An Index Structure for Main-memory Storage Systems using The Level Pre-fetching

Seok Jae Lee
Department of Computer and Communication Engineering,
Chungbuk National University, Cheongju, Korea

Jong Hyun Yoon
Department of Computer and Communication Engineering,
Chungbuk National University, Cheongju, Korea

Seok Il Song
Department of Computer Engineering,
Chungju National University, Chungju, Korea

Jae Soo Yoo
Department of Computer and Communication Engineering,
Chungbuk National University, Cheongju, Korea

ABSTRACT

Recently, several main-memory index structures have been proposed to reduce the impact of secondary cache misses. In main-memory storage systems, secondary cache misses have a substantial effect on the performance of index structures. However, recent studies still suffer from secondary cache misses when visiting each level of index tree. In this paper, we propose a new index structure that minimizes the total amount of cache miss latency. The proposed index structure prefetched grandchildren of a current node. The basic structure of the proposed index structure is based on that of the CSB+-Tree, which uses the concept of a node group to increase fan-out. However, the insert algorithm of the proposed index structure significantly reduces the cost of a split. The superiority of our algorithm is shown through performance evaluation.

Keywords: *Main-memory storage system, Main-memory index structure, Cache conscious, Pre-fetch.*

1. INTRODUCTION

As the cost of main memory in server computer systems becomes cheaper, MMSS (Main Memory Storage Systems) prevails over other types of memory for most application areas. It is a well-known fact that MMSS provides an order-of-magnitude performance improvement over disk based storage systems. The major bottleneck of traditional disk storage systems was disk I/O and much research has proceeded to hide the disk I/O that occupies the fraction of total execution time of transactions. As a result, the impact of disk I/O on the performance of storage systems is significantly reduced.

In a similar fashion, recently, as the performance gap between CPU and main memory increases, it becomes increasingly important to consider cache behavior and to reduce L2 cache line misses for higher performance, especially in MMSS[1][2]. Subsequently, several researchers in the storage system community have improved the performance of main memory index structures by reducing L2 cache misses [3-5]. In the end of the 1990's, Rao and Ross proposed two index structures that improved the cache performance of index searches for main-memory databases[4][5]: The Cache-Sensitive Search Tree (CSS-Tree) [4] and Cache-Sensitive B+-Tree (CSB+-Tree) [5]. These structures are the founding cache conscious index structures. Subsequently, [3] was proposed as a cache conscious index structure.

* *Corresponding author. E-mail : yjs@chungbuk.ac.kr*
Manuscript received Feb. 28, 2007 ; accepted Mar. 12, 2007

This work was supported by the Regional Research Centers Program of the Ministry of Education & Human Resources Development in Korea and the Program for the Training of Graduate Students in Regional Innovation which was conducted by the Ministry of Commerce, Industry and Energy of the Korean Government.

All of the existing index structures described above still suffer processor's L2 (level 2) cache miss stall at each level of index trees. Our approach is to prefetch grandchildren nodes of a visiting node in parallel. We modify the structure of CSB+-Tree to apply the prefetch technique. Our proposed index structure does not need additional space to prefetch grandchildren. Also, to reduce the update cost of CSB+-Tree, a new algorithm is presented.

This paper is organized as follows. In section 2, we describe existing cache conscious index structures. In section 3, we describe in detail the characteristics of our proposed index structure and algorithms such as insert and search algorithm. In section 4, we analyze the performance of the proposed index structure mathematically. The superiority of our algorithm is shown through performance evaluation. Finally, in section 5, we conclude this paper.

2. RELATED WORKS

2.1 Previous Works

CSS-Tree designed for the OLAP environment is a very compact and space-efficient B+-Tree. It eliminates pointers of index trees completely, and stores keys in contiguous space. Children of a node can be easily calculated by exploiting the k-ary method. The CSB+-Tree applies the idea of the CSS-Tree to an index structure for the OLTP environment that supports efficient update. The CSB+-Tree stores groups of sibling nodes in contiguous memory, and reduces the number of pointers stored in the parent node.

