Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems

Sang K. Cha Sangyong Hwang Kihong Kim Keunjoo Kwon

School of Electrical Engineering and Computer Science
Seoul National University
{chask, syhwang, next, icdi}@kdb.snu.ac.kr

Abstract

Recent research addressed the importance of optimizing L2 cache utilization in the design of main memory indexes and proposed the so-called cache-conscious indexes such as the CSB+-tree. However, none of these indexes took account of concurrency control, which is crucial for running the real-world main memory database applications involving index updates and taking advantage of the off-the-shelf multiprocessor systems for scaling up the performance of such applications. Observing that latching index nodes for concurrency control (CC) incurs the so-called coherence cache misses on shared-memory multiprocessors thus limiting the scalability of the index performance, this paper presents an optimistic, latch-free index traversal (OLFIT) CC scheme based on a pair of consistent node read and update primitives. An experiment with various index CC implementations for the B+-tree and CSB+-tree shows that the proposed scheme shows the superior scalability on the multiprocessor system as well as the performance comparable to that of the sequential execution without CC on the uniprocessor system.

1. Introduction

With the price of server DRAM modules continues to drop, the main memory DBMS (MMDBMS) emerges as an economically viable alternative to the disk-resident DBMS (DRDBMS) in many problem domains. MMDBMS can show orders-of-magnitude higher performance than DRDBMS not only for read transactions but also for update transactions. However, such a significant performance gain of MMDBMS over DRDBMS does not come automatically by just placing the database in memory but requires MMDBMS-specific optimization techniques. For example, the so-called differential logging scheme improves the update and recovery performance of MMDBMS significantly by enabling fully parallel access to multiple log and backup partition disks [LKC01]. Such a degree of parallelism was not achieved with the conventional, DRDBMS-oriented logging and recovery schemes such as ARIES [MH+92].

For the efficient processing of read-intensive transactions, recognizing the ever-widening speed gap between CPU and memory, recent research addressed the problem of optimizing L2 cache utilization in the design of main memory indexes and query processing. It has been shown that the T-tree, proposed in the 80's [LC86], shows poor L2 cache utilization with today's computer architecture [RR99]. The so-called cache-conscious index structures such as the CSB+-tree ([RR00]) reduce the cache misses and thereby improve the search performance. The CSB+-tree keeps only one child pointer of the B+-tree per node, almost doubling the fanout of the tree. Recognizing the pointer elimination is not effective for the R-tree whose MBR key is much bigger than a pointer, the CR-tree focuses on efficiently compressing the MBR keys to pack more entries per node [KCK01]. The pkT-tree and pkB-tree significantly reduce cache misses by storing partial key information in the index in the face of nontrivial key sizes [BMR01].

While all of these cache-conscious indexes effectively improve the search performance by increasing the index fanout and reducing the so-called cold and capacity cache misses, they were studied without much consideration of concurrency control (CC), which is crucial for running the real-world main memory database applications involving index updates and taking advantage of the off-the-shelf multiprocessor platforms for scaling up the performance

Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001

of such applications. One naïve approach to this problem is to use the conventional, disk-resident index CC schemes such as lock coupling [BS77], which involves latching nodes during the index traversal. However, this naïve approach leads to the poor scalability because latching involves memory write and thus incurs the so-called coherence cache misses on the shared-memory multiprocessor systems [CSG99].

The so-called physical versioning was proposed for the T-tree to be used in the Dalì main memory storage system [RS+97]. Its key idea is to create a new version of the node for the updater so that the index readers do not interfere with the updaters. The major advantage of this scheme is the latch-free traversal of indexes. As a result, a high degree of concurrency comparable to that of no concurrency control can be achieved for read transactions. Incorporation of the updated version into the index involves obtaining latches either at the tree-level or both on the tree and on the node to update. The major problem with this scheme is the high cost of creating versions. The index performance degrades sharply with the increasing update ratio. The scalability of update performance is also very poor, even on the dual processor platform where the reported experiment was conducted.

To the best of our knowledge, this paper is the first that addresses the importance of minimizing memory writes in the CC of main memory indexes to achieve the multiprocessor scalability of index search and update performance. After discussing the coherence cache miss overhead associated with latching index nodes, we propose a new CC scheme that supports the latch-free index traversal for B+-tree variants. Called an optimistic, latch-free index traversal (OLFIT) CC, for each node, this scheme maintains a latch, a version number, and a link to the next neighbor node at the same level. The next node link is borrowed from the B^{link}-Tree ([LY81]) to facilitate the split handling. The index traversal involves consistent node reads starting from the root. Here, the consistency means that no update occurs between the start and the end of a node read. To ensure the consistency of node reads without latching, every node update first obtains the node latch, updates the node content, increments the version number, and releases the latch. The node read begins with reading the version number into a register and ends with verifying if the node latch is free and the current version number is equal to the register-stored one. If these two conditions are true, the read is consistent. Otherwise, the node read is retried until the conditions become true.

To verify the superior scalability of the OLFIT scheme, we implemented the OLFIT and a few representative index CC algorithms for the B+-tree and CSB+-tree: lock coupling, tree-level latching, physical versioning with the node-level latch, and physical versioning with the tree-level latch. We chose not only the CSB+-tree but also the B+-tree because the latter shows reasonably good cache performance while not losing good update performance. The result of our experimental study on an eight-CPU shared-memory multiprocessor system shows that for the pure read workload, the OLFIT and the physical versioning schemes show almost the same scalable performance as that of no concurrency control. However, the OLFIT is the only scheme that shows almost linear scalability of update performance with the increasing number of processors.

