JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 33, 1359-1374 (2017)
DOI: 10.6688/JISE.2017.33.5.15

PCR*-Tree: PCM-Aware R*-Tree

ELKHAN JABAROV[1], MYONG-SOON PARK[1,+], BYUNG-WON ON[2]
AND GYU SANG CHOI[3]
[1]*Department of Computer and Radio Communications Engineering*
Korea University
Seoul, 02841 Korea
E-mail: {ejabarov; myongsp}@korea.ac.kr
[2]*Department of Statistics and Computer Science*
Kunsan National University
Gunsan, 54150 Korea
E-mail: bwon@kunsan.ac.kr
[3]*Department of Information and Communication Engineering*
Yeungnam University
Gyeongbuk, 38541 Korea
E-mail: castchoi@ynu.ac.kr

Phase change memory (PCM) is a byte-addressable type of non-volatile memory. Compared to other volatile and non-volatile memories, PCM is two to four times denser than dynamic random access memory (DRAM). It has better read latency than NAND flash memory. Even though the write endurance of PCM is 10 times better than NAND flash memory, it is still limited to 10^6 times per PCM cell. Decreasing and balancing the number of writes among PCM cells can solve the endurance problem and, where possible, keep PCM cells usable.

Our objective is to design a PCR*-tree – a novel PCM-aware R*-tree that can store spatial data. Initially, we examine how R*-tree causes endurance problems in PCM, and then, we optimize it for PCM. Furthermore, the performance of R*-tree is very poor, especially for insertion, which needs to be solved since it will be used for in-memory databases. According to our experimental results, when the benchmark dataset is used, PCR*-tree dramatically reduced the number of write operations to PCM in average 30 times and also improve the performance in terms of processing time. These results suggest our new method outperforms existing ones for the PCM endurance problem, as well as in its performance.

Keywords: PCM, R*-tree, R-tree, in-memory databases, spatial tree, endurance

1. INTRODUCTION

In-memory databases that consist of dynamic random access memory (DRAM) are used by enterprises for fast decision support. DRAM is a volatile memory; it requires periodic backup to prevent data loss. However, data will be lost if power is lost before the backup. Phase change memory (PCM) is a byte-addressable storage system established by the Samsung Electronics and Micron Technology group. This new storage system is two to four times denser than DRAM and has a better read latency magnitude than NAND flash memory [2, 6]. Even though PCM's endurance is 10 times better than NAND flash memory, it is still lower than DRAM. Despite this, PCM is a secondary

Received May 9, 2016; revised September 19 & October 30, 2016; accepted November 7, 2016.
Communicated by Jianliang Xu.
[+] Corresponding author.

memory preferable to NAND flash memory and a better main memory than DRAM.

In this paper, we propose using PCM for in-memory databases. As previously mentioned, PCM has endurance limitations. We aim to improve this so PCM can be used for in-memory databases. Furthermore, the existing wear-leveling techniques proposed for NAND flash memory [8-10, 12, 14] are not suitable for PCM as NAND flash memory is page addressable, but the PCM memory is byte addressable. Our main goal for this project is to decrease and balance the number of writes to PCM cells. This will solve the endurance problem and keep PCM cells usable as long as possible, so we can use it for in-memory databases.

Being the most used index structure algorithm, the B+ tree algorithm has been altered for PCM [3-5, 13]. Nowadays location information is very important; it is used by smartphones, T-map, Google Earth, and global positioning systems, *etc.* B+ tree cannot handle location information. R-tree, another tree structure, is very similar to B-tree. It can store spatial information like geographic coordinates and multi-dimensional objects. In our studies, we use R*-tree structures for PCM to reduce the number of writes per PCM cell. Of the various kinds of R-trees, we chose R*-tree as it has a more compact structure and more populated nodes, compared to other R-tree variants. Moreover, according to [17], R*-tree outperform the traditional R-tree in query processing and performance. However, these advantages also come with liabilities, such as a higher number of writes and low performance (compared to other R-tree variants), which need to be solved.

The basic idea of the R*-tree is to group nearby objects and represent them with minimum bounding rectangles in the next higher level of the tree. R*-tree reduces coverage and overlap with a modified node split; it reinserts algorithms when nodes overflow. R*-tree requires many writes during insertion and deletion, for keeping the given R*-tree requirements. Our goal is to design a PCM-aware R*-tree that is capable of storing spatial data.

In this paper, we initially evaluated whether R*-tree causes endurance problems, and at the same time, measured performance for insertion, search, and deletion. To check this, we inserted a synthetically developed dataset (consisting of three million elements) into R*-tree. From the figures depicted in Section 3, we saw that R*-tree causes endurance problems. Some records had a huge number of writes, indicating those cells would die much earlier than others. Moreover, performance measurements are presented in performance evaluation section of this paper. Our goal is to decrease the number of writes and spread them equally among all PCM cells in order to keep all cells usable as long as possible, and at the same time, improve performance.

To the best of our knowledge, we are the first researchers to show how R*-tree causes endurance problems in PCM and we believe we are the first to provide a solution to it based on the application level. According to our experimental results, by using both synthetic and benchmark datasets, PCR*-tree deals significantly with this endurance problem by reducing the number of PCM writes; it also improves performance.

2. RELATED WORK

Several attempts have been made in this research area to reduce the number of PCM

write operations by redesigning existing tree structures. A common index structure algorithm, B+ tree is mostly used for DRAM and hard disk drive (HDD) memory, in which case, the number of writes need not be considered. Nevertheless, the number of writes is important when using PCM.

