n spatial objects completely, we have to write at least

$\sum_{i=1}^{\lceil n/b \rceil} (\frac{b}{\Lambda} + N_{split}^i + \frac{N_{split}^i * 2}{\Lambda}) = (\sum_{i=1}^{\lceil n/b \rceil} \frac{b}{\Lambda}) + \frac{N_{split} * (\Lambda+2)}{\Lambda}$ $\approx (\sum_{i=1}^{\lceil n/b \rceil} \frac{b}{\Lambda}) + N_{split}$ sectors. Since the proposed R-tree implementation adopts the FIRST-FIT approximation algorithm, the number of sectors written could be bounded by the following formula (the approximation bound would be no more than the twice of an optimal solution):

$$W_{PR} = O(2 * (\sum_{i=1}^{\lceil n/b \rceil} \frac{b}{\Lambda}) + N_{split}) = O(\frac{2 * n}{\Lambda} + N_{split}) \quad (1)$$

W_{PR} is apparently far less than W_R, since Λ (that is the maximum number of index units in a sector) is usually larger than 2. However, we should point out that the compaction of the node translation table might introduce some run-time overheads in compactions, as discussed in the next section.

5.2 Analysis for Node Compaction

A compaction process is to restrict the length of each list in the node translation table. In the section, we explore the overheads due to the compaction. Assume that the n spatial objects are inserted into an R-tree. Let the capacity of the reservation buffer be of b spatial objects. The reservation buffer might be flushed out at least $\lceil n/b \rceil$ times. Note that each compaction of an R-tree node will read no more than ω sectors and then compact and write them back to one sector, where ω is the maximum list length in the node translation table (Assume that an R-tree node can be contained in one sector). The number of sectors written by the compaction could be derived as follows: $R_{compact} * \sum_{i=1}^{\lceil n/b \rceil} (b + N_{split}^i) = R_{compact} * (n + N_{split})$, where $R_{compact}$ denotes the ratio of spatial-object processing (including that of split nodes) in which a compaction is resulted. Note that each flushing of the reservation buffer could produce at most $(b+N_{split}^i)$ node modifications. $R_{compact} * (n+N_{split})$ denotes the worst-case number of sectors written for the compaction during the insertions of n spatial objects. Another overheads in the compaction come from the reads of sectors for the compaction. There are no more than $R_{compact} * (n + N_{split}) * \omega$ sectors being read for the compaction. We must point out that $R_{compact} * (n + N_{split}) * \omega$ is bounded by $O(R_{PR})$. When $T_{write} * (W_{PR} + R_{compact} * (n + N_{split})) + T_{read} * (R_{PR} + R_{compact} * (n + N_{split}) * \omega) \leq T_{write} * W_R + T_{read} * R_R$, the proposed R-tree implementation outperforms the original R-tree implementation for the insertions of n spatial objects, where T_{write} and T_{read} denote the time to write and to read a sector, respectively. The preferred bound on $R_{compact}$ could also be derived based on the following formula:

$$R_{compact} \leq \frac{T_{write} * (W_R - W_{PR}) + T_{read} * (R_R - R_{PR})}{T_{write} * (n + N_{split}) + T_{read} * (n + N_{split}) * \omega} \quad (2)$$

$R_{compact}$ is actually influenced by the access pattern of insertions. When the access pattern of insertions disperses over the whole R-tree, the length of lists in the node translation table could grow fast. This is because sectors could store index units that belong to different R-tree nodes and expands lists in the node translation table. Therefore, the compaction would be activated soon. In the next section, we will discuss how the impact of the characteristics of locality on the compaction.

6. EXPERIMENTAL RESULTS

6.1 Experimental Setup and Performance Metrics

A NAND-based system prototype was built to evaluate the performance of the proposed R-tree implementation and the original R-tree implementation. For the rest of this section, let PR and R denote the proposed R-tree implementation and the original R-tree implementation, respectively.