The prefetch techniques were applied to the B+-Tree to eliminate the cache miss latency efficiently [3]. Even though [4][5] made efforts to reduce cache misses, they still suffer from full cache miss latency when traversing to the next level during search, and when trying to read the next leaf node during scan. The modern CPU prefetch technique was introduced to solve these problems [3]. To reduce the number of cache misses incurred during search, a node size is widened by multiples of cache lines. This reduces the height of the index structure. To reduce the latency of cache misses incurred during scan, it introduces jump-pointer techniques. The wider nodes come almost for free through prefetching, and the jump-pointer allows the scan to prefetch arbitrarily far ahead leaf nodes so as to hide the normally expensive cache misses associated with traversing leaves within the range.

2.2 Motivation and Our Approach

All existing index structures described above still suffer cache miss stall at each level of index trees. Even though [3] uses a prefetching technique, cache miss stall occupies more than 50% of its search time. Reference [3] recognizes the facts and presents some clues for solutions that prefetch children or even grandchildren in parallel. [3] also mentions problems of the method: (i) the data dependence through the child pointer; (ii) the relatively large fan-out of the tree nodes; and (iii) the fact that it is equally likely that any child will be visited (assuming uniformly distributed random search keys).

The most difficult problem is that it is almost impossible to forecast descendant nodes to be visited. A possible approach is to prefetch whole children or grandchildren of a child node to be visited in the next level. However, to prefetch the descendants we must know their pointers before they are accessed. Even if it is possible to prefetch descendant nodes, usually the number of descendant nodes is too large to prefetch simultaneously.

Our approach is, also, to prefetch grandchildren nodes of a visiting node in parallel. We modify the structure of the CSB+-Tree to apply the prefetch technique. Our proposed index structure does not need additional space to prefetch grandchildren. To reduce the update cost of the CSB+-Tree, a new insert algorithm is presented.

To assist explanation of our index structure, we describe the structure of the CSB+-Tree briefly. The CSB+-Tree is a balanced multi-way search tree. Every node in the CSB+-Tree of order d contains m keys, where $d \leq m \leq 2d$. The CSB+-Tree puts all child nodes of any given node into a node group. Nodes within a node group are stored contiguously and can be accessed using an offset from the first node in the group. Since the CSB+-Tree node only needs to store one child pointer explicitly; it can store more keys per node than the B+-Tree. For example, if the node size (and cache line size) is 64 bytes and each key and child pointer occupies 4 bytes, then B+-Tree can only hold 7 keys per node whereas the CSB+-Tree can hold 14 keys per node. Our index inherits most of these properties.

3. THE PROPOSED INDEX STRUCTURE

3.1 Overview

As we mentioned in the previous section, our index tree is based on the CSB+-Tree. Fig. 1 shows the index structure of the lpCSB+-Tree (level prefetching CSB+-Tree). The child nodes of an internal node are stored in physically contiguous space. Let h be the height of the index tree and 0 be the level of the root node. In our index tree, the entries of internal nodes on the level less than h-2 have pointers for the node groups that contain grandchildren. The pointers are used to prefetch the grandchildren of a node that is being visited once the child node to be visited is decided. This means that except for the root node, we do not read a node but a node group.

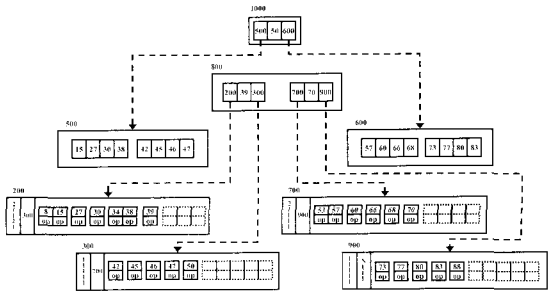


Fig. 1. The proposed index structure

In the CSB+-Tree, the node size is fixed as multiples of cache line size. However, we do not restrict the size of a leaf node to a fixed size, since we always read the entire node group. In addition, the CSB+-Tree creates a new node group when the number of leaf nodes in an old node group exceeds the fixed number, without concerning the space utilization of the old node group. On the contrary, our index tree creates a new node group only when the space of the node group is consumed by leaf nodes.

