Concurrent B-trees with Lock-free Techniques

Afroza Sultana
Concordia University

Helen A. Cameron
University of Manitoba

Peter C. J. Graham
University of Manitoba

October 23, 2011

Abstract

B-trees and their variants are efficient data structures for finding records in a large collection (e.g., databases). The efficiency of B-trees increases when a number of users can manipulate the tree simultaneously. Many algorithms have been developed over the last three decades to achieve both concurrency and consistency in B-trees. However, current lock-based concurrency-control techniques limit concurrency. Moreover, lock-based B-trees suffer from certain negative scheduling anomalies, such as deadlock, convoying and priority inversion. Lock-free concurrency-control techniques using, for example, Compare And Swap (`CAS`) can provide improved concurrent access to data structures including B-trees and other search structures. Besides this, correctly designed lock-free techniques prevent deadlock, convoying and priority inversion. Considering the advantages of lock-free techniques for other concurrent data structures, we develop a lock-free B-tree to support high performance concurrent in-memory searching in a Non Uniform Memory Access (NUMA) parallel computing environment.

The use and parallelization of B-trees have both been widely explored in the past—primarily for application to database implementation and, hence, disk-based operations. Moving B-trees into memory for use in new online searching applications, however, fundamentally changes the characteristics of managing them and will allow us to effectively exploit the use of lock-free techniques, something that has previously not been applicable to B-trees.

1 Introduction

Historically, memory capacity was limited, so large data collections had to be stored on disk in databases, which use data structures such as B-trees. With the availability of large memories, this restriction has been relaxed. Correspondingly, a number of new applications have emerged in such fields as bio-informatics and computational linguistics that require searching huge collections in memory. A B-tree-like data structure (built in memory) is still a good solution for such problems.

To achieve concurrency in B-trees, various lock-based algorithms have been designed, but those algorithms suffer from a number of problems. For example, some algorithms limit efficiency: when a process manipulates some nodes in a B-tree, all other processes that want to manipulate the same nodes are forced to wait, even if concurrent access would not cause any inconsistencies. Additionally, the use of locks introduces some negative side effects related to scheduling. To address these problems, we designed lock-free algorithms for a B-tree variant, where multiple processes can work on the tree simultaneously. More specifically, we developed a locality-of-reference-efficient version of Lehman and Yao's B^{link}-tree [12] that does not require locking and exploits memory management ideas from the work of Michael [15].

Our algorithms are also designed to tolerate variation in memory access times present in the increasingly common Non Uniform Memory Access (NUMA) class of parallel machines. The in-memory support is in contrast to the existing literature on B-trees, which focuses on operations performed on-disk. Lehman and Yao's lock-based B^{link}-tree achieves better concurrency than our B^{list}-tree. However, when comparing a variant of the lock-based B^{link}-tree that has the same structural complexity in the nodes as our B^{list}-tree (see Section 5.1 below), our B^{list}-tree performs better than the linked-list-based variant of the B^{link}-tree

when there is no thread sharing in the same processor. Furthermore, our lock-free B^{list}-tree is not prone to deadlock, priority inversions and convoying, unlike the lock-based B^{link}-trees.

A Non-Uniform Memory Access (NUMA) architecture [9] is a type of shared memory architecture where the shared memory is divided into segments and each segment is attached to one processor with a connecting bus. A processor in a NUMA architecture can access its associated memory directly through the bus, but it has to connect through the interconnection fabric to access the memory attached to other processors. Thus, a processor has faster access to its associated memory than other memories in the parallel machine.

In a shared memory architecture, if more than one process wants to update the same data at the same memory location at the same time, the content of the data might become inconsistent.

In pessimistic concurrency control, if there is a vulnerable portion of code (a *critical section*) where concurrent execution of that code by different processes might cause inconsistency in the data, then only one process is permitted to execute that portion of code at a time. The `Test-And-Set (TAS)` primitive is commonly used to implement most major pessimistic concurrency-control mechanisms, including locks.

Unlike pessimistic concurrency control, optimistic concurrency-control techniques assume that there will seldom be conflicts in accessing a shared data object. They allow concurrent accesses in a critical section without checking whether the access is safe. Instead, after computing the new value of a shared data object, but before storing it, the value used in computing the new value and the current value of the shared data object are compared. If both values are the same, the old value may be safely replaced by the new value since the shared object has not been changed. Otherwise, the computation must be redone using the changed value. Comparing the values of the shared object and then possibly updating the object must be done atomically[1]. The most common examples of optimistic concurrency control are the so-called lock-free techniques. Lock-free techniques use universal primitives [8] like CAS and DCAS to build other lock-free objects like lock-free linked lists [15], queues [16], and different types of trees [10, 13].

Non-blocking techniques and *wait-free* techniques are two types of widely-used lock-free techniques. If a lock-free technique guarantees that some processes will complete their operations in a finite number of steps, then that lock-free technique is *non-blocking*. If the lock-free technique guarantees that every process will complete its operations in a finite number of steps, regardless of the execution speeds of the other processes, then that lock-free technique is *wait-free*.

Optimistic concurrency-control algorithms have some advantages over pessimistic concurrency-control algorithms in the absence of frequent access conflicts between concurrent processes. Algorithms based on CAS (and similar primitives) have almost no overhead when compared to lock-based algorithms. Using lock-based algorithms, whether there are conflicts or not, a process needs to invoke the operating system to test the lock of the critical section. Further, other processes cannot access the critical section concurrently. Therefore, if no conflict occurs between processes, optimistic algorithms can accommodate more concurrency with less overhead than lock-based algorithms.

However, in the presence of frequent conflicts, optimistic algorithms may take more time than lock-based algorithms, since processes will need to recompute new values when other processes have changed the shared data objects. Designing an intelligent, cost-effective lock-free algorithm can, however, keep the cost at an acceptable level.

2 Related Work

Several concurrent B-tree algorithms were designed for use in databases in the mid 70's, but others are more recent. All of the algorithms presented here are pessimistic lock-based algorithms, where a process locks anywhere between a single node to a portion of the tree while performing an operation on that tree.

The first concurrent B-tree algorithm was developed by Samadi [19], whose approach was very simple. When a process needs to perform an insertion or deletion, Samadi's algorithm locks the entire subtree rooted at the node being manipulated. Bayer and Schkolnick [1] improved Samadi's algorithm by introducing write-exclusion and exclusive locks, where a concurrent process is allowed to read, but not write on a node (or

[1] An *indivisible* or *atomic* segment of code is guaranteed to execute in its entirety without any other process interfering.

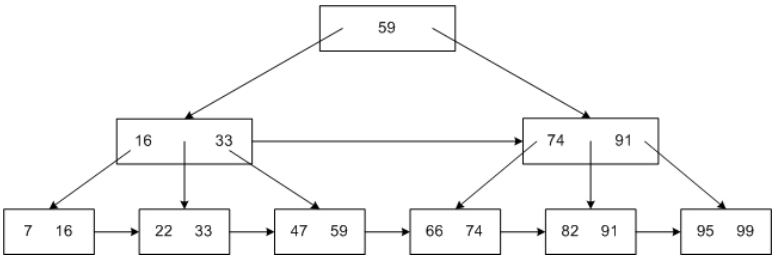

Figure 1: An example of a B^{link}-tree.

leaf) with write-exclusion locks, and a process is allowed neither to read nor to write on a node with an exclusive lock. The write-exclusion locks increase the concurrency compared to the regular locks used in Samadi's algorithm, but only by a limited amount.

Some other concurrent B-tree algorithms were developed by Miller and Snyder [17], Ellis [5], and Guibas and Sedgwick [6], but none of them make significant enough improvements to the algorithms described above to warrant detailed discussion.

Lehman and Yao [12] came up with an algorithm that introduced a bottom-up restructuring method for B-trees. Most later algorithms, such as the overtaking algorithm of Sagiv [18], the symmetric concurrent algorithm of Lanin and Shasha [11], the operation-specific lock algorithm of Biliris [2], the improved overtaking algorithm by de Jonge and Schijf [4], and the B^{mad}-tree algorithm of Das and Demuynck [3] were influenced by Lehman and Yao's algorithm.