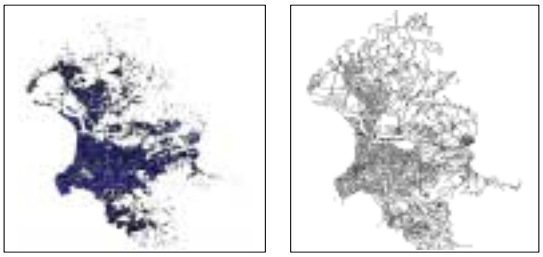


Figure 6: The Taipei map.

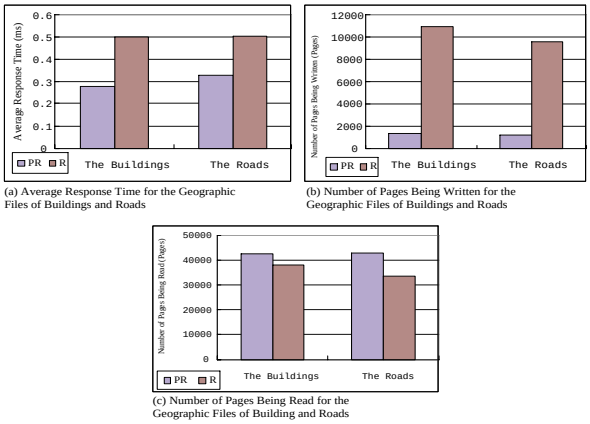
The system prototype was equipped with a 4MB NAND flash memory, where the performance of the adopted NAND flash memory was summarized in Table 1. Note that 4MB was a sufficiently large size for the experiments, because the contents of spatial objects were not stored. (Only index units created from the spatial objects were stored.) FTL was adopted to provide block-device emulation for PR and R. A *greedy* policy [8] was adopted in FTL to serve as the garbage collection policy. Two geographic files describing the roads and buildings of Taipei map were adopted as the data sets for the experiments, as shown in Figure 6. The geographic files were in the shapefile format [1]. The numbers of spatial objects of the buildings and roads were 8,590 and 7,340, respectively.

The parameters for an R-Tree were as follows: For both PR and R, the fan-out of the R-tree structure in the experiments was 16. For R, the size of an R-tree node fitted in a sector. For PR, the reservation buffer in the experiments could hold up to 80 objects (unless it was explicitly specified), and the bound on the length of each list in the node translation table was no more than 4.

The performance of PR and R were evaluated in terms of several performance metrics: the average response time of insertions and modifications (deletions) of an R-Tree index structure, the number of pages read, the number of pages written, and the number of blocks erased. Note that the average response time was calculated according to the number of pages written, read, and the number of blocks erased. We also explored the compaction overheads, the reservation buffer size and energy consumption issues. Note that sector reads/writes were issued by the upper applications, and FTL translated sector reads/writes into page reads/writes to physically access the flash memory.

6.2 Initiation Time for R-Tree Index Structures

We measured the average response time of the insertions during the creation of an R-Tree index structure, based on the two geographic files. A smaller response time denoted a better efficiency in the handling of the insertions. The average response time could also reflect the overheads, that



(a) Average Response Time for the Geographic Files of Buildings and Roads

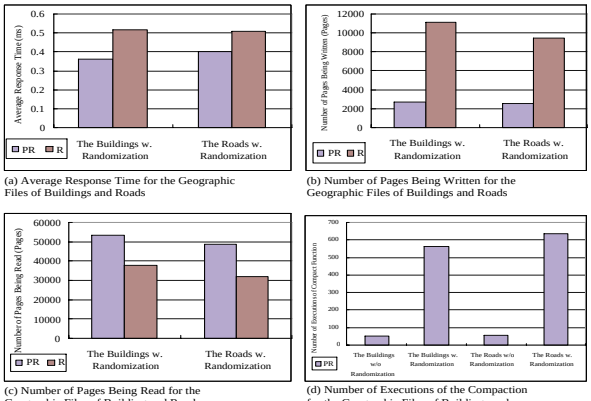
(b) Number of Pages Being Written for the Geographic Files of Buildings and Roads

(c) Number of Pages Being Read for the Geographic Files of Building and Roads

Figure 7: An R-tree index structure was entirely constructed based on the insertions of the spatial objects located in the two geographic files.