3.2 Structures of Nodes and Node Groups

The structures of nodes and node groups of our index tree are different from the CSB+-Tree. Internal nodes have different structure according to their level. The internal nodes whose level is less than h-2, consists of pointers for grandchildren groups of the current node and separators for children nodes.

Internal nodes on level h-2 do not have pointers for grandchildren groups, since the leaf level is h-1. Actually, in the node of the CSB+-Tree, there is a pointer for the node group that contains children of the node. On the other hand, internal nodes in our index tree are not required to maintain the pointers, because nodes are prefetches when traversers visit their grandparent node. Therefore, we require the additional space for the pointers of grandchildren groups.

As described in Sect. 3.1, in order to reduce the split cost of a node group, we do not restrict the size of a leaf node to multiples of cache line size When a new node is created, the fixed size of memory for the node is not allocated. Instead, only the space that is able to accommodate actual number of entries is allocated from the reserved space of the node group.

Then, whenever a new entry is inserted into the node, space is allocated for the new entry. This means that a node may occupy the whole space of a node group. Due to this property, node groups for leaf nodes need additional information that indicates the size of each leaf node in node groups. The space overhead of the additional information can be ignored since the number of nodes in a node group is small enough to represent with 4~8 bits.

3.3 Insert and Search Algorithms

A. Insert Algorithm

Our insert algorithm is carried out in two stages. In the first stage, we traverse the tree to find a leaf node for the newly inserted entry. First, we access the root node. Before accessing the root node, we prefetch the children of the root in parallel. We assume that the pointers for the root node and the node group that contains the children of the root are stored in the index descriptor. In the root node, once an inserter decides on a child node to be visited, the children of the child node are prefetched in parallel through a pointer stored in the root node, and then the inserter visits the decided child node. The inserter repeats the above procedure in the same manner on the next level until finding a leaf node for the newly inserted entry.

In the second stage, we insert the new entry into the found leaf node. If overflow occurs, we treat this situation as follows. Our split algorithm on the leaf level reduces the update cost of the CSB+-Tree. Leaf nodes of our index tree are not restricted to a fixed size. In Fig. 2, a node group is stored in the address 500 in memory, and the maximum number of leaf nodes is 5. If we restrict the number of entries in a leaf node to 2, when the new entry 53 is inserted into d, overflow occurs. Subsequently, d is split, and finally the node group is split since the maximum number of nodes is 5.

However, as in Fig. 2, our split algorithm does not split d. Instead, we allocate space for the new entry from the node group, and extend the space of d to accommodate the new entry. Subsequently, the node group is not split. Node groups in our index tree are split only when their reserved space is consumed completely. The split of internal nodes are similar to that of the CSB+-Tree [5].

B. Search Algorithm

The search algorithm is very similar to the traverse algorithm of insert operations. Before accessing the root node, we prefetch the children node group of the root node. In addition, once a child node is decided, we prefetch the children of the child node. The process is repeated on the next node. We can easily adapt the scan algorithm of [3] for the search algorithm to our index tree.

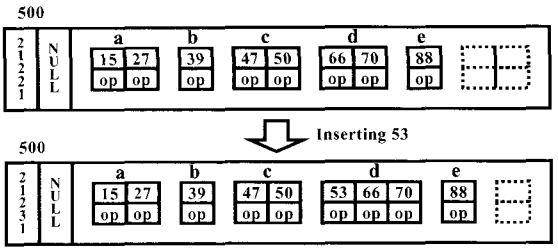


Fig. 2. Inserting a new entry into a leaf node group

4. PERFORMANCE EVALUATION

In this section, the search performances of our index structure and the CSB+-Tree mathematically are analyzed in terms of the latency caused by cache misses. We can present the total number of cache line misses (T_{total}) during a search operation, where the height of the index tree is h, as the following equations:

$$T_{total} = T_1 + \sum_{l=1}^{h-1} \{\alpha T_1 + wT_p(tn - \alpha - pn)\}$$

$$where, \begin{cases} \alpha = \begin{cases} pn = 0, & 1 \\ pn > 0, & 0 \end{cases} \\ T_1 = T_c + \{T_p(w-1)\} \end{cases} \quad (1)$$

