A Comparison of R-tree Variants for PCM Non-volatile Memory

Elkhan Jabarov, Myong-Soon Park
College of Information and Communications
Korea University
Email: {ejabarov, myongsp}@korea.ac.kr

Byung-Won On
Department of Statistics and Computer Science
Kunsan National University
Email: bwon@kunsan.ac.kr

***Abstract*—Phase Change Memory (PCM) is a byte-addressable type of non-volatile memory. Compared to other volatile and non-volatile memories, PCM is 2∼4 times denser than DRAM and much better in read latency than NAND Flash memory. Although the write endurance of PCM is 10 times better than NAND Flash memory, it is still limited to 10^6 times per PCM cell. Thus the write endurance of PCM should be improved in order PCM to be used as an efficient main or storage memory. In this paper, we comparatively study three different types of existing R-tree variants (i.e., R-trees, R^+-trees, and R*-trees) to be used in PCM, and then attempt to understand how R-tree variants cause the write endurance problem of PCM. We also look at which one is heavily affected, compared to the others. To evenly distribute write operations over the entire memory, we would like to find which one we have to re-design and important clues to be considered when we modify some R-tree algorithms. To the best of our knowledge, this is the first work that shows how R-tree variants work in PCM.**

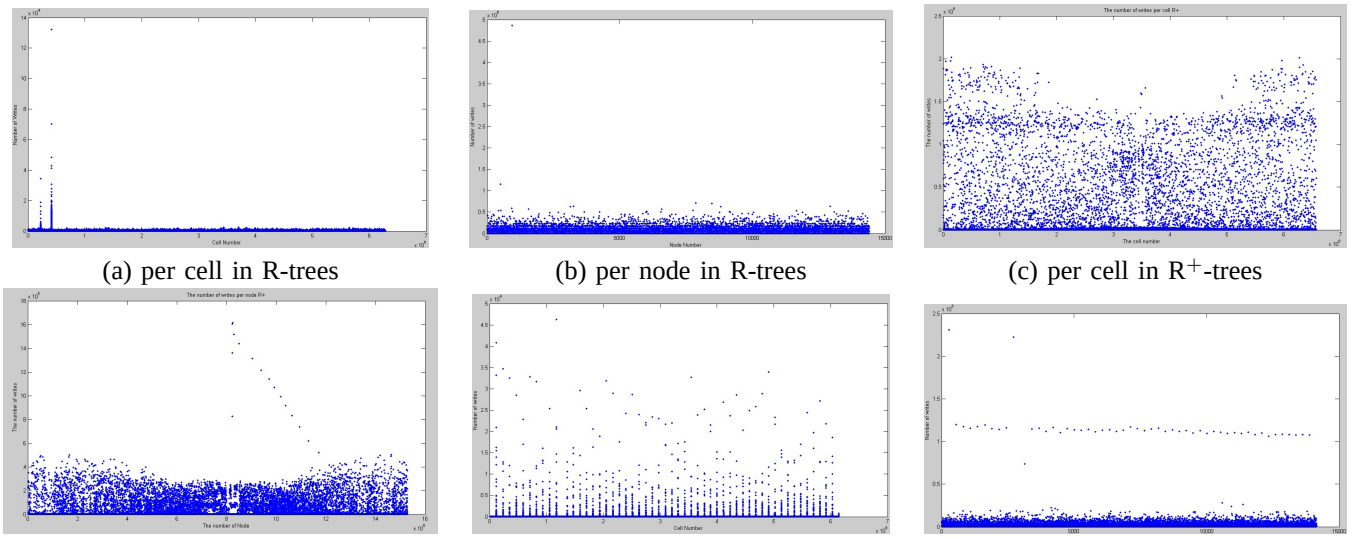
I. Introduction

These days Hard Disk Drives (HDDs) have been quickly replaced by non-volatile memories such as NAND Flash memory because there is no loss of data in such non-volatile memories in case of a blackout. In addition, because the non-volatile memories are electronic devices, the read and write latency times are much faster than those in HDDs. Recently Samsung and Micron Technology have developed a prototype of PCM that is non-volatile byte-addressable memory, 2∼4 times denser than DRAM, and orders of magnitude better than NAND Flash memory in write endurance [1]. Especially, it is known that the existing PCM prototypes are able to conduct 10^6 write operations at maximum for each memory cell. This is ten times better than that in NAND Flash memory. For details, please take a look at Table I indicating parameters – minimum access unit, read and write latency times, and endurance for DRAM, NAND Flash memory, PCM, and HDD. It is obvious that all the parameters of PCM are better than NAND Flash memory and HDD except that the endurance of HDD is non-limited. Moreover, the main drawback of NAND Flash memory is that the minimum access unit is pages, meaning that for updating only one byte we need to rewrite all the page. However, PCM do not have the limitation that each read (or write) operation should be executed on a page (i.e., 2 or 4 KB). In addition, NAND Flash memory has the erase-before-write limitation which means that an erase operation should be performed on a memory area before data is written into that area. This erase operation needs about 2ms and is a major performance bottleneck. However, among advanced non-volatile memories, PCM has no erase operation so data can be written in a certain cell without erase operation. In addition, since PCM is byte-addressable, it can update only a few bytes for a new write operation. Compared to NAND Flash memory, PCM outperforms NAND Flash memory in the aspect of both performance and endurance. Thus PCM is likely to be used rather than NAND Flash memory in near future. However, the main issue of PCM is still write endurance problem. In other words, a memory cell can no longer be used if more than 10^6 write operations are performed in the memory cell. To avoid this endurance problem in PCM, we need to re-design existing algorithms such as B^+-trees or R-trees so that the proposed new algorithms will distribute write operations evenly over memory cells.