introduced garbage collection. As shown in Figure 7.(a), PR could handle the insertions more effectively than R due to a better response time. Figure 7.(b) and Figure 7.(c) showed the number of pages written and the number of pages read for the construction of R-Tree index structures. We could observe that the number of pages written under PR was even one-tenth of that under R. Since writing to flash memory could eventually introduce garbage collection activities, a smaller number of pages written was reflected to improve the response time, as mention above. Compared the number of writes and reads, as shown in Figure 7.(b) and Figure 7.(c), we could observe that PR smartly traded a larger number of reads for a reduced number of writes. We could also observe that no garbage collections occurred under PR, and there were 112 and 68 blocks erases observed under R for the indexing of the two geographic files.

6.3 Node Compaction Overheads



(a) Average Response Time for the Geographic Files of Buildings and Roads

(b) Number of Pages Being Written for the Geographic Files of Buildings and Roads

(c) Number of Pages Being Read for the Geographic Files of Building and Roads

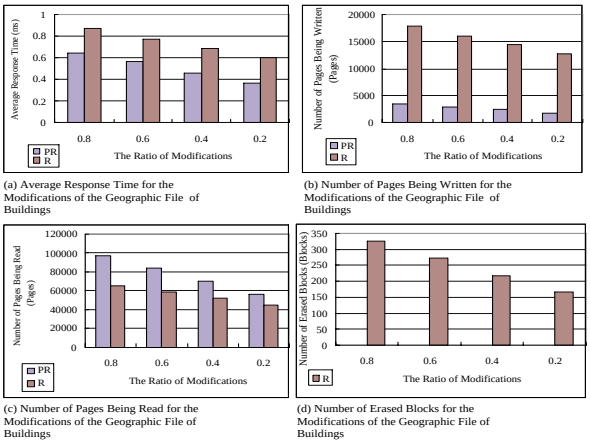
(d) Number of Executions of the Compaction for the Geographic Files of Buildings and Roads

Figure 8: The overheads of the compaction were measured.

We want to know if the spatial locality of the inserted objects could have a significant impact on the overheads for the compaction. In this part of experiments, we manipulated the spatial locality in the sequence of the inserted spatial objects. Two sequences to insert the spatial objects were considered: The first sequence is to sequentially insert the spatial objects (with high locality) according to their locations (i.e., top-down and then left-right). The second sequence is to randomly pick one spatial object to insert. Note that the second sequence could not have the characteristics of locality.

Note that the insertions for the experiments shown in Section 6.2 were in a sequential order, as mentioned above. The response time of the insertions, the number of page written, and the number of page read for the random insertion sequence under PR and R were shown in Figure 8.(a), 8.(b), and 8.(c), respectively. Compared Figure 7 with Figure 8, PR performed better when the input sequence was sequential. When the input sequence became random, the lists of the node translation table could grow fast and the compaction could be activated frequently. The compaction could impose significant overheads on the handling of the insertions. The phenomenon was also observed in Figure 8.(d), which showed the number of executions to compact a list in the node translation table under the sequential and random sequence. A strong evidence was shown that the overheads for the compaction highly could depend on the spatial locality in the insertion sequence.

6.4 Performance for Data Modi£cations



(a) Average Response Time for the Modifications of the Geographic File of Buildings

(b) Number of Pages Being Written for the Modifications of the Geographic File of Buildings

(c) Number of Pages Being Read for the Modifications of the Geographic File of Buildings

(d) Number of Erased Blocks for the Modifications of the Geographic File of Buildings

Figure 9: The performance of modifications to an R-tree index structure was measured.

An experiment parameter *modification ratio* was adopted to control the ratio of the number of the modified spatial objects to the total number of the spatial objects. Note that spatial objects were randomly chosen once to modify based on the specified *modification ratio*, and each modified object was modified once in the entire experiments. All spatial objects were first inserted and then were randomly selected to modify. There were 8,590 spatial objects located in the geographic file of the buildings of Taipei city. The average response time of the insertions and modifications, the number of pages written, the number of pages read, the number of block erased (for garbage collection) were shown in Figure 9.(a), Figure 9.(b), Figure 9.(c), Figure 9.(d), respectively.