2.1 Lehman and Yao's B^{link}-tree

The B^{link}-tree of Lehman and Yao [12], like B^*-trees, has pointers to records in leaf nodes only. Non-leaf nodes contain only routing values to guide searches to the appropriate leaf. Each B^{link}-tree node has an extra pointer, known as the link pointer, that points to the node immediately to its right at the same level in the tree. The link pointers of the rightmost nodes at each level are null pointers. When a node is split, the new node is made the sibling immediately to the right of the splitting node, so keys only move to the right in a split. Therefore, as a result of any manipulation in the B^{link}-tree, if any key is moved to a node to the right, the link pointers can be used to find that key. Also, each node in a B^{link}-tree has a `highkey` field that contains the value of the highest key in the subtree rooted at that node. Figure 1 shows an example B^{link}-tree.

The B^{link}-tree has parallel search, insert and delete operations that are similar to the basic non-parallel B^*-tree search, insert and delete operations. The update (insert and delete) operations in a B^{link}-tree use locks to conduct concurrent operations. However, no locking is required during searching in a B^{link}-tree. Use of the right-link pointers and comparison of the desired key with the `highkey` of a node or a leaf ensures the concurrency control in the search operations. When the search process encounters a node (or a leaf) N, the search process compares the key, k, to be searched for with the `highkey`, hk_N, of node N. If $hk_N < k$, then the search process knows that a concurrent insert process has split N and moved half of the keys to its right-sibling node. In that case, the search process follows right-link pointers starting from N until it finds a node whose `high-key` is greater than the desired key k.

The B^{link}-tree insert algorithm locks the entire node or a leaf when it inserts a key in that node or leaf. Also, when an insert process gets a parent node, F, of a node, N, from the stack (where a search stores its search path through the tree), due to concurrent splits of F, the node on the stack might no longer be the current parent of the node N. To solve this problem, an insertion process compares the `highkey`, hk_F, of F with the `highkey`, hk_N, of N. If hk_F happens to be less than hk_N, then the insert process has to follow right-link pointers from F to find the appropriate sibling of F that is the current parent of N.

The B^{link}-tree delete algorithm also uses locks to accommodate concurrency. Moreover, the B^{link}-tree

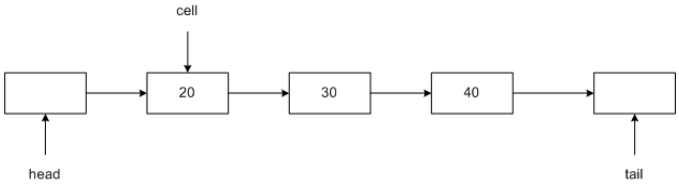

Figure 2: Michael's lock-free linked list.

delete algorithm allows a leaf to have fewer than $\lceil m/2 \rceil$ keys, whereas, in a standard B-tree and its variants, an underfull node is merged with a sibling. Allowing fewer than $\lceil m/2 \rceil$ keys in a node in Lehman and Yao's B^{link}-tree maximizes efficiency since no further locks are required for further merges (as merges are not required). Hence, more concurrency is achieved as other processes can access those unlocked nodes. This enhanced concurrency is useful when different processes perform frequent deletes on a B^{link}-tree. Overall space efficiency may, however, suffer.

2.2 Michael's Lock-free Linked List

Michael [14] designed a safe wait-free memory reclamation technique for lock-free linked lists, queues, stacks and hash tables. His memory reclamation technique assures that a process can safely return a cell to a free list so that other concurrent processes can reuse the same cell in the future. We describe only Michael's lock-free linked list [15], which uses his safe memory reclamation technique.

Like Valois' lock-free linked list, Michael's lock-free linked list has two dummy cells named `head` and `tail` at the beginning and the end of the linked list, respectively. Moreover, it maintains a free list as Valois' linked list does. However, Michael's linked list does not contain any auxiliary nodes. Figure 2 shows Michael's lock-free linked-list structure.

In his safe memory reclamation technique, Michael introduces k (k is usually not more than three) pointers named `hazard pointers` for each process. If a process is working on some cells of a linked list, the hazard pointers of that process point to those cells. These hazard pointers can be read by other concurrent processes, but can only be written by the associated process. Besides the hazard pointers, each process has its own stack that is used to store the cells it has deleted. These stacks are known as the `retired-cell-stack`. A `retired-cell-stack` is private to its associated process. Therefore, after the deletion of a cell, when the cell is stored in the `retired-cell-stack`, no process other than the deleter process can access that deleted cell.

The insert algorithm of Michael's lock-free linked list is straightforward. First, an insert process finds the correct (ordered) position in the linked list, and then it inserts the new cell in the linked list using a CAS operation. During these operations, the hazard pointers point to those cells being accessed so that other concurrent deletion processes cannot delete and return those cells to the free list.

The deletion algorithm also works in a similar way. First, it finds the cell to be deleted from the linked list. Then, it must delete and return that cell to the free list, assuming that no other processes are currently pointing their hazard pointers to that cell. To avoid deleting a cell that is pointed to by one or more hazard pointers, the deletion algorithm does not completely delete a cell from the linked list. Instead, a delete process first *marks* the cell as deleted. To mark a cell as deleted, a delete process adds 1 to the address stored in the cell's `next` field using a CAS operation. Since the least significant bit of any cell address is always 0, adding 1 sets the least significant bit of the `next` field to 1. Thus, the marking distinguishes a deleted cell from a cell that is not deleted.

After marking a cell as deleted, the delete process stores that cell in its associated `retired-cell-stack`. If the number of deleted cells in the `retired-cells-stack` reaches a previously set threshold level, R, where $R = 2 \times k \times$ number of processes, the delete process calls a `scan` routine to return the cells from the `retired-cell-stack` to the free list of the linked list. The scan routine takes a copy of all the hazard

```
class cell{
    int key;            // contains the key
    int thread-id;      // contains the id of the process (thread) that has deleted this cell
    node *left-child;   // contains the left child of the key
    node *right-child;  // contains the right child of the key
    cell *next;         // contains the address of the next cell
};
```

Figure 3: The `cell` class in a B^{list}-tree.

pointers of each process. The copies of the hazard pointers are stored in a list named the `p-list`. Once all hazard pointers have been copied by the scan routine, it sorts the `p-list`. Then, the scan routine moves the cells stored in the `retired-cell-stack` into a `temporary-stack`. Then, the scan routine compares the address of each cell in the `temporary-stack` with the addresses stored in the `p-list`. If there is a match, then a concurrent process's hazard pointer is pointing to that deleted cell. In this case, the scan routine re-stores the deleted cell on the `retired-cell-stack`. If there is no match found in the `p-list`, the scan routine returns the deleted cell to the free list of the linked list for further use.

When an insert or delete process finishes its insert or delete operations, it sets all of its hazard pointers to null.

3 The B^{list} Tree

We named our tree structure the B^{list}-tree (a B^{link}-tree with a list-based node structure). We based our B^{list}-tree algorithms on Lehman and Yao's B^{link}-tree algorithms [12]. We chose the B^{link}-tree because the link pointers in each node support fast searching, which is the primary operation of the applications we are targeting. Though nodes on the same level of our B^{list}-tree are linked with each other as in the B^{link}-tree, we significantly modified the node structure on our B^{list}-tree to reflect the fact that the tree will be stored in memory, not on disk like a B^{link}-tree.

A B^{list}-tree consists of nodes, both leaf and non-leaf. Each node of a B^{list}-tree has a lock-free `key-list` to store the keys. Each `key-list` consists of a collection of cells where each cell stores a key and descendent nodes with lesser and greater keys, respectively. A B^{list}-tree consists of a root pointer that points to the root node of the tree.