This paper is organized as follows. Section 2 presents the background of our study, and section 3 presents the basic idea of this paper and formulates our problem. Section 4 and 5 describe the node and tree operations of OLFIT, respectively. Section 6 presents the result of the experiment conducted to compare OLFIT with other representative index CC algorithms. Section 7 concludes this paper.

2. Background

2.1 Memory Hierarchy and Cache Misses

Today's computer systems use fast cache memory to fill the speed gap between CPU and main memory. For example, on the SUN Enterprise 5500 with 400MHz UltraSparc II CPU and EDO DRAM, L2 cache hit takes about 8 processor clocks, while L2 cache miss takes about 100 processor clocks [Sun97].

For the shared-memory multiprocessor system, which is the most common form of parallel computing environment, there is an additional type of overhead associated with synchronizing multiple caches and memory [CSG99]. The so-called coherence cache misses occur when one processor updates a specific cache block and the copies of that block that happen to be cached in other processors are invalidated. The L1 and L2 cache misses occur when these other processors attempt to access the updated block later. The overhead associated with the cache coherence is severe if multiple processors repeatedly update the same data. Note that the coherence cache misses do not occur for read-only applications where each processor simply reads data.

2.2 Cache-Conscious Index Structures

The T-tree, a balanced binary tree with many elements in a node, was proposed in the mid-80's when the speed gap between CPU and memory was not significant. Its traversal pattern of comparing the search key with the two end keys of a node (or only one key in the enhanced version) leads to poor L2 cache behavior on today's computer systems [RR99]. Recognizing the reasonably good cache performance of the B+-tree, Rao and Ross proposed a variant called CSB+-tree that further enhances the cache utilization by eliminating most of child pointers [RR00]. This pointer elimination, given the node size in the order of the cache block size, almost doubles the fanout and thus leads to significant reduction in the tree height and the cache misses during the index traversal.

However, because the CSB+-tree stores child nodes in the contiguous memory called the node group, it increases

the update cost. The so-called, full CSB+-tree avoids this problem by preallocating the memory for the full node group and handling the node split by shifting nodes in the full node group.

2.3 Concurrency Control of Index

The lock coupling involves latching index nodes heavily [BS77]. The index traversal proceeds from one node to its child by holding the latch on the node while requesting the latch on a child. The requested latch is a shared or an exclusive mode, depending on the action to be performed on the target node. The latch on the node is released after the child is latched. One variant of lock coupling is the optimistic descent algorithm. Upon insertion, the algorithm first traverses to a leaf with the optimistic assumption that the target leaf is a safe node. If the assumption is not true, it tries again without the optimistic assumption.

The B^{link}-Tree removes the need for lock coupling by linking each node to its right neighbor [LY81]. If a node is split by an insertion, the split node is linked with its neighbors. In the original B+-tree without links, the index traversal from one node to its child should hold the latch on the child before releasing the latch on the node to prevent the split of the child. However, in the B^{link}-Tree, the index traversal can release the latch on the node before holding the latch on the child because it can reach the split node from the child through the links.

The tree-level locking is a naïve approach that latches the whole tree in the shared mode for search and in the exclusive mode for update. This approach leads to severe degradation of concurrency on multiprocessors.

The physical versioning is a scheme that traverses the index nodes without latching through versioning [RS+97]. However, the index update involves significant overhead of memory allocation and write. For every update, updaters allocate a new version of a node, copy the content of the old version to the new one, and update the new one. It then atomically swaps the old version and the new one. When node latches are used, the node to update and its parent are latched. A garbage collector deallocates the old version of the node when there are no readers that are reading it. We consider two variants of the physical versioning: one that places latches on nodes, and another with a tree latch only.

3. Motivation and Problem Formulation

3.1 Coherence Cache Misses Caused by Latching

For the node-level CC of main-memory index, latches are typically placed inside the index node. A latching operation, whether it is for acquiring or releasing a latch or whether it is a shared-mode or an exclusive-mode, involves a memory write. With the conventional index CC schemes, the invalidation of the cache block containing the latch occurs even if the index is not updated. It is possible to separate latches from index nodes, but such separation does not bring any advantage because the cache blocks containing the latches are subject to the coherence cache miss anyway.

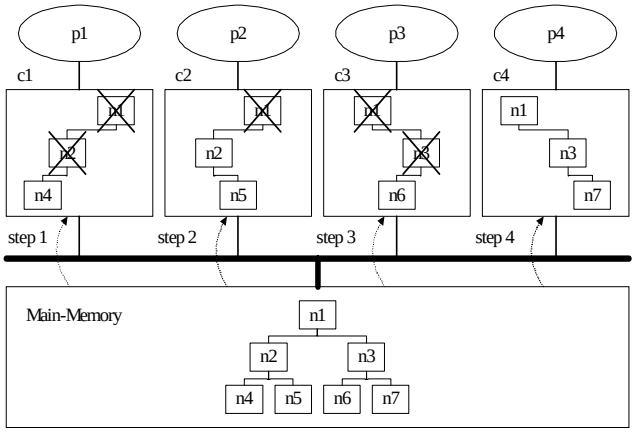


Figure 1. Coherence Cache Miss Caused by Latching

Figure 1 illustrates how the coherence cache misses occur with the index tree consisting of nodes n1 to n7. For the simplicity, let's assume that each node corresponds to a cache block and contains a latch. Let the processor p1 traverse the path n1, n2, and n4 upon the cold start of the main memory query processing system, and then these nodes are replicated in the cache c1 of p1. Latches are held and released on n1, n2, and n4 on the way. Let p2 traverse the path n1, n2, and n5, then these nodes are copied into c2 and the nodes n1 and n2 cached in c1 are invalidated by p2 latching on these nodes. Note that these cache invalidation occurs even if there is enough room in the cache c1. If p3 traverses the path n1, n3, and n6, then the n1 in c2 gets invalidated. Finally, if p4 traverses the path n1, n3, and n7, the n1 and n3 in c3 get invalidated.