The approaches [3-5, 13, 15] focus on optimizing B+-tree for PCM. As previously stated, our goal is to store spatial objects in memory. One dimensional index structure B+-tree does not work well with spatial data since search space is multidimensional.

R-tree and its variants are commonly used to store spatial objects. Many researchers [8-10, 12, 14] redesigned and applied R-tree to solid state disk (SSD) and flash memory. Wu *et al.* aimed to efficiently handle fine-grained updates caused by R-tree index access to spatial data over flash memory [10]. They proposed using a reservation buffer and a node translation table to reduce the number of unnecessary and frequent updates of information in flash memory storage systems.

Lv *et al.* [8] introduced a tree index structure named log compacted R-tree (LCR-tree). They combined newly arrival logs with the original ones in the same node, which rendered a great decrement in random writes and, at most, one additional read for each node access.

Pawlik and Macyna [9] separated R-tree metadata and aggregated data into different sectors of flash memory.

[16] presents scalable QSF-tree that rely on heuristic optimization to reduce the number of false drops into pages, which does not contain objects satisfying the query.

Jin *et al.* [12] propose to defer the node-splitting operations on R-tree by introducing overflow nodes. In addition, they present a new buffering scheme to efficiently cache the updates to the tree in order to reduce the number writes to flash memory. However creating the buffer does not solve the endurance problem in our case due to the reason that different from NAND Flash memory, PCM is byte addressable.

However, none of the above-mentioned proposed versions of R-tree [8-10, 12, 14, 16] could be directly applied to PCM as they have been proposed for NAND flash memory. NAND flash memory is page addressable, but PCM is byte addressable. Because the structure of PCM is different from the structure of NAND flash memory, the ones designed for NAND flash memory are not suitable for PCM. To the best of our knowledge, we are the first researchers to show how R*-tree cause endurance problems in PCM, and we believe we are the first to provide a solution to it based on the application level.

In [18] a new logging scheme called PCMLogging is presented, which exploits PCM for both data caching and transaction logging to minimize I/O accesses in disk-based databases. PCMLogging enables simplified recovery and prolongs PCM lifetime by integrating log and cached updates. However, in this paper we are focusing on the proposing new R-Tree scheme for PCM without considering logging scheme.

3. PCR*-TREE

As previously mentioned, the maximum number of writes per PCM cell is limited to 10^6 times (then the cell dies). For that reason, we initially need to check whether R*-tree causes the endurance problem. We counted the number of writes per node (and per rec-

ord) for R*-tree while inserting a dataset of 3 million elements. We simulated our experiments with 32, 64, 128, 256, and 512 maximum fill factor values.

3.1 Motivation

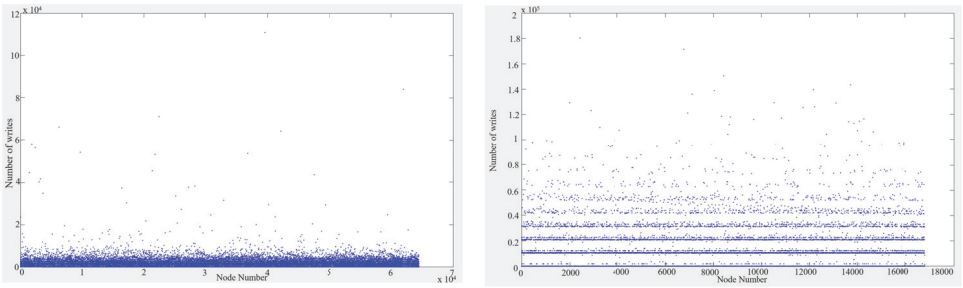
3.1.1 Simulation environment

As it is infeasible to get the real PCM we have simulated the results on DRAM. For all the experiments in this paper the server with the following test-bed configurations have been used: CPU – Intel Xeon E3-1220 v3 3.10GHZ, L2 cache size – 8192KB, Main memory – 8GB, HDD – 1TB, OS – Linux 3.19.0. We used two kinds of dataset for our experimental evaluations: a synthetic dataset consisting of five million objects and a benchmark dataset consisting of 76,999 objects. We have created the synthetic dataset by using the uniform distribution. The benchmark dataset was downloaded from an R-tree portal that is a geographic spatial dataset of Germany in a 2D space[1].

Execution time for insert operation is the total time to insert the given dataset to the tree structure. In case of Search operation, access time is the time spent to find the each data object of a given dataset (500,000 objects for synthetic dataset and 20,000 objects for benchmark dataset) in the leaf nodes. Execution time for deletion is the access time to Search for every data object plus the time spends on its removal as well as merging and updating parent nodes if necessary for the a given dataset (500,000 objects for synthetic dataset and 20,000 objects for benchmark dataset).

3.1.2 Simulation results

Fig. 1 presents the original R*-tree simulation results after inserting synthetically generated three million dataset.



(a) Per node with maximum fill factor 64. (b) Per node with maximum fill factor 256.

Fig. 1. The number of write operations per node when inserting three million objects with 64 and 256 maximum fill factor value into the R*-tree.

According to the results (Fig. 1), the number of writes per PCM node dramatically increases when the R*-tree node splits. It happens for the following reasons.

- The node needs to be sorted in order to find the proper split point.
- R*-tree removes elements from the split point that are somewhere in the middle of the

[1] http://chorochronos.datastories.org/?q=node/54.

node, and furthermore, removes elements one by one. So each time an element is removed during a split, the rest of the elements must come one step to the left, which dramatically increases the number of writes.