Besides the nodes in the B^{list}-tree, there are some additional structures that help make the tree lock-free and concurrent. Like the B^{link}-tree structure, our B^{list}-tree algorithms use a `path` stack to store the path taken by a process from the root to a leaf of the tree. Besides this, like Michael's lock-free linked-list algorithms [15], our B^{list}-tree algorithms use hazard pointers and a `retired-cell` stack. In addition to these data structures, the B^{list}-tree algorithms use a `please-scan` array with one entry per process to establish communication between the concurrent processes. We describe the data structures that are used in the B^{list}-tree in the following subsections.

3.1 Cell

In each node of our B^{list}-tree, the keys are stored as a linked list known as the `key-list` of the node. Each `key-list` consists of `cells`, where each cell contains a `key`, two children (a `left-child` and a `right-child`) associated with the key, a `thread-id` to indicate which process (if any) has deleted the cell from the `key-list`, and a `next` field to point to the next cell in the `key-list`. Figure 3 show the `cell` structure, and Figure 4 shows an example of two consecutive cells in a B^{list}-tree node.

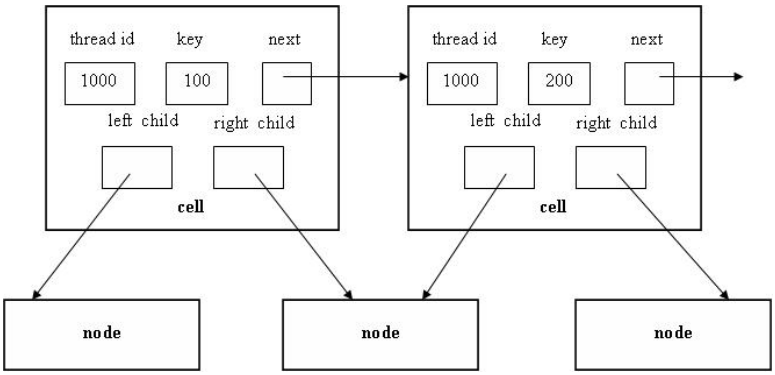

Figure 4: An example of two consecutive cells in a B^{list}-tree.

```
class key-list{
    cell *head;         // the address of the first cell (dummy cell) in the key-list
    cell *tail;         // the address of the last cell (dummy cell) in the key-list
    cell *free-head;    // the address of the first cell in the free cell list
    cell cells[m+2];    // array of [m+2] cells to store the keys
};
```

Figure 5: The `key-list` class in a B^{list}-tree.

3.2 Key-List

Unlike Lehman and Yao's array-based key-list structure in their B^{link}-tree [12], our B^{list}-tree's `key-list` is a lock-free linked list, where keys are stored in sorted order. To design our `key-list`, we followed the concurrent lock-free linked-list structure of Michael [15] (see Section 2.2). However, unlike Michael's lock-free linked list, our `key-list` starts and ends with two dummy cells named `head` and `tail`. These dummy cells do not contain any keys. The `next` field of `head` points to the cell that contains the smallest key in the `key-list`, and the `next` field of the cell that contains the largest key in the `key-list` points to the `tail` of the key list.

Each `key-list` contains $m + 2$ cells, where m is the maximum number of children of a node in the B^{list}-tree. So that cells within a node are allocated from the same chunk of memory, and therefore have good locality of reference. These cells are implemented as an array of size $m + 2$ (pointers between cells are still used to provide the flexibility of a list).

The cells that are not currently being used are referred to as `free cells`, and are stored as a linked list called the `free-list`. The `free-head` pointer points to the first cell of the `free-list`. Figure 5 show the `key-list` structure in a B^{list}-tree node. Figure 6 shows an example of a `key-list` in a leaf of a B^{list}-tree.

3.3 Node

Besides the `key-list`, each node of a B^{list}-tree contains an `is-leaf` field (to indicate whether the node is a leaf or a non-leaf node), a `key-counter` field (to indicate the number of keys currently stored in the node), a `high-key` field (to store the largest key in the descendants of the node), a `split-bit` (to indicate if the node is currently being split by some process), and a `right` field (which points to the node immediately to the right of the node). Moreover, each node of the B^{list}-tree contains a `finally-linked-in-tree` field. When a node is created by a process in a B^{list}-tree, only the process that created the node has authority to access and work on the node. When the creator process turns the node's `finally-linked-in-tree` field on,

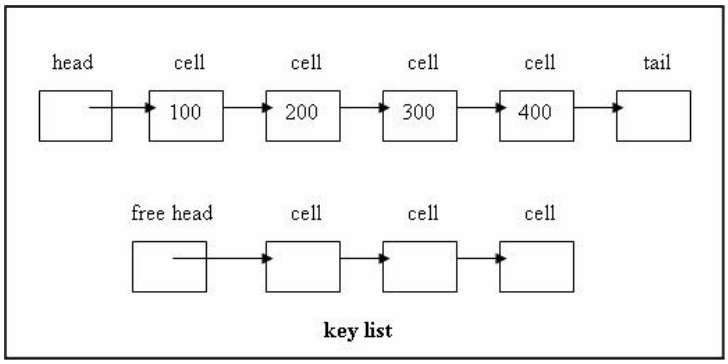

Figure 6: An example of a `key-list` in a leaf of a B^{list}-tree.

```
class node{
    k-list *key-list;   // contains the key-list
    bool is-leaf;       // indicates whether the node is a leaf or a non-leaf node
    int key-counter;    // contains the number of keys in the node
    int high-key;       // contains the high-key of the node
    int split-bit;      // indicates an ongoing splitting process
    bool finally-linked-in-tree;  // indicates whether the node is linked or not in the tree
    node *right;        // contains the address of the immediately right node
};
```

Figure 7: The `node` class in a B^{list}-tree.

other concurrent processes working in the tree can access the node. Figure 7 show the structure of a `node` in a B^{list}-tree.

3.4 The Hazard Pointers Array

Since our B^{list}-tree uses a lock-free linked-list structure to store the keys in each node, we needed to implement `hazard pointers` [15] to allow each process to safely perform concurrent operations including deletion on the `key-list`. Two globally-accessible hazard pointers are assigned to each process that may perform operations on the B^{list}-tree. These pointers are use to prevent hazardous concurrent operations (see Section 2.2 for a detailed description of how hazard pointers do this). Our B^{list}-tree algorithms use two hazard pointers (`hp1` and `hp2`) for each process. A process points its hazard pointers at cells on which the process is currently performing an operation. Hazards pointers keep on pointing to those cells until the the process completes its operation so that any concurrent delete process can be prevented from completely deleting a cell (that is in use) from the `key-list` of the corresponding node.

3.5 The Path Stack

Like the B^{link}-tree, each process in a B^{list}-tree has its own `path` stack that stores the path it has followed from the root to some leaf. Any operation in the B^{list}-tree starts at the root. Until it finds the appropriate leaf, each process stores the rightmost node it encounters on each level of the tree during it search on its `path` stack (see Section 4.1.1). The stack is later used to find the parent of a node. In an insert operation, when a process is splitting a node, it needs to find and modify the node's parent. An inserting process may also need to find and modify a parent's `high-key` (see Section 4.1.2). Since the `path` stack stores the path

from the root to a node (or leaf), the parent of that node is stored on the top of the `path` stack. Therefore, whenever it is necessary to find the parent of a node, a process gets it from the process's `path` stack. Due to concurrent splits during the insert operations, however, the stored parent may no longer be the parent of a particular node. In this case, the `right`-link pointer of the stored parent can be followed to find the current parent of that node.

3.6 The Retired-Cell Stack

The B^{list}-tree delete algorithm uses a `retired-cell` stack like the lock-free linked-list delete algorithm of Michael [15] (See Section 2.2 to know how Michael's `retired-cell` stack works). Each process has its own `retired-cell` stack, which cannot be viewed or accessed by other concurrent processes. After the deletion of a cell, instead of returning that deleted cell to the `free-list`, a process stores the address of the deleted cell in its `retired-cell` stack. The cell remains in the `retired-cell` stack until the process performs the scan algorithm, which returns the cell to the `free-list` if it is safe to do so.