3.2 Analysis for Problem Formulation

To show the problem of latching index nodes for concurrency control, we first analyze the probability of coherence cache misses. We assume that index nodes are latched once before accessing them. This is a good approximation even for the lock coupling that acquires a latch and releases it soon during the traversal because the interval between the acquisition and the release is very short. We also assume that the size of the cache memory is infinite and each node has been already cached in a processor to isolate the effect of capacity and cold cache misses, respectively. Finally, we assume that all processors perform the same task and the probability of a node being accessed by a specific processor is identical to its probability of being accessed by another processor.

A cached node is invalidated if another processor, other than the one that caches the node, accesses the node. Therefore, the probability that the coherence cache miss occurs for a node cached in a processor is the same as the probability that another processor accesses the same node. Let p denote the number of processors in the system. Then, the $(p - 1)$ processors among the p processors can

invalidate the node cached in a specific processor. Since each processor has the same probability of accessing the node, the probability of coherence cache miss is:

$$\frac{p-1}{p} \quad (1)$$

This formula leads to a conclusion that the probability of coherence cache misses increases with the number of processors. Thus, if we don't pay attention to the coherence cache miss, even the *infinite* cache memory may not be very helpful for reducing the memory access cost.

We showed that if we can read nodes consistently without latching them as we propose in the OLFIT scheme, we can save many cache misses caused by latching during index search. Now we will show that we can also save many cache misses with latch-free traversal during index updates. We support this claim by showing that even during index inserts, many nodes are only read without updating them.

If we denote the maximum fanout of nodes as f_M, the probability that a leaf node split by an insert operation is as follows [JS93]:

$$\frac{1}{0.69\ f_M}$$

If we denote the level of a node as l, and the height of the tree as h, since a split of a child is an insert of entry into its parent node, the probability of update at a level l is:

$$\left(\frac{1}{0.69\ f_M}\right)^{(h-l)} \quad (2)$$

Figure 2 shows the graph calculated from the equation (2) as well as the relative frequency of the actual updates at a specific level l measured when 1 million keys are inserted into a B+-tree with 10 million keys. We choose f_M = 15, because it is the maximum fanout of the B+-tree whose node size is two cache blocks. This B+-tree will be used as the basis of our experimental evaluation to be presented in section 6. With f_M = 15, $h = \lceil \log_{15\times0.69} 10{,}000{,}000 \rceil = 7$. The conclusion that we can draw from Figure 2 is that most accesses to non-leaf nodes are read-only even when all index operations are insertions.

3.3 Optimistic, Latch-Free Index Traversal CC

The purpose of the index CC is to guarantee that index readers reach the correct leaf nodes without interfering with concurrent index updaters, and that index updaters do not interfere with each other. One of the key criteria for judging whether a concurrency control scheme is good is the degree of parallelism that it provides. The objective of this subsection is to draw the rationale for the latch-free traversal for both the index readers and updaters.

In the previous analysis, we found that most of actual B+-tree index node updates occur at the leaf level or near the leaf level. Namely, the probability that the update at

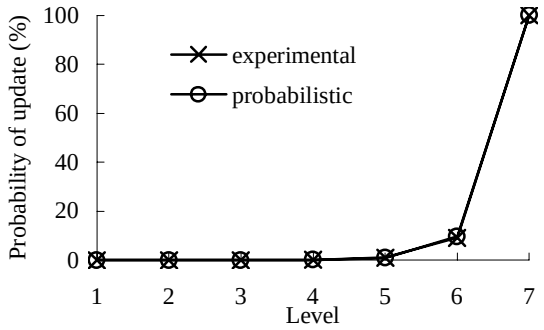


Figure 2. Probability of Update with 100% Insert Workload

the leaf node of the B+-tree propagates to the upper level nodes decreases sharply with the distance from the leaf. In this regard, the conventional index CC schemes that latch the nodes during the traversal are too conservative when applied to the in-memory B+-tree or the CSB+-tree, which is significantly deeper than the disk-resident B+-tree because its node size is given in the order of up to a few times of the L2 cache block size. Most of latches are needlessly acquired and released by both index readers and updaters, especially for the upper level nodes, even if the probability of actually updating such nodes is extremely low.

The duration of holding a latch is very short. If we consider that the number of leaf nodes is a lot bigger than the typical number of processors available in today's multiprocessor platforms, the probability of the conflict among the concurrent index readers and updaters is also extremely low. So the latches on leaf nodes are also acquired too pessimistically.

In the disk-resident database systems, the cost of latching may not be significant because of the dominant cost of disk access. However, in the main-memory database systems, the latching cost is a dominant portion of the processing cost of read transactions, and also is a nontrivial portion of the processing cost of update transactions. Thus the latching overhead is one of the primary factors limiting the scalability of the index performance on the multiprocessor platform. This will be proved later in the experimental evaluation by the poor scalability of the lock coupling scheme on the multiprocessor platform.