- In R*-tree, the split operation degrades performance as the algorithm decides whether to reinsert the element or split it. In any case, it will start to solve the overflow node problem by reinserting elements; however, if the problem is not solved, it then splits the node. It should be noted that the Reinsert operation is an expensive operation for PCM because it consumes many writes and also downgrades the performance.
- All the parent nodes must be updated if there is any change in leaf nodes.

3.2 Proposed Scheme

The detailed algorithms of the proposed scheme are shown in Appendixes A, B and C.

3.2.1 Increased leaf node size

The node needs to be sorted right before reinsert or split operation that deteriorates the performance. As all the elements are being stored in leaf nodes, the split or reinsert will initially execute when the leaf node overflows. If the size of a leaf node will be increased over the size of intermediate nodes, we are then able to postpone executing the OverflowTreatment function that will split or reinsert elements within the node. R*-tree uses the same maximum fill factor value both for intermediate and leaf nodes and accordingly the minimum fill factor value is half of the maximum fill factor value. In case of PCR*-tree, the maximum fill factor value is separated for leaf and intermediate nodes. In initialization step the maximum fill factor value for intermediate nodes, as well as for leaf nodes must set. Figs. 2 (a) and (b), respectively, show examples of the R*-tree and PCR*-tree structures that stores 32 data objects. It is obvious that the proposed structure is more compact, as fewer intermediate nodes are required. In other words, the proposed version will postpone calling the OverflowTreatment function that will decrease the number of writes, and at the same time, improves performance. In performance evolution section we will compare various leaf node size such as increasing the leaf node size two (2X), three (3X), four (4X) and five (5X) times compared to intermediate nodes.

(a) R*-tree max fill factor = 4. (b) PCR*-tree (leaf max ff = 8, non-leaf max fill factor = 4).

Fig. 2. Example of R*-tree and PCR*-tree 2X that stores 32 data elements.

3.2.2 Move once

When the node splits, it is mandatory to find the split-index, indicating which part of the node should be moved to the newly created node. The split-index is somewhere in the middle of the node. R*-tree moves elements starting from the split-index point to the new node one-by-one, which dramatically increases the number of writes. The number of writes rapidly rises because after removing the elements in the middle, the remaining

ones move one step to the left to keep the node consistent. Fig. 3 illustrates the case when datum I is inserted into a node with a maximum fill factor value of 8. Because the node is full, it should split, and the split-index is the fifth element of the node. The number in the top-left corner of each record shows the number of writes, and the sum of those numbers within a node is the number of writes to that node. So, all the data objects starting from datum E should move to the newly created node. When data object E is moved, the remaining ones must move one step to the left. In this small example, the number of writes to that node has already increased four times. Furthermore, data objects F, G, H, and I must also move to the newly created node, which will rapidly increase the number of writes to the split node.

In order to solve this problem, we propose to move all the elements of the node, starting from the split-index point, at once. This approach rapidly decreases the number of writes in the split node, and at the same time, improves the performance.

3.2.3 Split node replacement

Prior to splitting the node, the split algorithm must figure out the point where the node should split. In order to find the split point, the algorithm does some rearranging of the elements within the node, which dramatically increases the number of writes. Our solution to this problem is to write a split node into a new node with the minimum number if writes. Once the node successfully splits, we create a new blank node and then copy the split node where all the rearrangements have been handled in this newly created node. Finally, we replace the split node with the newly created node. A new node is an empty node with a minimum number of writes. Even though this approach increases the number of writes by one time per updated record, it lets us balance the number of writes among all PCM cells, and keeps all PCM cells alive as long as possible.

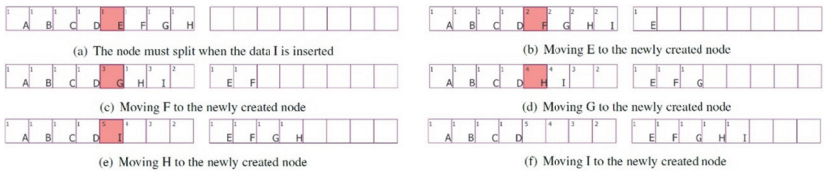


Fig. 3. Example of an original R*-tree split. The number in the top left corner of each record shows the number of writes. The split-index point is the record marked in red.

3.2.4 Single parent update

In R*-tree, if there is any change in leaf nodes, it is necessary to update all the parent nodes of that node up to the root node. It is obvious that each time the node is updated, its endurance goes down. Fig. 4 depicts an example that shows how the parent node is updated in R*-tree. In Fig. 4 (a) the root node must updated after modifications in the leaf nodes. Initially, it writes the MBR of data object A, which is (1,1)(2,2), to its parent, as shown in Fig. 4 (b). Then, the MBR of data object C should be compared with its parent's MBR, and in our case, the parent's MBR should be updated again to (1,1)(3,6), as shown in Fig. 4 (c). Afterwards, the MBR of data object E should be compared with its

parent's MBR, and in our case, the parent's MBR should be updated again to (1,1)(4,6), as shown in Fig. 4 (d). The above process continues till the MBR of the parent node has been compared and updated with all the MBRs of the child node. In this example the root node MBR will be modified two more times while updating parent node with neighboring leaf node. In a given small example, the parent node MBR has been updated five times, as the maximum fill factor value was small. However, if we use a larger maximum fill factor value, number of updates will be huge. Moreover, more writes will be required with the increase of the R*-tree height.