Unlike Michael's lock-free linked list, if a process wants to return a cell from its `retired-cell` stack to the `free-list` in a B^{list}-tree, it has to return the cell to the `free-list` of the node to which the cell belongs. To get the address of the appropriate node, the B^{list}-tree's `retired-cell` stack stores the node address in addition to the cell address.

Like Michael's algorithm, when the number of cells retired by a process reaches a threshold level (for B^{list}-tree algorithms, the threshold level, R, is $m/5$), the process performs a scan routine that examines the retired cells in the process's `retired-cell` stack, and sees if any hazard pointers of any concurrent processes are pointing to those cells. If no hazard pointers are pointing to a cell, the scan routine retrieves the associated node address, and returns the cell to that node's `free-list`. Otherwise, the cell is returned to the process's `retired-cell` stack.

3.7 The Please-Scan Array

Every B^{list}-tree has a `please-scan` array shared by all processes. The `please-scan` array contains a `scan-bit` for every process. If a process's `scan-bit` is on, that process is requested to perform a scan by some other concurrent process(es).

In Michael's lock-free linked-list algorithms, only the delete processes check the threshold level, and, if necessary, call the scan routine to return the cells retired by these processes. Similarly, in Section 3.6, we saw that delete processes return a deleted cell to the `free-list` by calling the scan routine in B^{list}-trees. Unlike Michael's lock-free linked-list algorithms, which assume that there is only one shared linked list, in B^{list}-trees, each node has its own linked list, which contains a `free-list` of its own. If a process deletes some cells from the same node and stores them in the retired-cells stack, but does not perform a scan to return them in the near future, then the node's `free-list` could become empty. Any update process that is looking for a `free cell` from the list will then have to wait until that delete process returns the retired cells to the `free-list` (see Sections 4.1.2 and 4.1.3). If many processes have to wait for a long time to get a `free cell` from a particular node, there will be a bottleneck. Furthermore, for example, if process P_1 is waiting for a `free cell` that can only be returned by process P_2, process P_2 is waiting for a `free cell` that can only be returned by process P_3, and process P_3 is waiting for a `free cell` that can only be returned by process P_1, then all three processes will be in a deadlock.

To solve this problem, we added the `please-scan` array to our ^{list}-tree. If a process P_1 is waiting for a `free cell` in a node, N, but the free list is empty, the process P_1 checks its associated `scan-bit` immediately in the `please-scan` array. If that `scan-bit` is on, the process P_1 calls the scan routine to return the retired cells to their (the cells') associated free lists. If the scan returns some `free cells` in the `free-list` of the node, N, where the process P_1 is waiting, some waiting processes (perhaps P_1) can get `free cells`.

If the scan performed by process P_1 does not return a `free cell` to the `free-list` of node N, process P_1 needs to request other processes to scan so that they can return some cells to the `free-list` of node N where P_1 is waiting for a cell. To send the request, process P_1 goes through each cell of the `key-list` of node N, and looks for a retired cell. If there is a retired cell, then the process P_1 finds the `thread-id`

(see Section 3.1) of the process that retired the cell to identify which process (say, P_2) has retired this cell. Then, process P_1 turns on process P_2's `scan-bit` in the `please-scan` array. When process P_2 sees that its `scan-bit` is on, it immediately performs a scan and returns some cells to their associated free lists. This action will eventually provide some `free cells` to the processes that are waiting for the `free cells`.

3.8 Benefits of B^{list}-trees

Compared to the B^{link}-tree algorithms and their variants described in Section 2, our B^{list}-tree algorithms have some advantages in terms of potential performance. By storing the tree in the main memory of the parallel machine, using a custom linked-list structure to ensure memory locality while maintaining flexibility, and using lock-free operations in the nodes, our B^{list}-tree will potentially be more efficient than concurrent B^{link}-trees.

3.8.1 Benefits of an In-Memory Structure

Access to an in-memory structure will always be faster than one stored on secondary storage. This speed is necessary for the types of applications we are targeting.

3.8.2 Benefits of the List Structure

In B^{list}-trees, a process does not have to shift the keys in a list-based `key-list` after an update operation, as is required in an array-based `key-list` used in B^{link}-trees. Furthermore, in our B^{list}-tree algorithms, we designed the cells of the `key-list` to be allocated from an array of cells. The benefit of the array of cells implementation is that, when a process accesses one cell in a node, many adjacent cells in the array of that node will also be put in the cache (due to locality of reference since the cells are stored in consecutive memory locations). Hence, searching in the nodes should be fast. Ou B^{list}-tree algorithms are therefore suitable for shared memory parallel machines. Further, for Non Uniform Memory Access (NUMA) machines, with additional effort to localize operations on tree nodes to specific processors, additional locality benefit could be realized at the level of main memory access. In the NUMA architecture, each processor has its own local memory. For a processor, accessing local memory is always faster than accessing non-local memory. By using localized access, NUMA machines can be faster in parallel computations than other parallel machines.

3.8.3 Benefits of Lock-free Concurrency-Control Techniques

All the B-trees and their variants described in Section 2 use lock-based pessimistic concurrency-control techniques. Unlike those algorithms, our B^{list}-tree algorithms use lock-free techniques to handle concurrency. Lock-free concurrency control gives our B^{list}-tree a finer granularity of concurrent access, compared to lock-based B-trees and their variants. In B^{list}-trees, each process accesses and manipulates data in smaller chunks (i.e., a cell in a node), whereas a process in a B^{link}-tree accesses and manipulates data in bigger chunks (i.e., a node in the tree). Moreover, in a B^{link}-tree, a process acquires the locks of nodes one at a time. Any concurrent process that wants to work on the same node has to wait until the process that holds the lock of the node finishes its job, and releases the lock. However, in a B^{list}-tree, more than one process can operate on a single node (also, with restrictions, on a single cell) simultaneously. If the result of an operation of one process affects the results of the operation of other concurrent processes, the other concurrent processes either have to redo the work in that cell or the processes have to go to the beginning of the `key-list` of that node and restart their work within that node, yet all processes can redo their work simultaneously. If the concurrent operations have no effect on each other, all processes can continue with their operations simultaneously without any interruption. In a B^{list}-tree, if N processes want to operate in the same node and each operation takes time t, if there is no interference between them, all processes will finish in time t. In a B^{link}-tree, however, all processes will finish in time $N \times t$ (since only one process can work in the node at a time), even if there is no interference between concurrent processes. Therefore, if there is high data access contention in both trees (i.e., many processes want to concurrently access the same node), the B^{list}-tree

algorithms should take comparatively less running time to complete an operation than the lock-based B-tree algorithms and their variants.

Besides this, lock-free concurrency-control techniques assure that our B^{list}-tree algorithms will be free from convoying [22] and priority inversions [20].

Lock-based concurrency-control techniques are prone to deadlocks [7], but the lock-free concurrency-control techniques are not. To prevent deadlocks, our lock-free B^{list}-tree uses a `please-scan` array where every process has a `scan-bit` that is associated with their process identification number (see Section 3.7). In the B^{list}-tree update algorithms, if a process P_1 needs a free cell, then P_1 scans its own `retired-cell` stack to get a free cell for itself. If the scan cannot provide any free cell for P_1, then P_1 requests other process(es) (say process P_2) that can reclaim a free cell for the process P_1. To send a scan request to process P_2, process P_1 sets the associated `scan-bit` of process P_2 in the `please-scan` array. In a B^{list}-tree, before starting an operation, each process checks their associated `scan-bit` in the `please-scan` array. If the associated `scan-bit` of a process is set, then the process scans its `retired-cell` stack to reclaim free cells.