Based on these observations, we propose the OLFIT, an index CC scheme that traverses down the index optimistically without latching any nodes. The index updaters start acquiring the latch from the leaf node after reaching the leaf node to update. The latch acquisition proceeds upward if the update of a node leads to the node split or the node deletion.

To guard the latch-free index traversal from interfering with node updates and also to guard the node updates from interfering with other node updates, we include the version number and the latch in the B+-tree index node and provide a pair of node read and update

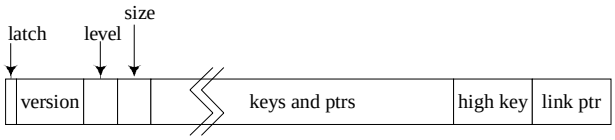


Figure 3. Node Structure of B+Tree for OLFIT

primitives that use this information. Updaters hold a latch on a node before the update, increment the version number, and release the latch after the update. This guarantees that the updater changes a node from a consistent state into another consistent state. Readers read the node latch to check if there is any updater and read the version number twice before and after reading the node to verify whether the nodes they read are in a consistent state. Readers retry reading until the verification succeeds. However, the probability of retrial is very low because of the first and the second observations that we made in the above.

In the following two sections, we first present the pair of node read and update operations used by the OLFIT scheme, and then present how the index-level traversal and update work.

4. Node Operations for OLFIT

Figure 3 shows the structure of the B+-tree node used by the OLFIT. The `latch` and the `version` are added for the OLFIT. The `level` represents the level of the node in the tree and the `size` represents the number of entries in the node. The usage of the `high key` and the `link ptr` is explained in the section 5. In the following algorithms for reading and updating nodes, we assume reading and writing of words are atomic, as are in most of the modern architectures.

Algorithm `UpdateNode`
`U1`. Acquire `latch`.
`U2`. Update the content.
`U3`. Increment `version`.
`U4`. Release `latch`.

Update operations on one node are isolated and serialized using the `latch` on the node. Updaters acquire the `latch` on the node for the exclusive access before updating the content of the node: `version`, `level`, `size`, `keys`, `ptrs`, `high key`, and `link ptr`. The `latch` is released after the update. Updaters also increment the `version` before releasing the `latch`. Differently from node reads, node updaters always succeed and there is no need for retry. The U3 step of incrementing the `version` is intended to enable readers to read the node without latching and verify whether the node they read is in a consistent state or not.

Algorithm `ReadNode`
`R1`. Copy the value of `version` into a register *R*.
`R2`. Read the content of the node.
`R3`. If `latch` is locked, go to `R1`.
`R4`. If the current value of `version` is different from the copied value in *R*, go to `R1`.

The steps `R3` and `R4` of `ReadNode` guarantee readers only read a node in a consistent state without holding any latches. Readers can pass both `R3` and `R4` only if the data read in `R2` is in a consistent state, otherwise readers start again from `R1`. In other words, if the content of a node read in `R2` is in an inconsistent state, either the condition in `R3` or the condition in `R4` becomes true and the reader cannot pass both R3 and R4.

An implementation of `ReadNode` and `UpdateNode` can be directly derived from the algorithm description if we just implement latching using one of the atomic read-and-write operations such as `test-and-set` and `compare-and-swap` that are universally available on the modern computer architectures. However, since consuming two words for the latch and the version per every node is expensive given the node size in the order of one or two cache blocks, we combine these two words into a single word named `ccinfo` in the Figure 4. The LSB of `ccinfo` is used for the latch and other bits are used for the version number. This combination enables further optimization of using only one conditional jump for checking the conditions of R3 and R4 in `ReadNode` algorithm.

In Figure 4, the operation `compare-and-swap(A, B, C)` is an atomic operation that compares the values of `A` and `B`, and if they are equal, replaces the value of `A` by the value of `C` and returns the original value of `A`. `Turn-on-LSB(word)` and `turn-off-LSB(word)` are bit operations that turn on and off the LSB of a `word`, respectively.

The procedure `update_node` in Figure 4 is an implementation of the algorithm `UpdateNode`. The step 2 of the procedure `latch` copies the value of `ccinfo` with its LSB turned off. The step 3 of the procedure `latch` checks whether the LSB of `ccinfo` is turned on or not by comparing the value of `ccinfo` and the value of `t` copied in the step 2, and atomically turns on the LSB of `ccinfo` by replacing the value of `ccinfo` by `t` with its LSB turned on. The procedure `unlatch` releases the latch and increments the version number by increasing `ccinfo`. Since the LSB of `ccinfo` is turned on in the procedure `latch`, the increment in the procedure `unlatch` turns off the latch and increments the version number simultaneously.