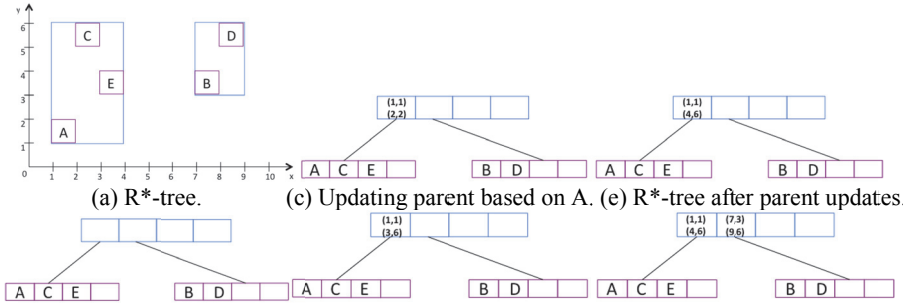


Fig. 4. Example of updating a parent node in the original R*-tree.

We propose to compare the value of the MBRs of child node elements and figure out the MBR value for the parent node. Next, we write the computed value to the parent node. In other words, with the proposed approach, we write to the parent node only one time.

3.2.5 No minimum fill factor requirement after deletion

In order to satisfy the minimum fill factor requirements of R*-tree, after the data have been deleted, if the nodes contains less elements than minimum fill factor value, the elements of that node should merge with other nodes. However it may be very early to merge nodes when they just become less than half full as the merge operation waste the endurance of PCM and degrades the performance. Merge operation wastes the endurance and lowers the performance because the node that does not meet the minimum fill factor requirement should be deleted and the data objects of that deleted node should be re-inserted to tree again. So in the proposed scheme we disable the minimum fill factor requirement for delete operation. In performance evaluation sections we will compare the proposed R*-tree with and without merge in case of the number of writes, as well as performance.

4. SPACE CONSUMPTION

The size of leaf nodes in PCR*-tree is larger than the size of non-leaf nodes, and it is challenging to figure out whether the maximum fill factor value of the tree is based on leaf nodes or non-leaf nodes. This issue is very important for a fair comparison of the results against the original R*-tree. We decided to find out the number of leaf and non-

leaf nodes in PCR*-tree, and we categorize them based on majority. Considering all the maximum fill factor values based on the majority, we will determine whether the maximum fill factor of PCR*-tree is equal to leaf-node size or non-leaf-node size. For this experiment, we refer to the results that we gained by using the benchmark dataset.

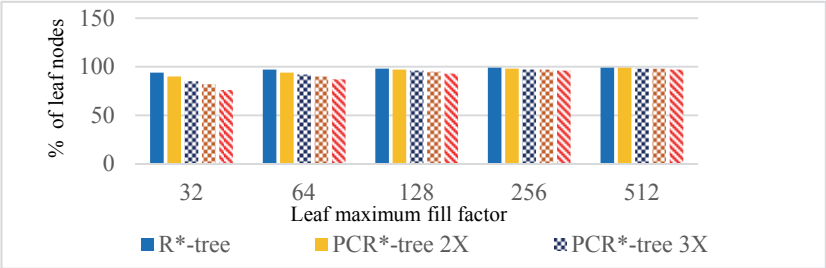


Fig. 5. The percentage of leaf nodes by inserting benchmark dataset.

From Fig. 5, we can see that the number of non-leaf nodes is much lower than the number of leaf nodes, where at most, 24% are non-leaf nodes (PCR*-tree 5X). At least, only 1% are non-leaf nodes (in cases when leaf maximum fill factor = 256/512). As the maximum fill factor value increases, the percentage of leaf nodes radically increases, compared to non-leaf nodes. However, as the leaf-node maximum fill factor increases compared to the non-leaf node size, the number of non-leaf nodes also increases. According to an experiment results, in an average case 94% of the nodes are leaf nodes. Thus, the maximum fill factor value for PCR*-tree is defined based on the number of elements the leaf nodes can store.

However, as the maximum fill factor values for leaf nodes and non-leaf nodes are different, it is essential to figure out the consumed space. Eqs. (4) and (5), respectively, show how the space consumption is calculated for R*-tree and PCR*-tree, where num_{leaf} is the number of leaf nodes, $num_{nonleaf}$ is the number of non-leaf nodes, and ff is the maximum fill factor value. Furthermore, each record takes 32 bytes of memory.

$$SpaceConsumption_{Rstar} = (num_{leaf}+num_{nonleaf})\times ff\times 32 \tag{4}$$

$$SpaceConsumption_{Rstar} = ((num_{leaf}\times ff_{leaf})\times(num_{nonleaf}\times ff_{nonleaf})\times 32 \tag{5}$$

Based on Fig. 6, we can see that PCR*-tree consumes more space in memory when the maximum fill factor value is low. However, its space consumption gets closer to R*-tree as the maximum fill factor value increases. Moreover, the more we increase the leaf node maximum fill factor value the more space is being consumed. Interestingly, PCR*-tree 2X with maximum fill factor value 512 consumes less space than R*-tree. As this experiment have been simulated with benchmark dataset, because of the relatively small amount of the dataset, PCR*-tree 2X consumes less space in memory. Moreover, at worst case, PCR*-tree uses 13% more memory than R*-tree and in average case PCR*-tree consumes 4.5% more space in memory compared to R*-tree. As our main objective is to handle endurance and improve performance, a slight increase in space consumption is not a problem.