Without the use of the `please-scan` array in a B^{list}-tree, if process P_1 waits for a cell that can only be reclaimed to the `free-list` by process P_2, and process P_2 is waiting for a cell that can only be reclaimed to the `free-list` by process P_1, then both P_1 and P_2 have to wait forever to get a free cell for themselves. This waiting for free cells will ultimately cause a deadlock in the tree. However, in B^{list}-trees with a `please-scan` array, when both processes P_1 and P_2 are waiting for a free cell, both processes scan their own `retired-cell` stacks for free cells. If the scans cannot release a free cell for themselves, then process P_1 sends a scan request to process P_2 through the `please-scan` array, and vice versa. When both processes see the scan requests, both processes scan their `retired-cell` stacks and reclaim a free cell (if the cell is not in use by other concurrent processes) for the requested process (i.e., P_1 reclaims a free cell for P_2, and P_2 reclaims a free cell for P_1). This free cell reclamation prevents the potential deadlocks in the B^{list}-tree, and makes the B^{list}-tree algorithms to be deadlock-free.

4 Algorithms

4.1 Algorithm Overview and Discussion

Our B^{list}-tree algorithms can perform three major operations: search, insert, and delete on a B^{list}-tree. The search operation looks for a particular key in the B^{list}-tree, the insert operation inserts a new key in the tree if that key is not already in the B^{list}-tree, and the delete operation deletes a desired key from the B^{list}-tree, if it exists in the tree.

Sections 4.1.1, 4.1.2 and 4.1.3 describe the search, insert and delete algorithms for a B^{list}-tree, respectively. Section 4.2 discusses the concurrent correctness of the algorithms.

4.1.1 Search Algorithm

The search algorithm works in two major steps. First, the algorithm finds the leaf in the B^{list}-tree where the desired key should reside. Next, the algorithm looks for the desired key in that leaf.

Step 1: Find the Appropriate Leaf

The search process finds the appropriate leaf in several steps. If the search process searches for the desired key in an empty B^{list}-tree, then the search process terminates right away, and knows that the desired key is not found in the B^{list}-tree. Otherwise, the search process starts looking for the appropriate leaf by looking at the root first, and then, by examining one node at each level, the process propagates to the leaf level. In each level where the node to be examined is a non-leaf node, the search process examines the keys from the key list of the selected node, and tries to find the appropriate child to move to, that is, the root of the sub-tree containing the appropriate leaf and the desired key.

Before starting the key comparisons within a node N, a search process sets one hazard pointer (`hp1`) to point to the `head` of the node's `key-list`, and then sets another hazard pointer (`hp2`) to point to the cell

immediately after `head` (see Sections 2.2 and 3.4 for a description of hazard pointers). The cell pointed to by `hp2` is the leftmost cell in the `key-list`, and contains the smallest key in the `key-list`. (From this point on, we refer to the cell pointed to by `hp1` as the previous cell, and we refer to the key stored in the previous cell as the previous key. Furthermore, we call the cell pointed to by `hp2` the current cell, and the key stored in that current cell as the current key.)

After setting the hazard pointers to their initial locations, the search process first compares the key to be searched for with the `high-key` of the node N in case other concurrent processes have split this node. If the `high-key` of the node N is less than the key to be searched for, a split has occurred, so the process follows the `right` link pointer of N to examine the node, R, that is immediately to the right of N. If the `high-key` of the node N is less than the key being searched for, then the search process compares the key to be searched for with the current key. If the current key is less than the desired key, the search process knows that the desired key might be in any of the following cells of the `key-list`. Therefore, it moves `hp1` and `hp2` to the immediately following pair of cells in the `key-list`. Before moving the hazard pointers, the search process examines whether the previous cell has been marked as deleted by another concurrent process, or if the current cell is not the immediate next cell of the previous cell. Both cases prevent the search process from moving to the cells immediately to the right of the previous cell and the current cell. To avoid such inconsistencies, some concurrency-control techniques are applied during the search operation. Please see Section 4.2 for a description of the concurrency-control techniques used in the search operations.

Each time the search process moves the hazard pointers to the next cells, it compares the desired key with the current key. It keeps on moving the hazard pointers until it finds a current key that is greater than or equal to the desired key. After finding such a key, the search process knows that the `left-child` of the current cell is the root of the sub-tree that contains the leaf for which it is searching. Thus, the search process chooses the `left-child` of the current cell to examine next in the search for the appropriate leaf containing the desired key.

If the search process cannot find any current key in the `key-list` that is greater than the desired key and reaches the end of the `key-list` (i.e., `hp1` points to the cell that contains the largest key in the `key-list`, and `hp2` points to the `tail` of the `key-list`), the search process knows that the `right-child` of the previous cell (pointed to by `hp1`) points to the correct child of this node in which to find the appropriate leaf. Thus, the search process looks in the `right-child` of the previous cell in the search for the appropriate leaf containing the desired key. Before navigating to the selected child, the search process stores the currently examined node in the `path` stack (see Section 3.5) for further use. The search process continues to move from each node to a child node until it finds a leaf node.

Step 2: Find the Desired Key in the Leaf

When the search process finds the appropriate leaf for a desired key, it compares the keys from that leaf's `key-list`, one by one, with the desired key. The initial locations of the hazard pointers and the movement of the hazard pointers follows the same rule as followed in step 1 of the search algorithm.

The search process keeps on moving `hp1` and `hp2`, and comparing the current key with the desired key, until it finds a current key that is greater than or equal to the desired key. If the search process finds a current key that is equal to the desired key, then it has found the key in the leaf, and returns success. Since the keys in the `key-list` in a node of a B^{list}-tree are stored in sorted order, if the search process finds a current key that is greater than the desired key, then it knows that the desired key does not exist in the leaf. In this case, the search process empties the `path` stack and terminates with a failure result. If the search process examines all keys from the `key-list` but cannot find the desired key, then, again, it knows that the key is not in the leaf

4.1.2 Insert Algorithm

The insert algorithm for a B^{list}-tree is the most complicated algorithm, compared to the search and the delete algorithms. An insertion in a B^{list}-tree works in the following steps.

Step 1: Create a Root

Inserting a key in an empty tree is different than inserting a key in a non-empty tree. If an insert process wants to insert a key in a non-empty B^{list}-tree, the insert process does not execute step 1, it goes directly to step 2 to find the appropriate leaf. Otherwise, if the tree is empty, the process creates a new leaf node for the tree. Then, it allocates a cell from that leaf's `free-list`, and copies the new key into the `key` field (see Section 3.1) of that cell. After that, the insert process inserts that cell between the `head` and the `tail` of the leaf's `key-list`. Then, it increases the `key-counter` of the leaf, and updates the `high-key` of the leaf with the new key. In the last step, the insert process uses CAS to assign the newly-created leaf's address to the `root` of the B^{list}-tree. If other processes already have created another root for the B^{list}-tree, the CAS fails, and the insert process goes to step 2. Otherwise, the CAS successfully assigns the new leaf as the root of the B^{list}-tree. Finally, the insert process makes the new root of the tree accessible to other processes by setting the `finally-linked-in-tree` bit (see Section 3.3) of that leaf. From this point on, the new root is available to other concurrent processes.

Step 2: Find the Appropriate Leaf

In the case of inserting a key into a non-empty tree, the insert process needs to find the appropriate leaf for the key insertion. The search for the appropriate leaf in an insert algorithm in a B^{list}-tree is identical to the search for the appropriate leaf in the search algorithm. See step 1 of the search algorithm in Section 4.1.1 for details.

Step 3: Allocate a Cell for the New Key

The insert process needs to allocate a cell from the `free-list` of the leaf to store the new key. Before it allocates the new cell, it checks the status of the `split-bit` (see Section 3.3) of that leaf. If the `split-bit` is on, another concurrent insertion process is splitting the same leaf. Therefore, this insert process has to busy wait until the other inserter completes the split process and turns off the leaf's `split-bit`.