The procedure `read_node` in Figure 4 is an implementation of the algorithm `ReadNode`. The step 4 of `read_node` checks conditions in `R3` and `R4` of `ReadNode` simultaneously, because if `t` is equal to `ccinfo`, the LSB of the current `ccinfo` must be turned off and other bits must have not been changed since step 2.

```
procedure latch(word) {
1.   do {
2.      t:= turn-off-LSB(word);
3.   } while (compare-and-swap
            (word, t, turn-on-LSB(t)) ≠ t);
}

procedure unlatch(word) {
   word:= word + 1;
}

procedure update_node(node) {
1.   latch(node.ccinfo);
2.   // Update the content of node
3.   unlatch(node.ccinfo);
}

procedure read_node(node) {
1.   do {
2.      t:= turn-off-LSB(node.ccinfo);
3.      // Read the content of node
4.   } while (t ≠ node.ccinfo)
}
```

Figure 4.Implementation of `UpdateNode` and `ReadNode`

5. Tree Operations for OLFIT

5.1 Dealing with node split

Since the OLFIT does not use lock coupling while traversing down a tree index, concurrent updaters may split the target child node of the traversal before the traversal reaches the child node. Moreover, since no latch is held while reading a node, concurrent updaters may also split the node currently being read. To deal with this problem, we use the technique of using a high key and a link pointer proposed by [LY81] and improved by [Sag85]. All splits are done from the left to the right and to each node, a high key and a link pointer are added. The high key is the upper bound of the key values in the node and the link pointer is the pointer to the right neighbor of the node. The purpose of the link pointer is to provide an additional method for reaching a node and whether to follow the link pointer or not can be determined by the high key. With this link pointer, since splits are done from the left to the right and each node has its high key and its link pointer to the right neighbor, all nodes split from a node are reachable from the node and the correct child node can be reached in the presence of concurrent splits of nodes.

5.2 Tree traversal algorithm

Figure 5 shows the pseudo code for the tree traversal. The `find_next(node, key)` primitive finds the next node to traverse. If the `key` is greater than the high key of the node, this operation returns the link pointer. Otherwise, it returns the pointer to the appropriate child node to traverse down.

```
procedure traverse(root, key){
1.   node:= root;
2.   while (node is not a leaf) {
3.      t:= turn-off-LSB(node.ccinfo);
4.      next:= find_next(node, key);
5.      if (node.ccinfo = t) node:= next;
6.   }
7.   return node;
}
```

Figure 5. Pseudocode of Tree Traversal

The procedure `read_node` is embedded in the procedure `traverse` of Figure 5. The while loop of the `read_node` is removed by assigning the value of `next` to the variable `node` only if the value of `next` is computed from a node in a consistent state.

5.3 Dealing with node deletion

The updater can delete a node being read if the node is empty. To deal with this case, when a node becomes empty, the updater only removes links directed to the node and registers the node into a garbage collector. The value of the link pointer in the empty node is preserved until the node is actually deallocated. The garbage collector actually deallocates the registered node when there are no index operations that can read the node. To determine whether there is any operation that can read the node or not, we use the algorithm originally proposed for the physical versioning [RS+97]. However, the overhead is quite different, because the physical versioning uses the garbage collector on every update, while we use it only when an updater removes an empty node.

5.4 Putting together with transaction-duration locking

To support the serializability of transactions by transaction-duration locking, both the locking protocol of ARIES/KVL [Moh90] and that of ARIES/IM [ML92] can be combined with our OLFIT algorithm without any modification. The OLFIT replaces only the latching protocols of ARIES/KVL and ARIES/IM.

5.5 Adaptation to CSB+-tree

In the CSB+-tree, nodes with the same parent are clustered in a contiguous space called node group and non-leaf nodes of the CSB+-tree store only the pointer to the child node group instead of storing all pointers to child nodes. CSB+-tree nodes for the OLFIT only store high keys without link pointers because the right neighbor in the same node group can be located without a pointer. Only one link pointer to the right node group is stored for each node group to locate the right neighbors in other node groups. The link pointer has its own latch and version, because the link pointer does not belong to any node in the node group. Here, the CSB+-tree means the full CSB+-tree. Extensions to other variations of the CSB+-tree are straightforward.

The split operation of the CSB+-tree is different from that of the B+-tree because nodes with the same parent are clustered in a node group. When a node splits, if the node group is not full, all right siblings in the node group shift right to make a room for the split. In this case, the node to split, all shifting nodes, and the parent node are latched before the split. If the node group is full, the node group splits into two node groups. In this case, the node to split, the shifting nodes, the parent node, the nodes to be moved into the new node group, and the link pointer to the right node group are latched.

6. Experimental Evaluation

To verify the superior scalability of the OLFIT scheme experimentally, we implemented five index CC schemes for the B+-tree and the full CSB+-tree: lock coupling with node latches (LC), tree-level locking with a tree latch (TL), physical versioning with node latches (VN), physical versioning with a tree latch (VT), and OLFIT (OL) proposed in this paper. The performance of index operations without concurrency control (NO) is also measured for the 100% search and the single thread experiment. For the lock coupling, we used the optimistic descent algorithm in [BS77], and for the physical versioning schemes, we adapted the algorithm for the T-tree in [RS+97] to the B+-tree and to the full CSB+-tree.

We ran our experiment on a Sun Enterprise 5500 server with 8 CPUs (UltraSPARC II, 400MHz) running Solaris 7. Each CPU has 8MB L2 cache whose cache line size is 64 bytes. We ran each concurrency control scheme with following workloads: 100% search, 100% insert, 50% insert + 50% delete and a mixture of search, insert and delete with varying update ratio. We do not show the graph of 100% insert due to lack of space. The graph is very similar to that of 50% insert + 50% delete.

For the fair comparison with the physical versioning schemes that require substantial memory allocation for updates, we used memory pools to reduce the overhead of system calls for memory allocation. We created the same number of memory pools as the number of processors in the system, and assigned one to each thread to minimize the contention for the memory allocation.