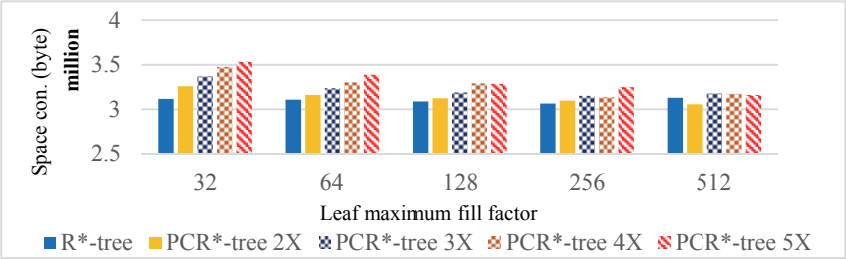


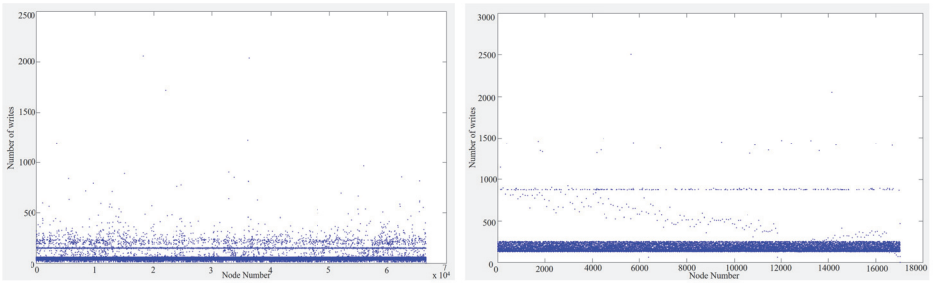
Fig. 6. Space consumed by inserting benchmark dataset.

5. PERFORMANCE EVALUATION

In this section, we compare the proposed PCR*-tree with the original R*-tree by using both a large synthetic and benchmark datasets presented in Section 3.1.1. Comparison is based on the number of writes per node, as well as on performance in terms of insertion, deletion, and search.

5.1 Synthetic Dataset

As previously presented, our synthetic dataset consists of 3 million elements and has been developed by us based on the uniform distribution. To measure search and deletion, we used random 500,000 elements among the inserted ones.



(a) Per node with maximum fill factor 64. (b) Per node with maximum fill factor 256.

Fig. 7. Number of write operations per node while inserting 3 million objects with 64 and 256 maximum fill factor values to PCR*-tree 2X. We also observed similar results using 32, 128 and 512 maximum fill factor values. Due to space limitation, we will omit the other results.

In contrast to Fig. 1, Fig. 7 depicts the number of write operations per node while inserting three million objects with 64 and 256 maximum fill factor values into PCR*-tree 2X. From Fig. 7, we can easily see that the number of writes dramatically decreases, and at the same time, is more proportionally distributed among nodes, which will increase the lifetime of the PCM. Due to similarity of the results and lack of space we omit the results for remaining maximum fill factor values.

Table 1 sums the results from the Figs. 1 and 7 and also shows the results for various leaf node sizes increase (3X, 4X and 5X) by presenting the minimum and maximum

Table 1. Detailed analysis on number of writes per node by using synthetic dataset.

Maximum fill factor		Min. value	Max. value	Average	Stan. Dev.
64	R*-tree	32	110913	987	1741
	PCR*-tree 2X	7	2058	56	43
	PCR*-tree 3X	2	1738	52	35
	PCR*-tree 4X	4	1468	48	27
	PCR*-tree 5X	2	2183	46	33
256	R*-tree	128	180420	11233	15923
	PCR*-tree 2X	3	2507	192	108
	PCR*-tree 3X	6	10889	189	119
	PCR*-tree 4X	9	1842	185	73
	PCR*-tree 5X	14	11631	184	114

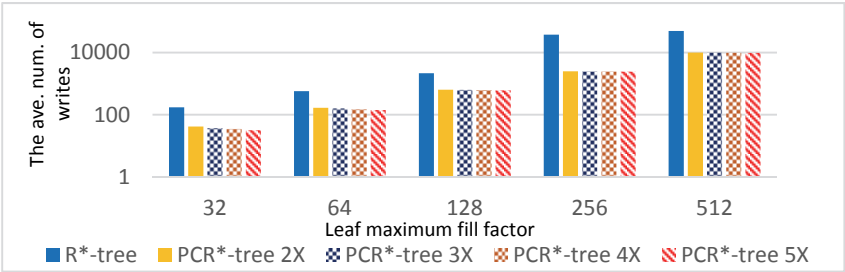


Fig. 8. The average number of writes per node while inserting the synthetic dataset. Logarithmic scale graph (base 10).

number of writes among nodes, average number of writes per node and standard deviation of number of the writes per node. Based on the simulation results, minimum and maximum number or writes of PCR*-tree is much lower than R*-tree. Also, according to Table 1 average number of writes per node is decreased by 17 and 58 times accordingly for 64 and 256 maximum fill factor values by using the PCR*-tree 2X. Furthermore, reducing the number of writes alone cannot completely solve the endurance problem as it is essential to reduce the hot spot on PCM. For checking how the PCR*-tree deals with hot spot on PCM we have calculated the standard deviation on the number of writes on PCM node. The results show that compared to R*-tree, PCR*-tree decreases the standard deviation of the number of writes per node 40 and 147 times accordingly for 64 and 256 maximum fill factor values. Furthermore, for PCR*-tree 2X the standard deviation of the number of writes per node is much lower than the average number of writes per node that shows the number of writes are clustered closely around the average. All this results proves that the PCR*-tree, solves the endurance problem of PCM by decreasing the number of writes as well as spreading the number of writes among all PCM cells equally.