Once the `split-bit` is off, the insert process checks whether the leaf is still the appropriate leaf in which to insert the new key after the split operation (see Section 4.2 for a description of how the insert process handles concurrent splits while allocating a new cell for the new key). Once the insert process is in the correct node, it looks for an empty cell in the `free-list` of that leaf's `key-list`. If there is no cell in the `free-list`, the insert process checks the status of the `split-bit`, and the `finally-linked-bit` (see Section 3.3) of that leaf. It has to busy wait until the `split-bit` is off, or the `finally-linked-bit` is on. If both conditions are satisfied, but there are no cells available in the `free-list`, the insert process has to communicate with all concurrent processes, including itself, to try to get a free cell reclaimed in the `free-list`. Section 3.7 describes how a process communicates with other processes using the `please-scan` array.

The insert process keeps on requesting a free cell from all concurrent processes until it gets one. Once it finds a free cell in the `free-list`, it stores the new key in that cell, and goes to step 4 to search for the correct position to insert the cell containing the new key in the leaf's `key-list`.

Step 4: The Search for the New Key's Position

The search for the new key's position in a particular leaf is similar to searching for the desired key in a leaf in the search algorithm. While comparing the keys, if the insert process finds the key in the `key-list`, it knows that the key already exists in the B^{list}-tree, and cannot be inserted in the tree again. So, the insert process returns the allocated cell to the `free-list`, empties the `path` stack that was filled during the search for the appropriate leaf, and then terminates. Otherwise, the insert process goes to step 5 to insert the allocated cell containing the new key between the cells pointed to by `hp1` and `hp2`.

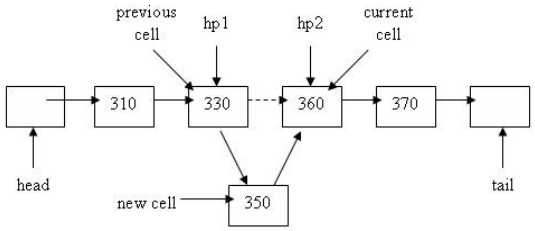

Figure 8: CAS swings the `next` field of the previous cell to point the new cell.

Step 5: Try to Insert the Key in the Leaf

From this point on, we refer to the cell pointed to by `hp1` as the *previous* cell, the cell pointed to by `hp2` as the *current* cell, and the cell containing the new key as the *new cell*. To insert the new cell in the key list of the leaf, the insert process sets the `next` pointer (see Section 3.1) of the new cell to point to the current cell. Then, the insert process uses CAS to update the `next` field of the previous cell to point to the new cell. Here, CAS takes the address of the `next` field of the previous cell as the shared variable, the address of the current cell as the old value, and the address of the new cell as the new value. If the current cell is still the cell immediately after the previous cell, the CAS operation succeeds and sets the `next` pointer of the previous cell to point to the new cell, thereby linking the new cell into the list (Figure 8). If the CAS fails, then the insert process has to try again and has to repeat steps 4 and 5 until it succeeds in inserting the new cell in the `key-list`.

After the insert process inserts the new cell in the `key-list` of the leaf, it increments the `key-counter` of the leaf. If the `key-counter` of the leaf is incremented to m (the maximum number of children that a node of a B^{list}-tree can have), then the insert process knows that the leaf is overflowing, and must split the leaf. Therefore, the process executes step 6. Otherwise, the process skips steps 6–9, and goes to step 10 to fix the `high-key` of the parent node, if the leaf's `high-key` became greater than the parent's `high-key` because of the insertion. Otherwise, the insert process empties the `path` stack that was created during the search for the appropriate leaf, and terminates.

Step 6: Split the Leaf

To split a leaf (or node), the insert process has to set the `split-bit` of that leaf. Before it tries to turn the `split-bit` on, it has to check whether the leaf is accessible to all concurrent processes by checking the `finally-linked-in-tree` bit of the leaf (see Section 3.3). Even though an insert or delete process is allowed to insert or delete a key in a leaf (or a node) that is not linked into the tree yet, it is not allowed to split that leaf until the leaf is linked into the tree. If the leaf is not yet linked into the tree, the process needs to busy wait until the leaf is completely linked into the tree and accessible to any concurrent process.

Once the leaf's `finally-linked-in-tree` bit is set, the insert process again checks the `key-counter` of the leaf. This check is necessary because, when the insert process was busy waiting for the leaf to be linked into the tree, the `key-counter` might have gone below m due to some concurrent key deletion(s) or node split(s). In that case, there is no longer an overflow in the leaf, and, hence, there is no need to split the leaf. In this case, the insert process goes to step 10 to fix the `high-key` of this leaf's parent, if necessary.

The insert process uses CAS to safely set the `split-bit` of the leaf. The CAS uses the `split-bit` field as the shared variable, 0 as the old value, and 1 as the new value of the `split-bit`. If the CAS fails, the insert process busy waits until the CAS succeeds. When the busy wait is over, the insert process again checks the `key-counter` of the leaf, and tries to set the `split-bit` again, if the leaf is still overflowing.

The insert process creates a new leaf for the B^{list}-tree after it sets the `split-bit`. After creating the new leaf, the insert process finds the *middle cell* containing the *middle key* (the $\lceil m/2 \rceil^{th}$ key) from the leaf's `key-list`. Then, it copies the cells that contain keys larger than the middle key from the original leaf's

`key-list` to the new leaf's `key-list`. Next, the insert process updates the `key-counter` of the new leaf with the number of cells copied to the new leaf, then updates the `high-key` of the new leaf with the `high-key` of the original leaf, and, finally, sets the new leaf's `right` pointer to point to the leaf currently pointed to by the original leaf's `right` pointer.

After updating the fields of the new leaf, the insert process changes the necessary fields of the original leaf. First, it sets the `right` pointer of the original leaf to point to the new leaf. Then, it updates the `high-key` of the original leaf to be its middle key. Next, it decreases the `key-counter` of the original node to $\lceil m/2 \rceil$. Finally, the insert process reclaims the cells that were copied to the new leaf adding them to the original leaf's `free-list` for reuse.

Step 7: Find the Parent

When the insert process completes the split operation, it needs to find the parent of the original leaf to insert a copy of the original leaf's middle key into its parent. Furthermore, a child pointer to the new leaf from the parent is also required. To find the parent of the original leaf, the insert process gets the topmost node from the `path` stack (see Section 3.5) that was created during the search for the appropriate leaf in step 2.

When the insert process has a non-empty `path` stack, it gets the topmost node of the `path` stack, which should be the parent of the original leaf. However, due to the possibility of concurrent splits on the parent, the parent node popped off the `path` stack might no longer be the parent of the original leaf. In that case, the insert process finds a node to the right of the node stored on the `path` stack as the parent, a node that has a `high-key` greater than or equal to the `high-key` in the original leaf (see Section 4.2 for details). After the split process finds the correct parent for the original leaf, it goes to step 9 for the middle key insertion into the parent.

If the insert process split the root of the B^{list}-tree in step 6, the `path` stack is supposed to be empty (since, the root has no parent), and there is no way to return any node from the empty `path` stack. In that case, the insert process goes to step 8 to add an extra level to the tree.

Concurrent operations can also cause an empty `path` stack, even if the insert process did not split the root of the B^{list}-tree in step 6. This case can happen if, during the operations in step 2, the original leaf was the root of the tree, but due to other concurrent operations, the tree has had a new root added (and has added levels between the current root and the original leaf). The only way to get the parent of the original leaf is to now re-fill the `path` stack with the visited nodes in the path from the root to the original leaf. In this case, the insert process re-fills the `path` stack in the same way that it filled the `path` stack in step 2. After completing the re-filling of the `path` stack, the insert process gets the address of the parent node from the topmost node stored in the `path` stack, and goes to step 9 to insert the middle key in the parent node.