For each experiment, we used non-unique indexes that allow duplicate key values, and the indexes are initialized by the insertion of 10 million uniformly distributed 4-bytes integer keys and associated pointers. The size of pointers is 4 bytes because we ran our experiment in the 32-bit addressing mode. After the initialization, the height of B+-trees is 7 and the height of full CSB+-trees is 6, and their sizes are about 140 MB and 190 MB, respectively. Indexes are about 70% full because they are initialized by insertions [JS93].

		OL	LC	TL	VN	VT
B+-tree	Leaf	14	15	15	15	15
	Non-leaf	14	15	16	15	16
CSB+-tree	Leaf	14	15	15	15	15
	Non-leaf	28	29	30	29	30

Table 1. Max Node Fanout of Various CC Schemes

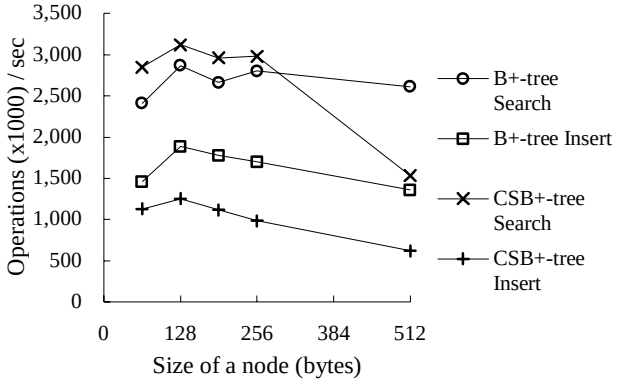


Figure 6. Performance of OLFIT with Varying Node Size

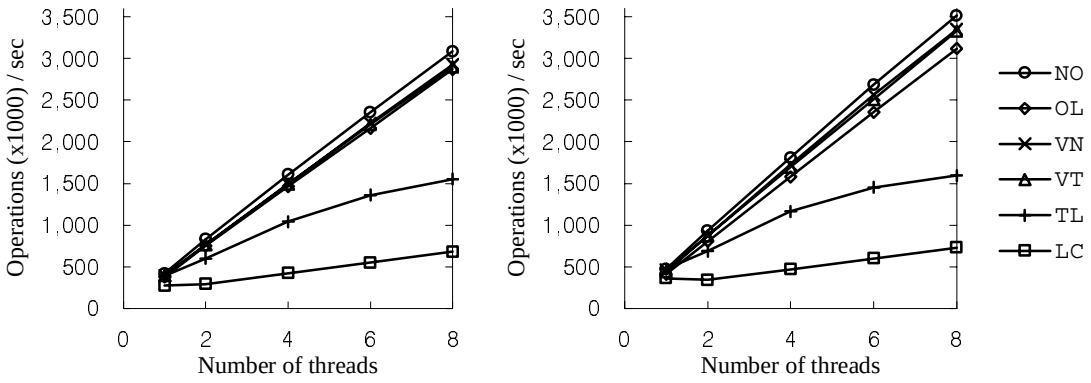
We chose 128 bytes for the size of index nodes because the 128-byte node produces the best performance when the indexes are built by insertions. The eight-thread performance of the OLFIT with varying node size is shown in Figure 6. The 128-byte node is the best as shown in the graph because with 70% full nodes on average, there is a high probability of accessing only the first 64-byte block of 128-byte nodes.

Table 1 shows the maximum fanout of nodes when each concurrency control scheme is applied. TL and VT allow the largest fanout because they need no concurrency control information on nodes. LC and VN allow the smaller fanout because they need a latch on each node, and OL allows the smallest fanout because it needs a latch, a version, a high key and a link pointer on each node.

6.1 Pure search performance

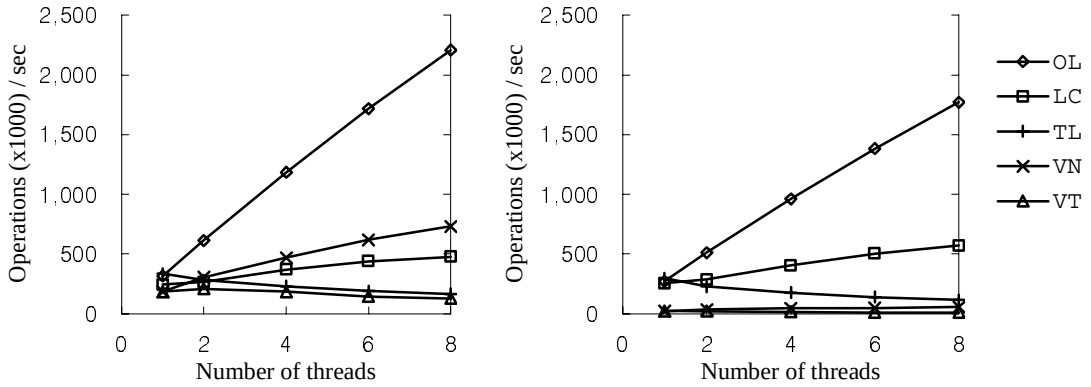
Figure 7 shows the scalability of the search performance of the experimented CC schemes. We measured the throughput of exact match search varying the number of threads that perform the search operations. The x-axis shows the number of threads performing search operations, and the y-axis shows the aggregated throughput.