Parameteres	DRAM	NAND Flash	PCM	HDD
Volatile	O	X	X	X
Minimum Access Unit	Byte	Page	Byte	Sector
Read Latency	1.25 ns	25 μs	1.25 ns	12.7 ms
Write Latency	1.25 ns	200 μs	15.6 ns	12.7 ms
Endurance	∞	10^5	10^6	∞

TABLE I
Parameters of DRAM, PCM, NAND Flash Memory, and HDD

To reduce the number of write operations in particular memory cells, Chen et al. proposed a B^+-tree indexing algorithm which is used in commercial database management systems such as Oracle and IBM-DB2. In the work, the authors note that the number of write operations can increase in B^+-tree algorithm when a new record is inserted to a node and the records are sorted in the node for fast search. To minimize the number of write operations in B^+-tree algorithm, they consider that leaf nodes are not sorted, while internal nodes are sorted. As a result, their approach might significantly reduce the number of write operations in leaf nodes, while the number of write operations does not decrease in internal nodes. In the similar context, we also need to re-design the existing R-tree algorithm which fits to PCM. This is, the modified R-tree algorithm is required to distribute write operations evenly over the entire PCM cells. In general, R-trees are tree data structures to index spatial objects such as restaurant locations



(a) per cell in R-trees (b) per node in R-trees (c) per cell in R^+-trees

(d) per node in R^+-trees (e) per cell in R*-trees (f) per node in R*-trees

Fig. 1. Number of write operations in inserting 5 million objects. We also observed similar results in the other data sets. Due to space limitation, we will omit the other results.

in order to quickly answer to queries such as "retrieve all Italian restaurants within 1 km of my current location."

In this paper, we focus mainly on R-tree variants in PCM. We apply three different types of existing R-tree variants such as R-trees, R^+-trees, and R*-trees to PCM, and then understand how explicitly R-tree variants cause the write endurance problem of PCM. In addition, we will have a look at which one is heavily affected, compared to the others. To reduce the number of write operations in particular cells, we would like to find which one we have to re-design and important clues to be considered when we modify some R-tree algorithms. To the best of our knowledge, this is the first work that shows how R-tree variants work in PCM.

II. R-tree Variants

R-tree algorithm represents nearby objects with their minimum bounding rectangle (MBR) that describes a leaf or internal node in the tree. The R-trees are balanced search trees so that all leaf nodes are at the same height. Except for the root node, each node is guaranteed to contain a minimum fill. If an object is inserted to a leaf node and then the node is full, the node should be split. This process will be continually made in higher levels of the tree. On the other hand, when an object is deleted in the leaf node, where the number of entries is less than the minimum fill, the leaf node should be merged into another node. To search an object, R-trees make use of the bounding boxes to determine whether or not to search inside a subtree. In this way, most of nodes in the tree are never read during a search. R^+-trees and R*-trees are variants of R-trees. Unlike R-trees, R^+-trees avoid overlapping of internal nodes by inserting an object into multiple leaves. The R*-trees use a combination of a revised node split algorithm and the concept of forced reinsertion at node overflow. This is based on the observation that R-tree structures are highly susceptible to the order in which their entries are inserted, so an insertion-built structure is likely to be sub-optimal [2].

III. Experimental Results and Concluding Remarks

We first implemented R-trees, R^+-trees, and R*-trees, using Java, and run all the experiments on a single computer with 2.4 GHz Intel Core i5 processor and 8GB 1600 MHz DDR3 memory. Then, we randomly generated five data sets including 1 million, 2.5 million, 5 million, 7.5 million, and 10 million objects. In addition, we set 512 to the maximum fill factor. Finally, we insert the objects to each R-trees and variants.

As illustrated in Figure 1, R-trees outperform their variants for PCM by taking in account the number of writes per cell and node, while R^+-trees are similar to R*-trees. The number of the cells being used is quite similar to other R-tree variants. However, the number of nodes that are being used are much higher than other R-tree variants, meaning that many of the nodes are not full. This due to the reason that the R^+-trees do not have minimum fill factor requirement. Furthermore, because of the many split operations, as the R^+-trees do not allow internal nodes to overlap, increasing the number of nodes being used. The figure (x,y) depicts the number of writes per node and per cell in R^+-trees. It is easily seen that some nodes and cell have more number of writes than the others. It is clear that R^+-trees split nodes more often than any other R-tree variants that cause the increase in number of writes per node and cell of certain nodes.

References

[1] S. Chen, P. Gibbons, and S. Nath, "Rethinking database algorithms for Phase Change Memory", In 5th Conference on Innovative Data Systems Research (CIDR), USA, 2011

[2] Y. Manolopoulos, A. Nanopoulos, A. Papadopoulos, and Y. Theodoridis, "R-trees: Theory and applications", In Database Management and Information Retrieval, Springer, 2006