T_1 is the average latency to load a node consisting of w cache lines. It is composed of cache miss latency (T_c) to load the first cache line and the latency of prefetching the following cache lines that consists of the node. T_p is the time taken to prefetch a cache line. pn is the number of prefetched nodes in a node group. If pn is greater than 0, the first node of the node group is already prefetched at least. tn

is the position of a target child node to be visited in the node group.

If the *tn* is equal to *pn*, there is no cache miss and no prefetching latency. If the *tn* is greater than 0 and less than *pn*, there is only prefetching latency. *pn* impacts on the performance of index tree significantly. If we are able to assure that *pn* is greater than 0, we obtain considerable performance gain.

The T_{total} of the pCSB+-Tree(prefetching CSB+-Tree) is $T_{total} = \sum_{l=0}^{h-1} T_l$ since it suffers from cache misses whenever accessing each level.

The search performance of our index tree is definitely dependent on *pn* and *h*. If *pn* is 0, we suffer from T_l delay whenever accessing a node in the next level. *pn* depends on the processing time in a node. *h* of our index tree may be higher than that of the pCSB+-Tree because we cannot extend the node size as the pCSB+-Tree due to the number of grandchildren to prefetch.

We performed simulation on a 1.7Ghz Pentium 4 CPU with 512Mbytes main memory. The OS was Redhat Linux 7.1 and compiler was gcc 2.96. Table 1 shows the parameters used for simulation.

Table 1. Parameters for performance evaluation

Parameters	Values
total size of L2 cache	256 KByte
size of a L2 cache line	64 Byte
full cache miss latency	88.2 ns
prefetching time	5.5 ns
the time of binary search in a cache line	54.2 ns
data set	500,000 entries
node size	1 - 8 cache lines

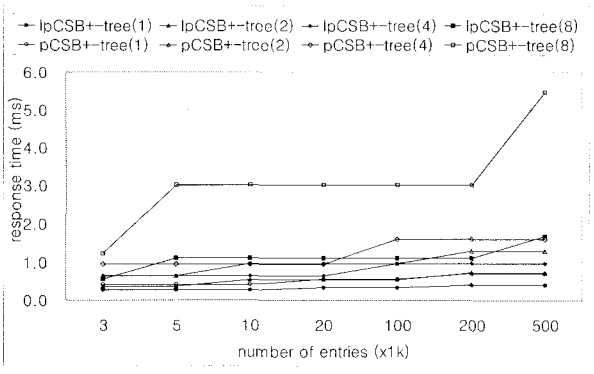


Fig. 3. Response time of search transaction (node size: n cache line size)

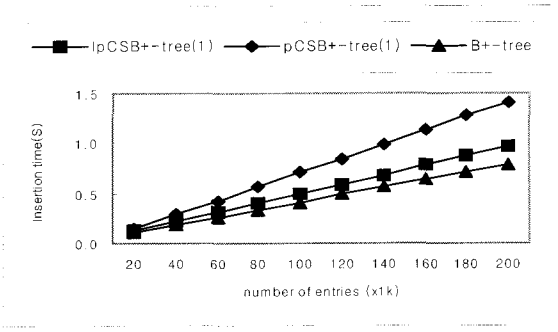


Fig. 4. Insertion time of lpCSB+ tree, pCSB+ tree and B+ tree (node size: 1 cache line size)

We measure the response time of searches and compare these with the pCSB+-Tree. As shown in Fig. 3, the response time of search of the lpCSB+-Tree is faster than that of the pCSB+-Tree in all cases. The reason that we fix the node size as the n cache lines is to reduce the cache miss latency time of the prefetched child nodes. As mentioned earlier, optimum performance is achieved when the node size is one cache line. Fig. 4 shows the insertion time of lpCSB+ tree, pCSB+ tree and B+ tree when the node size is fixed at one cache line.