The OLFIT (OL) and the physical versioning schemes (VN, VT) are similar to the no concurrency control (NO). The gap between these schemes and no concurrency control is the cost of the interaction with the garbage collector. The gap between OL and NO on the CSB+-Tree is wider than on the B+-tree, because of the different cost of checking the high key. For the B+-tree, high keys need not be specially treated on non-leaf nodes because traversing to the right neighbor and traversing down to one of the children are not different. However, for the CSB+-tree, traversing to the right and traversing downward are different: positions of children are



(a) B+-tree (b) Full CSB+-tree

Figure 7. Search Performance of Concurrency Control Schemes



(a) B+-tree (b) Full CSB+-tree

Figure 8. Insert and Delete Performance of Concurrency Control Schemes

computed from the pointer to the child node group while the position of the right neighbor is computed from the position of the current node. This special treatment on high keys consumes slightly more time and makes the small gap from the physical versioning schemes.

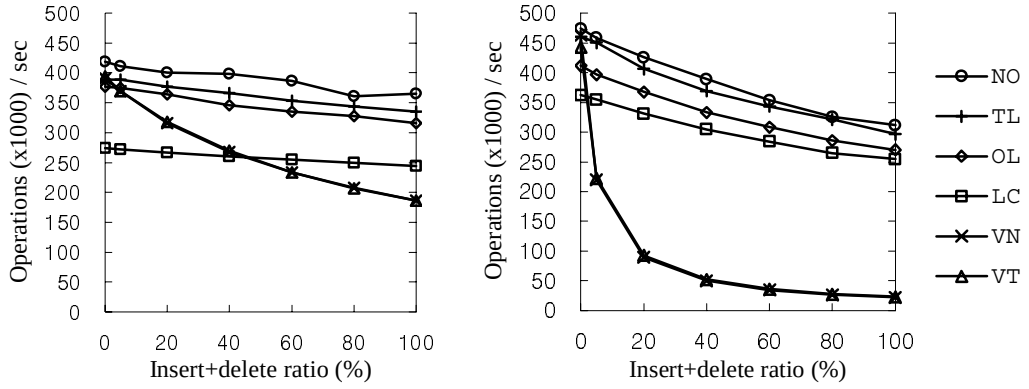
The tree-level locking (TL) becomes worse as the number of threads increases due to the contention at the tree latch and two more coherence cache misses generated by holding and releasing the tree latch. The performance of the lock coupling (LC) is worst and the gap from the no concurrency control widens almost linearly with the number of threads, because the lock coupling generates many coherence cache misses by latching many nodes.

6.2 Pure update performance

Figure 8 shows the scalability of the update performance of the experimented CC schemes. We measured the update throughput, varying the number of threads that perform updates. The x-axis shows the number of threads, and the y-axis shows the aggregated throughput. Half of operations are insertions and the other half are deletions.

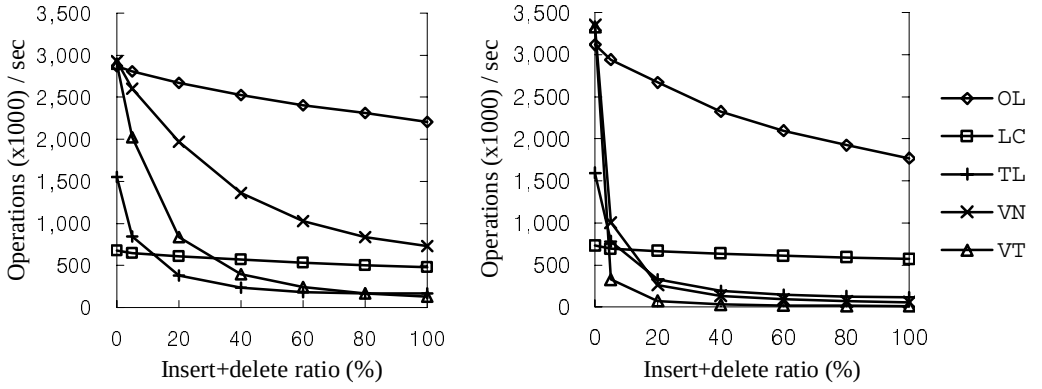
In Figure 8, only OLFIT (OL) shows a scalable performance. The physical versioning schemes (VT, VN) show poor update performance due to the high cost of versioning, especially for the CSB+-tree where the whole node group must be versioned for each update. Differently from [RS+97], VN is better than VT in update performance because we changed the algorithm of VN that was originally proposed for the T-tree in the process of adapting it to the B+-tree. In [RS+97], if structure modifications take place, VN holds a tree latch. We eliminated the need for the centralized tree latch from VN because the split of the B+-tree is different from the rotation of the T-tree and the centralized tree latch is not needed for structure modifications. The centralized tree latch degenerates the update performance significantly as the number of threads increases.

Although the physical versioning with node latches (VN) and the lock coupling (LC) do not produce good performance, their performance increases slowly as the number of threads increases. However, the physical versioning with a tree latch (VT) and the tree-level locking (TL) degrades due to the contention on the centralized tree latch as the number of threads increases.



(a) B+-tree (b) Full CSB+-tree

Figure 9. Single Thread Performance with Varying Update Ratio



(a) B+-tree (b) Full CSB+-tree

Figure 10. Eight-Thread Performance with Varying Update Ratio

6.3 Performance with varying update ratio

Figure 9 and Figure 10 show the throughput of a single thread and eight threads, respectively, for the mixed workload with varying update ratio. The workload consists of a sequence of one million operations, each of which is randomly decided to be search, insert, or delete based on the given update ratio. Updates are evenly divided into inserts and deletes.