When the *modification ratio* was increased, more spatial objects would be modified in the experiments. Because the modifications could result in byte-wise updates to the

R-tree nodes, many pages writes were needed. It was observed that PR substantially reduced the number of pages written without introducing a noticeable number of pages read. As a result, PR did'nt bring any garbage collection in the experiments due to a small number of pages written. Garbage collection that happened frequently would degrade the overall performance. As a result, PR could outperform R according to the experimental results.

6.5 Reservation Buffer Size and Energy Consumption

A large reservation buffer could have benefits by analyzing these objects, however, it could damage the reliability of the R-tree index structure due to power-failures. The Reservation buffer with different sizes was evaluated to find a reasonably good setting. We evaluated the performance of PR for the indexing of the file of buildings under different sizes of the reservation buffer. The size of the reservation buffer was set between 20 objects and 100 objects, and the size was incremented by 10 objects. The average response time was significantly reduced from 0.37 ms to 0.28 ms when the size of the reservation buffer was increased from 20 objects to 80 objects. After that, the average response time was linearly reduced from 0.28 ms to 0.24 ms and no significant improvement could be observed. Since increasing the size of the reservation buffer could damage the reliability of the R-tree index structure, the recommended size of the reservation buffer for the experiments was 80 objects.

Energy consumption is also a critical issue for portable devices. According to the numbers of reads/ writes/ erases generated in the experiments, we calculated the energy consumption under PR and R. The energy consumptions of reads/ writes/ erases are included in Table 1. The energy consumption of the two different approaches for indexing two geographical files was shown in Table 2. The energy consumed under PR was clearly less than R. Since page writes and block erases consume relatively more energy than page reads, the energy consumption was reduced when PR smartly traded extra reads for the number of writes. Furthermore, energy consumption contributed by garbage collection was also reduced under PR since it consumed free space slower than R.

Table 2: Energy consumption of the proposed R-Tree and the original R-Tree (*joule*)

Creation		
	The Proposed R-Tree	The Original R-Tree
The Buildings	4.52	6.43
The Roads	4.55	5.63

7. CONCLUSION

In this paper, we propose an efficient R-tree implementation over flash-memory storage systems. The implementation is over the flash translation layer (FTL) [3, 2] for the compatibility of applications and systems. When an R-tree node is inserted, deleted, or modified, the corresponding objects are held by the R-tree implementation. The proposed R-tree implementation then transforms objects into index units and packs units into sectors. The objective is not only to improve the performance of flash-memory storage systems but also to reduce the energy consumption of the systems, where energy consumption is an important issue for the design of portable devices. We conducted a series of experiments over a system prototype and had very encouraging results. For future research, we shall further exploit the energy consumption issue for embedded systems, especially when various application semantics is considered. Research on the joint considerations of flash-memory storage systems and system/application programs might further improve the energy dissipation of the entire system.

8. REFERENCES

[1] Esri shapefile technical description. Technical report, ESRI.

[2] Ftl logger exchanging data with ftl systems. Technical report, Intel Corporation.

[3] Understanding the flash translation layer(ftl) specification. Technical report, Intel Corporation.

[4] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The r*tree: An efficient and robust access method for points and rectangles. In *In Proc. ACM SIGMOD Intl. Symp. on the Management of Data*, pages 322–331, 1990.

[5] M. R. Garey and D. S. Johnson. *Computers and intractability.* 1979.

[6] A. Guttman. R-tree: A dynamic index structure for spatial searching. pages 45–57. In Proc. ACM SIGMOD Intl. Symp. on the Managementof Data, 1984.

[7] K. Han-Joon and L. Sang-goo. A new flash memory management for flash storage system. In *Proceedings of the Computer Software and Applications Conference*, 1999.

[8] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. USENIX Technical Conference on Unix and Advanced Computing Systems, 1995.

[9] V. V. Vazirani. *Approximation Algorithm.* Springer publisher, 2001.

[10] C. H. Wu, L. P. Chang, and T. W. Kuo. An efficient b-tree layer for flash-memory storage systems. The 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003), 2003.

[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.