Step 8: Make the Tree Taller

In step 6 of the insert algorithm, if an insert process splits the root of the tree, it needs to add an extra level to the tree by creating a new root for the tree. This step is similar to step 1 (create a new root) of this algorithm. For making the tree taller, by creating a root for a non-empty tree, the insert process copies the middle key of the original node (or leaf) in the allocated new cell from the `key-list` of the new root. Then, the `left-child` pointer of that new cell is set to point to the original node (or leaf), and the `right-child` pointer of the same cell is set to point to the new node (or leaf) that was created by the split operation in step 6. Additionally, the insert process updates the `high-key` with the new node's `high-key`, then sets the `finally-linked-in-tree` bit (see Section 3.3) of the new node, and clears the `split-bit` of the original node. Finally, it sets the root pointer of the B^{list}-tree to point to the new root, followed by setting the `finally-linked-in-tree` bit of the new root.

Step 9: Insert the Middle Key into the Parent

Once the insert process finds the parent of the original node (or leaf), it allocates a new cell from the `free-list` of the parent, similar to the way it allocates a new cell from a leaf's `free-list` in step 3. The

only difference here is that after allocating the new cell from the parent's `free-list`, besides copying the middle key to that cell, it copies the address of the original leaf to the `left-child` pointer of the new cell, and the address of the new leaf to the `right-child` pointer of the new cell.

The next step is to search for the middle key in the parent. This search is identical to step 4 of the insert algorithm. If the insert process does not find the middle key in the parent, it tries to insert the new cell between the cells pointed to by its hazard pointers, `hp1` (the previous cell) and `hp2` (the current cell). Insertion of the new cell between the previous and the current cell is identical to step 5.

During a middle key insertion, the insert process needs to fix the child pointers of the previous and current cells after the insertion of the new cell in the `key-list`. If the middle key is not the largest key in the parent node, after the insertion, the `left-child` pointer of the current cell points to the original leaf. The insert process changes the current cell's `left-child` pointer to point to the new leaf. If the middle key is the largest key in the parent, and the new cell becomes the rightmost cell in the `key-list`, then the insert process sets the `high-key` of the parent to be the `high-key` of the new leaf.

If this insertion creates an overflow in the parent, then the parent needs to be split. In that case, the insert process repeats step 6 to 9. A node split is very similar to a leaf split as described in step 6. The only difference is that when the insert process reclaims the copied cells to the original node's `free-list`, along with the copied cells it also reclaims the middle cell to the `free-list`, since B^{list}-trees store the keys only in their leaves. Furthermore, the `key-counter` is decremented to $\lceil m/2 \rceil - 1$, instead of decrementing it to $\lceil m/2 \rceil$.

Step 10: Fix the Parent's High-key

After the key is inserted into a leaf or a non-leaf node, if the `high-key` of that node becomes greater than its parent's `high-key`, then the insert process updates the parent's `high-key` with that node's `high-key`. If, the parent's `high-key` becomes greater than the `high-key` of the parent's parent, then the `high-key` of the parent's parent is updated with the `high-key` of the parent. These `high-key` updates continued until the insertion process finds a node whose `high-key` is less than or equal to its parent's `high-key`, or the node has no parent (i.e., the node is the root of the tree). Once the parent's `high-key` is fixed, the insert process empties the `path` stack and terminates.

4.1.3 Delete Algorithm

The delete algorithm uses multiple steps to delete a key from a B^{list}-tree. Those steps are described bellow:

Step 1: Find the Appropriate Leaf

Finding the appropriate leaf in a deletion is identical to finding the appropriate leaf in a search. See step 1 in the search algorithm (Section 4.1.1) for details.

Step 2: Search for the Desired Key

The search for the desired key in an insertion and the search for the desired key in a delete algorithm are identical. See step 4 of the insert algorithm (Section 4.1.2) for details.

Step 3: Delete the Requested Key from the Leaf

If the delete process does not find the desired key in the appropriate leaf, the process knows that the key does not exist in the B^{list}-tree, so the desired key cannot be deleted, and the deletion process terminates. Before the termination of the deletion process, the `path` stack is emptied.

When the delete process finds the desired key in step 3, the hazard pointer `hp2` of the delete process points to the cell that contains the desired key. Let us call this cell the `current-cell`. The other hazard pointer, `hp1`, points to the cell immediately to the left of the current cell. Let us call this cell the `previous-cell`. Before deleting the `current-cell`, the delete process *marks* (see Section 2.2) the `next`

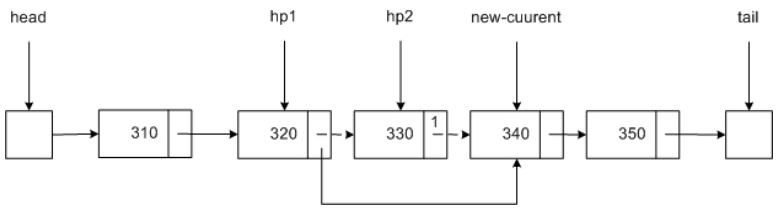

Figure 9: The CAS operation swings the `next` field of the `previous-cell` (pointed to by `hp1`) from the `current-cell` (pointed to by `hp2`) to the `new-current` cell.

field of the `current-cell` to be deleted. Marking a cell protects that cell from manipulated by other concurrent process(es). Before marking a cell, the delete process has to deal with some concurrency-control issues that are described later in Section 4.2.

After marking the cell to be deleted, the delete process uses CAS to update the `next` pointer of the `previous-cell` to point to the cell, `new-current`, immediately following the `current-cell`. The CAS takes the `next` field of the `previous-cell` as the shared variable, the address of the `current-cell` as the old value, and the address of `new-current` as the new value to be stored in the `next` field of the `previous-cell`. If the CAS succeeds, the `next` field of the `previous-cell` points to the `new-current` cell (see Figure 9). Then, the delete process decrements the key counter of the leaf, and stores the process identification number of the deletion process in the `current-cell`'s `thread-id` field. This identification number is used later to reclaim the `current-cell` to the `free-list` of the leaf. In case of an unsuccessful CAS (due to some concurrent updates), the delete process unmarks the `current-cell`, and repeats steps 2 and 3 until the CAS operation succeeds.

Step 4: Retire the Deleted Cell

Since a B^{list}-tree does not use locks for any operation, returning a deleted cell to the `free-list` of a leaf after a deletion is unsafe in a concurrent environment. For example, if a deleter returns a deleted cell to the `free-list` while other concurrent processes are concurrently working on that cell, it may create incorrect results (see Section 2.2). Therefore, instead of returning a cell directly to the `free-list` of the leaf, our deletion algorithm marks the deleted cell as retired, and stores the address of the cell, along with the address of the associated leaf, in the `retired-cell` stack (see Section 3.6). Each process in a B^{list}-tree has its own `retired-cell` stack from which only that process can later reclaim the cells to the `free-list` of the cell's corresponding leaf.

Every B^{list}-tree is assigned a threshold level R (as described by Michael [15]) that is usually related to m. If the size of a process's `retired-cell` stack reaches the given threshold, R, the delete process calls for a scan and goes to the next step. Otherwise, the delete process empties the `path` stack and terminates.

Step 5: Scan the Retired-cell Stack

The scan routine returns retired cells to their free lists, if they are not currently pointed to by the hazard pointers of any process. First, the scan routine clears the `scan-bit` of the deletion process in the `please-scan` array (see Section 3.7). In the next step, it copies the global list of cell addresses, along with their corresponding leaf addresses, of all cells that are currently pointed to by the hazard pointers of any process. These copies are stored in a local list, `p-list`. Then, the scan routine sorts the `p-list` according the cell addresses. Next, the scan routine empties the `retired-cell` stack and stores all the cells, along with their associated leaves, in a temporary stack. The scan routine then compares the cells from the `p-list` with the cells stored in the temporary stack. Any cells that appear in the `p-list` are being used by some process, so they are returned to the `retired-cell` stack. Any cells that do not appear in the `p-list` are not being used by other processes, and can be safely returned to their free lists as described in step 6.