In order to show how the proposed scheme outperforms the original R*-tree, Figs. 8 and 9, respectively, demonstrate the average number of writes per node while inserting the synthetic dataset and deleting 500,000 random elements among the inserted ones for both PCR*-tree and R*-tree. The number of writes per node in average is decreases by 80 times when using the proposed scheme for inserting. Postponing execution of the

OverflowTreatment function by increasing the leaf node size, as well as optimized parent update and splitting mechanisms, helps to keep PCM cells usable as long as possible. The proposed scheme also decreases the number of writes during deletion by not merging the nodes if the minimum fill factor requirement is not met and also by updating the parent nodes only once (as the data are removed). Based on the experimental results, the number of writes per node in average decreased by 7 times for deletes operation.

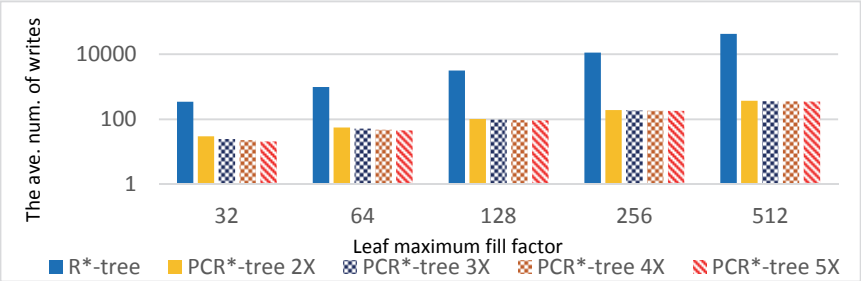


Fig. 9. The average number of writes per node while deleting random 500,000 elements after inserting synthetic dataset. Logarithmic scale graph (base 10).

Based on the Figs. 8 and 9, with the increase of the leaf maximum fill factor value the average number of writes per node increases both for insert and delete operations. However by more increasing the leaf node size compared to intermediate node size, the proposed scheme shows better results.

Table 2. Execution times while using the synthetic dataset for insertion, search, and deletion. All times shown are in seconds.

Maximum fill factor		32	64	128	256	512
Insertion	R*-Tree	157.1	461.7	1869.1	8671.5	33691.2
	PCR*-Tree 2X	70.9	137.5	403.2	1704.9	7842.1
	PCR*-Tree 3X	53.9	89.8	197.0	733.5	3624.8
	PCR*-Tree 4X	47.9	66.9	131.5	393.2	1944.7
	PCR*-Tree 5X	46.5	54.6	95.8	259.7	1145.5
Search	R*-Tree	7032.4	23302.9	26194.5	38251.8	50296.6
	PCR*-Tree 2X	7372.8	17480.6	22153.4	30741.5	47578.3
	PCR*-Tree 3X	2535.6	14995.7	24070.2	30121.5	44610.8
	PCR*-Tree 4X	1786.9	13185.2	21650.7	33457.7	39008.5
	PCR*-Tree 5X	1265.6	8533.7	21508.3	31580.5	36208.1
Deletion	R*-Tree	6519.6	20894.5	21307.5	38519.4	56934.1
	PCR*-Tree 2X	6976.6	16487.5	20614.6	28526.4	43152.1
	PCR*-Tree 3X	2458.7	14241.4	22640.1	28049.9	40783.0
	PCR*-Tree 4X	1705.6	12560.2	20540.9	31179.7	35839.0
	PCR*-Tree 5X	1209.9	8157.9	20431.9	29564.1	33533.5

Since one of goals was to improve performance, Table 2 shows how the PCR*-tree improves performance. The Reinsert function in OverflowTreatment is a very expensive

operation, because it checks all possible combinations to handle the procedure without a split, which decreases performance. Furthermore, if that situation cannot be handled by the Reinsert function, the node must be split, meaning that the time spent by the Reinsert operation was wasted. Postponing the OverflowTreatment function by doubling the leaf node size decreased the time spent on inserting operation. Thus, PCR*-tree improves the Insert performance in average 9 times.

We see a similar pattern in performance improvement for both the Search and Delete operations, as well. It is obvious that the shorter tree height makes it is relatively faster to reach an appropriate leaf node. As the height of PCR*-tree is decreased by doubling the leaf node sizes, it also improves performance when searching and deleting. In average the performance for search operation is decreases by 22%, and for deletion is decreased by 27%. Only PCR*-tree 2X with 32 maximum fill factor value takes slightly more time for search and delete operation. Our analysis shows that in a given case the elements that need to be search or deleted was in an unlucky location (end of the node). Moreover according to the results, the execution time for deleting 500,000 data objects are faster than searching for that 500,000 data objects. The reason behind that is after each deletion, the number of elements in a given tree decreases, making the next search or delete operation faster.

5.2 Benchmark Dataset

In this section, we show the experimental results when the benchmark dataset was used. For search and delete operations, we used a random 20,000 objects from among the inserted ones. The results of the benchmark dataset are quite identical to the results that we have gained from the synthetic dataset.

Figs. 10 and 11, respectively, demonstrate the average number of writes per node for insert and delete operations for both PCR*-tree and R*-tree. For the insert operation, in average the number of writes per node is decreased by 30 times when the benchmark dataset have been used. Similar to the results gained from synthetic dataset, with the increase of the maximum fill factor value, the number of the number of writes increases. Moreover according to the results, with more increase the size of the leaf nodes compared to intermediate nodes, the number of writes slightly decreases.