5. CONCLUSIONS

This paper proposed a new index structure to minimize the total amount of cache miss latency of insert and search operations. The proposed index structure prefetches grandchildren once the searcher or inserter visits an internal node through pointers for node groups of grandchildren nodes. We modify the structure of the CSB+-Tree to apply the proposed prefetching techniques. To reduce the insert and update cost of the CSB+-Tree, the new management method of leaf node groups has been proposed. We presented a few factors that have impact on search performance through mathematical analysis. The performance evaluation shows that our index structure improves search and insertion performance.

REFERENCES

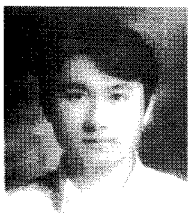
[1] Stefan Manegold, Peter A. Boncz, Martin L. Kersten., "Optimizing database architecture for the new bottleneck: memory access," **In VLDB Journal**, Vol.9, No.3, 2000, pp. 231-246.

[2] Peter A. Boncz, Stefan Manegold, Martin L. Kersten., "Database Architecture Optimized for the New Bottleneck: Memory Access," **In Proceedings of VLDB Conference**, 1999, pp. 54-65

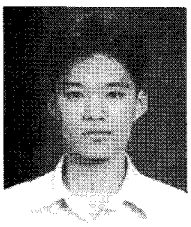
[3] Shimin Chen, Phillip B. Gibbons, Todd C. Mowry., "Improving Index Performance through Prefetching," **In Proceedings of ACM SIGMOD Conference**, 2001, pp. 235-246.

[4] Jun Rao, Kenneth A. Ross., " Making B+-Trees Cache Conscious in Main Memory," **In Proceedings of ACM SIGMOD Conference**, 2000, pp. 475-486.

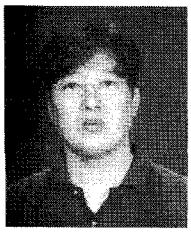
[5] Jun Rao, Kenneth A. Ross., "Cache Conscious Indexing for Decision-Support in Main Memory," **In Proceedings of VLDB Conference**, 1999, pp. 78-89..



Seok Jae Lee
He received the B.S., M.S. and Ph.D degrees in Computer and Communication Engineering in 2000, 2002 and 2006 from Chungbuk National University, Cheongju, South Korea. He is now a Post Doc. in Chungbuk National University. His main research interests are the database system, main memory storage system, cluster system and real-time distributed computing.



Jong Hyun Yoon
He received the B.S. and M.S. degrees in Computer and Communication Engineering in 2003 and 2005 from Chungbuk National University, Cheongju, South Korea. He is now a doctoral course in Chungbuk National University. His main research interests are the database system, main memory storage system, object based cluster storage system and cluster backup system.



Seok Il Song
He received the B.S., M.S. and Ph.D degrees in Computer and Communication Engineering in 1998, 2000 and 2003 from Chungbuk National University, Cheongju, South Korea. He is now a professor in the department of Computer Engineering, Chungju National University, Chungju, South Korea. His main research interests are the database system, index structure, distributed computing and storage management system.

Jae Soo Yoo
He received the B.S. degree in Computer Engineering in 1989 from Chunbuk National University, Chunju, South Korea. And he received the M.S. and Ph.D. degrees in Computer Science in 1991 and 1995 from Korea Advanced Institute of Science and Technology, Taejeon, South Korea. He is now a professor in the department of Computer and Communication Engineering, Chungbuk National University, Cheongju, South Korea. His main research interests are the database system, multimedia database, distributed computing and storage management system.