In Figure 9 showing the result of sequential execution, the OLFIT (OL) and the tree-level locking (TL) are similar to the no concurrency control (NO). The lock coupling (LC) shows worse performance than OL and TL due to the overhead of latching. The physical versioning schemes (VN, VT) show good performance for no updates, but as the update ratio increases, their throughput drops sharply due to the high overhead of versioning. As shown in Figure 9 (b), the versioning overhead is even heavier for the CSB+-tree because in the CSB+-tree, nodes with the same parent are grouped in a contiguous space and versions are created per node group basis.

Figure 10 shows the result with eight threads. When the ratio of update is zero, the performance of the physical versioning schemes (VN, VT) is comparable to that of the OLFIT (OL). However, as the update ratio increases, OL becomes significantly better than any other concurrency control algorithms.

The performance gap between OL and other schemes is wider with eight threads than with a single thread. The gap widens partly due to the large amount of coherence cache misses generated by other algorithms and partly due to the contention at the centralized tree latch in the case of TL and VT. Note that VT is slightly better than TL but it approaches to TL as the update ratio further increases and eventually crosses TL due to the high cost of versioning.

The performance drops more sharply for the CSB+-tree than the B+-tree with the update ratio mainly because of the higher split cost of the CSB+-tree. In the CSB+-tree, when a node splits, if the node group that contains the node is not full, all the right neighbors of the node in the same node group are shifted right, and if the node group is full, half of nodes in the group are moved into a new node group. Note that VN and VT are even worse than TL as the update ratio exceeds 20% due to the high cost of versioning.

7. Conclusion and Future Work

This paper addressed the importance of considering the cache effect, especially, the coherence cache miss overhead associated with latching main-memory index nodes for the concurrency control on multiprocessor platforms, and proposed a new optimistic index CC called OLFIT for the B+-tree and the CSB+-tree. Observing that most of latches are held too conservatively for the cache-conscious main-memory index, this scheme completely eliminates latching during the index traversal. Even the index update does not incur any latching operation until the traversal reaches the node to update. Latches are requested upward upon the split of nodes.

To prevent the index updates from interfering with index reads and other updates, the OLFIT first provides a pair of consistent node read and update operations. Based on this pair, we presented the design of the tree search and update operations. An experiment comparing the OLFIT with various representative index CC schemes for the B+-tree and the CSB+-tree shows that the OLFIT shows superior update scalability on the eight-CPU multiprocessor system while showing the read scalability comparable to those of the no concurrency control.

Although we presented the OLFIT algorithm for the B+-tree and the CSB+-tree, it can be easily adapted to other cache-conscious index schemes such as the CR-tree, the cache-conscious version of the R-tree. For the future work, we are integrating the OLFIT with P*TIME, a highly parallel transact in memory engine, to investigate the impact of the OLFIT in the real main memory database applications.

References

[BMR01] Philip Bohannon, Peter McIlroy, and Rajeev Rastogi, "Improving Main-Memory Index Performance with Partial Key Information," In *Proc. of ACM SIGMOD Conf.*, 2001.

[BS77] Rudolf Bayer and Mario Schkolnick, "Concurrency of Operations on B-Trees," *Acta Informatica 9*, 1977, pages 1-21.

[CSG99] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta, *Parallel Computer Architecture*, Morgan Kaufmann Publishers, Inc., 1999.

[JS93] Theodore Johnson and Dennis Shasha, " The Performance of Current B-Tree Algorithms," *ACM TODS*, Vol. 18, No. 1, March 1993, pages 51-101.

[KCK01] Kihong Kim, Sang K. Cha, and Keunjoo Kwon, "Optimizing Multidimensional Index Trees for Main Memory Access," In *Proc. of ACM SIGMOD Conf.*, 2001.

[LC86] Tobin J. Lehman and Michael J. Carey, "A Study of Index Structures for Main Memory Database Management Systems," In *Proc. of VLDB Conf.*, 1986, pages 294-303.

[LKC01] Juchang Lee, Kihong Kim, and Sang K. Cha, "Differential Logging: A Commutative and Associative Logging Scheme for Highly Parallel Main Memory Database," In *Proc. of IEEE ICDE Conf.*, 2001.

[LY81] Philip L. Lehman and S. Bing Yao, "Efficient Locking for Concurrent Operations on B-Trees," *ACM TODS*, Vol. 6, No. 4, 1981, pages 650-670.

[MH+92] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh and Peter Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM TODS*, Vol. 17, No. 1, 1992, pages 94-162.

[ML92] C. Mohan and Frank Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," In *Proc. of ACM SIGMOD Conf.*, 1992, pages 371-380.

[Moh90] C. Mohan, "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes," In *Proc. of VLDB Conf.*, 1990, pages 392-405.

[RR99] Jun Rao and Kenneth Ross, "Cache Conscious Indexing for Decision-Support in Main Memory," In *Proc. of VLDB Conf.*, 1999, pages 78-89.

[RR00] Jun Rao and Kenneth Ross, "Making B+-trees Cache Conscious in Main Memory," In *Proc. of ACM SIGMOD Conf.*, 2000, pages 475-486.

[RS+97] Rajeev Rastogi, S. Seshadri, Philip Bohannon, Dennis Leinbaugh, Avi Silberschatz, and S. Sudarshan, "Logical and Physical Versioning in Main Memory Databases," In *Proc. of VLDB Conf.*, 1997, pages 86-95.

[Sag85] Yehoshua Sagiv, "Concurrent Operations on B*-Trees with Overtaking," In *Proc. of SIGACT/SIGMOD PODS*, 1985, pages 28-37

[Sun97] Sun Microsystems, *UltraSPARC™ User's Manual*, 1997.