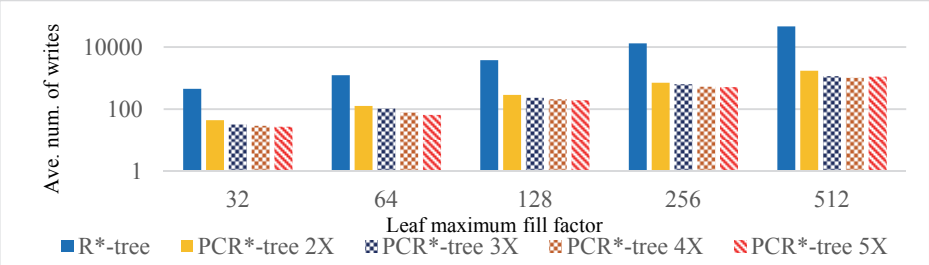


Fig. 10. The average number of writes per node while inserting the benchmark dataset. Logarithmic scale graph (base 10).

According to the results (Fig. 11), PCR*-tree decreases the number of writes in case of deletion as well, due to not merging nodes if the minimum fill factor requirement does not meet and also due to optimized parent node update algorithm. In average case the number of writes have been decreased by 20 times by using PCR*-tree while deletion, when the benchmark dataset is used.

Due to reason that simulation results of the performance (execution time) show similar patterns with the synthetic dataset, we leave this out because of space limitation in this article.

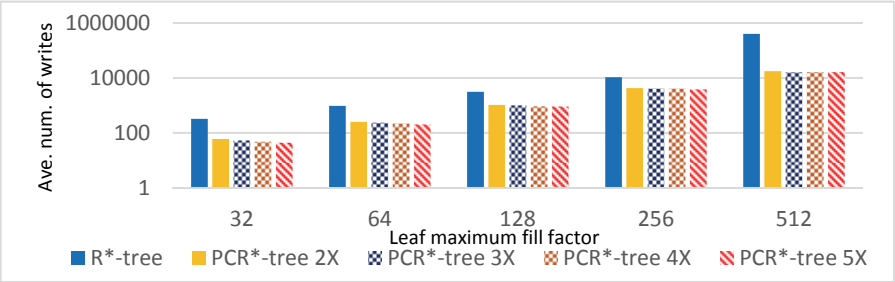


Fig. 11. The average number of writes while deleting 20,000 elements after inserting the benchmark dataset. Logarithmic scale graph (base 10).

We have compared the PCR*-tree scheme with and without merging while deletion (Table 3) in order to check how efficient is not to merge nodes that do not fulfill the minimum fill factor requirement. The benchmark dataset have been used for the experiments. The proposed PCR*-tree with 256 maximum fill factor value have been used for the experiment. According to result no merge version of the PCR*-tree shows much better results in case of average number of writes per node, as well as in standard deviation of number of writes per node. Furthermore, the no merge version of PCR*-Tree improves the delete performance as there is no need to re-insert the element of the node that do not fulfill the minimum fill factor requirement. In average, the proposed no merge version of R -Tree improves the delete performance up to 39%.

Table 3. Comparison of average number of writes per node, standard deviation of writes per node and time spend while deletion for PCR-tree scheme with and without merge operation (maximum fill factor value 256).

Tree	Ave.	Stan. Dev.	Time (sec)
PCR*-tree 2X merge	3764	2082	19.95
PCR*-tree 2X no merge	708	1270	15.32
PCR*-tree 3X merge	3509	2227	13.85
PCR*-tree 3X no merge	638	1226	9.19
PCR*-tree 4X merge	3605	2119	6.80
PCR*-tree 4X no merge	528	1032	5.58
PCR*-tree 5X merge	3273	2154	7.41
PCR*-tree 5X no merge	512	1109	4.14

6. CONCLUSION AND FUTURE WORK

Phase change memory is a byte-addressable type of non-volatile memory. Compared to other volatile and non-volatile memories, PCM is two to four times more dense than DRAM, and it has better read latency than NAND flash memory. Even though the write endurance of PCM is 10 times better than NAND flash memory, it is still limited to 10^6 times per PCM cell. Nowadays, many current applications use spatial data, such as location information; for this reason, storing spatial data in memory is very important. R-tree is a well-known data structure that can handle spatial data; we propose using its variant, R*-tree, over PCM because it is more compact and the nodes are more populated. However, R*-tree performs a lot of writes, and moreover, its performance is poor, especially for insertion. We propose a novel PCM-aware R*-tree algorithm called PCR*-tree. By increasing the leaf node size, moving once while splitting, writing the split node to a blank node, updating parent nodes one time, and disabling the minimum fill factor requirement for delete operation PCR*-tree achieves a dramatic reduction in the number of writes, and at the same time, improves performance. According to our experimental results, when we used a benchmark dataset, the proposed novel scheme in average reduces the number of write operations to PCM node 30 times and also improves performance in terms of processing time. These results suggest our new method outperforms existing ones that address the PCM endurance problem.

The limitation is that PCR*-tree (at the application level of the S/W stack) cannot become aware of the "write count number" for each PCM cell. Therefore, PCR*-tree needs to obtain from the kernel (at the operating system level or the H/W level) the write count number. Then, PCR*-tree will be able to select memory cells with the smallest count number while creating a new node, as well as replace the split node with the node that has smallest number of writes. Our future work will be finding a solution for this problem.

In this paper we propose a PCR*-tree scheme without considering the logging. As a future work we will extend our work by applying the logging scheme.

ACKNOWLEDGMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT, and Future Planning (2013-012524) for the third author (Byung-Won On) and supported by the Ministry of Trade, Industry and Energy (MOTIE, Korea) under Industrial Technology Innovation Program No. 10063130 and by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2016R1A2B4007498) for the fourth author (Gyu Sang Choi).

REFERENCES

1. R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Software Pioneers*, 2002, pp. 245-262.
2. F. Bedeschi, R. Fackenthal, C. Resta, E. M. Donze, M. Jagasivamani, E. Buda, F.

Pellizzer, D. Chow, A. Cabrini, G. M. A. Calvi, and R. Faravelli, "A multi-level-cell bipolar-selected phase-change memory," in *Proceedings of IEEE International Solid-State Circuits Conference-Digest of Technical Papers*, 2008, pp. 428-625,.
3. S. Chen, P. Gibbons, and S. Nath, "Rethinking database algorithms for phase change memory," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, 2011, pp. 21-31.
4. P. Chi, W. C. Lee, and Y. Xie, "Making B+-tree efficient in PCM-based main memory," in *Proceedings of ACM International Symposium on Low Power Electronics and Design*, 2014, pp. 69-74.
5. G. S. Choi, B. W. On, and I. Lee, "PB+-tree: PCM-aware B+-tree," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 27, 2015, pp. 2466-2479.
6. H. Chung, B. H. Jeong, B. Min, Y. Choi, B. H. Cho, J. Shin, J. Kim, J. Sunwoo, J. M. Park, Q. Wang, and Y. J. Lee, "A 58nm 1.8 v 1gb pram with 6.4 mb/s program bw," in *Proceedings of IEEE International Solid-State Circuits Conference*, 2011, pp. 500-502.
7. A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of ACM SIGMOD Iinternational Conference on Management of Data*, Vol. 14, 1984, pp. 47-57.
8. Y. Lv, J. Li, B. Cui, and X. Chen, "Log-compact R-tree: an efficient spatial index for SSD," in *Proceedings of International Conference on Database Systems for Advanced Applications*, 2011, pp. 202-213.
9. M. Pawlik and W. Macyna, "Implementation of the aggregated R-tree over flash memory," in *Proceedings of International Conference on Database Systems for Advanced Applications*, 2012, pp. 65-72.
10. C. H. Wu, L. P. Chang, and T. W. Kuo, "An efficient R-tree implementation over flash-memory storage systems," in *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems*, 2003, pp. 17-24.
11. N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *Proceedings of ACM SIGMOD Record*, Vol. 19, 1990, pp. 322-331.
12. P. Jin, X. Xie, N. Wang, and L. Yue, "Optimizing R-tree for flash memory," *Expert Systems with Applications*, Vol. 42, 2015, pp. 4676-4686.
13. L. Li, P. Jin, C. Yang, Z. Wu, and L. Yue, "Optimizing B+-tree for PCM-based hybrid memory," in *Proceeding of International Conference on Extending Database Technology*, 2016, pp. 662-663.
14. M. Sarwat, M. F. Mokbel, X. Zhou, and S. Nath, "Generic and efficient framework for search trees on flash memory storage systems," *GeoInformatica*, Vol. 17, 2013, pp. 417-448.
15. E. Tousidou, M. Vassilakopoulos, and Y. Manolopoulos, "Performance evaluation of parallel S-trees," *Journal of Database Management*, Vol. 11, 2000, p. 28.
16. R. Orlandic and B. Yu, "Scalable QSF-trees: retrieving regional objects in high-dimensional spaces," *Journal of Database Management*, Vol. 15, 2004, pp. 45-59.
17. P. Patel and D. Garg, "Comparison of advance tree data structures," 2012. arXiv preprint arXiv:1209.6495.
18. S. Gao, J. Xu, T. Härder, B. He, B. Choi, and H. Hu, "PCMLogging: Optimizing transaction logging and recovery performance with PCM," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 27, 2015, pp. 3332-3346.

Elkhan Jabarov received MSc degrees in both Computer Science and Engineering from Korea University, Korea. He is currently pursuing a Ph.D. in the Department of Computer and Radio Communications Engineering, Korea University, Korea. His research interests include databases and data mining.

Myong-Soon Park is Professor of Computer Science department at Korea University in Seoul, South Korea. He received his BSc in Electronics Engineering from Seoul National University, an MSc in Electrical Engineering from the University of Utah in 1982, and a Ph.D. in Electrical and Computer Engineering from the University of Iowa in 1985. He was an assistant professor at Marquette University from 1985 to 1987.1 and at Postech from 1987.2 to 1988.2. Since 1988.3 he has been an Assistant, Associate and Full Professor at Korea University until now. Professor Park was the chair of the SIG on parallel processing of KIISE (1997-2000) and has been on program committees for various international conferences. His research interests include sensor networks, internet computing, parallel and distributed systems, and mobile computing.

Byung-Won On earned his Ph.D. degree in Department of Computer Science and Engineering, Pennsylvania State University at University Park, PA, USA in 2007. Then, he worked as a full-time researcher in University of British Columbia, Advanced Digital Sciences Center, and Advanced Institutes of Convergence Technology for almost seven years. Since 2014, he has been a faculty member in Department of Statistics and Computer Science, Kunsan National University, Gunsan-si, Jeollabuk-do, Korea. His recent research interests are around Data Mining and Databases, mainly working on AI-based text mining and big data management technologies.

Gyu Sang Choi received his Ph.D. in Computer Science and Engineering from Pennsylvania State University. He was a research staff member at the Samsung Advanced Institute of Technology (SAIT) for Samsung Electronics from 2006 to 2009. Since 2009, he has been with Yeungnam University, where he is currently an Associate Professor. He is now working on non-volatile memory and storage systems, whereas his earlier research mainly focused on improving the performance of clusters. He is a member of ACM and IEEE.