Step 6: Reclaim the Retired Cells

In this step, the delete process uses CAS to return each safe cell to its corresponding leaf's `free-list`. First, the delete algorithm links the cell to be reclaimed (`new-free`) with the `free-list`. Then, CAS takes the address of the `free-head` (see Section 3.2) of the `key-list` as the shared variable, a copy of the `free-head` (`copy-free`) as the old value, and the `new-free` as the new value. If no concurrent process(es) changes the `free-head`, then CAS swings the `free-head` pointer to point to `new-free`. If the CAS fails, the delete process repeats this step until the CAS succeeds.

4.2 Concurrency Correctness of the Algorithms

Unlike the B^{link}-tree algorithms, the B^{list}-tree algorithms do not lock a node for an update or a split operation. Therefore, when a process performs an operation in a node, other concurrent processes can search, insert, or delete some keys in the same node, or can even split the entire node into two nodes. To avoid any inconsistency, our B^{list}-tree algorithms need to handle these concurrent operations without locking the node. In the following subsections, we describe where and how we dealt with the concurrency issues in our B^{list}-tree algorithms.

Consider the three major types of operations in our B^{list}-tree. They are the search operations, where the processes just read, but do not write, any data in the tree; the update operations (insert and delete), where the processes write data in the tree, and finally, the split operations, where the processes split a node into two different nodes. We ensure that none of these types of operations interfere with each other while operating concurrently in a B^{list}-tree.

4.2.1 Concurrency Control in Search Operations

Selecting the correct child to follow to get to the appropriate leaf, and searching for the desired key in the leaf are the key tasks in any search operation in a B^{list}-tree. Since the search tasks do not change any contents of the B^{list}-tree, concurrent search operations in a node (both leaf and non-leaf) do not create conflicts with each other. Conflicts can, however, arise when one or more concurrent update and/or split operations are performed simultaneously with a search operation.

Conflicts with Concurrent Update Operations

When a searcher compares a desired key with the keys of a node's key list, concurrency control is required if a concurrent inserter is inserting a new cell between the cells pointed to by the searcher's `hp1` and `hp2`, or a concurrent deleter is deleting the cell pointed to by searcher's `hp1` or `hp2`.

In case of a concurrent insert operation, the `hp1`s of both searcher and inserter point to the same cell and `hp2`s of both searcher and inserter point to the same cell. If the key to be inserted is equal to the key to be searched for, then without concurrency control (when a concurrent inserter completes the insertion during the search operation), the search operation would see that its hazard pointer `hp2` points to a cell that contains a key that is greater than the key to be searched for. Even though the key to be searched for is now in the `key-list`, the searcher would conclude that the desired key is not in the `key-list` of the node. To avoid this inconsistency, before comparing the desired key with the key in the cell pointed to by `hp2`, and before moving the `hp1` and `hp2` pointers one cell ahead in the `key-list`, the search operation always tests whether the cell pointed to by `hp2` is still the next cell immediately to the right of the cell pointed to by `hp1` in the `key-list`. If the test fails, the search operation must re-assign `hp2` to point to the cell immediately following the cell pointed to by `hp1`, and continue the search for the desired key.

Similarly, if a concurrent delete operation tried to delete any of the cells pointed to by the searcher's hazard pointers `hp1` or `hp2`, the searcher needs to handle the potential inconsistency caused by the concurrent deletion. To handle these situations, before comparing each key in the `key-list`, and before moving to the next cell, the searcher checks whether the `next` fields of the cell pointed to by `hp1` or `hp2` are marked as deleted by the deleter. If the cell pointed to by the searcher's `hp1` is marked as deleted, then the `next` field of that cell no longer points to a valid cell address, and following the `next` pointer will generate an

invalid operation for the search operation. To avoid the invalid operation, the search operation moves back to the beginning of the `key-list`, and starts searching for the desired key again, from the beginning of the `key-list`. If the search operation sees that the cell pointed to by `hp2` is marked as deleted, it checks whether the concurrent deleter has completely removed the cell pointed to by the searcher's `hp2` from the `key-list`. If that cell is removed from the key-list, the search operation re-assigns its hazard pointer `hp2` to the immediate next cell of the cell pointed to by `hp1`. Otherwise, the search operation moves back to the beginning of the `key-list` and starts search for the desired key again.

Conflicts with Concurrent Split Operations

Handling conflicts between a search operation and a concurrent split operation is more difficult than handling conflicts between a search and a concurrent update operation. This difficulty arises because a search operation can start searching for a desired key in the middle of an ongoing split operation. Moreover, a search process allows a split operation to split the node while the process searches for the desired key. While searching for a key in a node, if the node gets split by a concurrent split process, and the desired key no longer remains in the examined node, then the search operation would conclude that the desired key does not exist in the tree, which is incorrect. The situation gets worse when a cell that is currently being examined by the search process now belongs to the `free-list` of the node due to the split operation (see step 6 of the insert algorithm in Section 4.1.2).

A search operation handles the concurrency issues with split operations in different ways. A search operation compares the desired key with the `high-key` of a node before it starts looking for the key to be searched for in that node, and before comparing each key of the `key-list` with the key to be searched for. As a result of a split operation, if the desired key now resides in the node created by the split operation, then the `high-key` of the original node will be less than the desired key. Therefore, if the search operation finds that the desired key is greater than the `high-key` of the original node (the node it is now looking for the desired key in), it must move right to the immediately following node to search for the desired key. In a split operation (described in Section 4.1.2), as we saw, a splitter copies half the keys from the original node's `key-list` to the new node's `key-list`, then it links the original node to the new node, and finally, it changes the `high-key` of the original node. Therefore, once a concurrent split operation changes the `high-key` of the original node, it has already created the new node, has copied the keys from the original node to the new node, and has linked the original node to the new node. Furthermore, the split operation moves the right-half of the original node to the `free-list` after it changes the `high-key`. Thus, it is safe for the search operation to go to the new node after noticing the change in the `high-key`, and start searching for the desired key in that node.

If the split operation changes the `high-key` of the node after the search operation compares the desired key with the `high-key`, that does not create a problem between the searcher and the splitter because, before comparing the next key in the `key-list` with the desired key, the searcher will do the same comparison again, and will be able to detect the change in the `high-key` of the original node. If the search process is looking at the rightmost key in the `key-list` of the original node, and the concurrent split operation changes the `high-key` after the comparison, that also does not affect the search operation (even though there is no next turn to compare the desired key with the `high-key` of the original node). If the last key is the search key, this key has already been copied to the new node, which is now linked to the tree. Thus, if the search operation correctly concludes that the desired key exists in the tree. Conversely, if the last key is not the desired key being searched for, then its copy in the new node is also the last key in the new node's `key-list` and the desired key does not exist in the tree. In that case, the search process knows that the key being searched for does not exist in the tree, which is again a correct conclusion.

4.2.2 Concurrency Control in Update Operations

There are two types of update operations in a B^{list}-tree. The first is the insert operation, where an insert process inserts a new cell containing the key to be inserted in the `key-list` of a node or a leaf. The second

is the delete operation, where a delete process deletes the cell containing the key to be deleted from the `key-list` of a leaf.

Conflicts with Other Concurrent Update Operations

The most common conflicts that update operations face are conflicts with other concurrent update operations. There can be conflicts between two concurrent insert operations, or between two concurrent delete operations, or between one insert operation and one delete operation. Use of CAS (see Section 1) in the B^{list}-trees prevents these conflicts between more than one concurrent update operation.

Handling conflicts between two concurrent insert operation is straightforward. When two insert operations want to insert two new cells, each containing the same `key`[2], in the same position of the `key-list` of a node. Both inserters have their `hp1` pointing to the same cell (this cell will be referred to as `previous-cell`) and both inserters have their `hp2` pointing to the same cell (this cell will be referred to as `following-cell`), respectively. Before swinging the `next` pointer of `previous-cell` from `following-cell` to the cell to be inserted, both inserters use CAS to check whether `following-cell` still immediately follows `previous-cell` in the `key-list`. Here, if both inserters want to swing the `next` pointer of `previous-cell` at the same time, because of the atomicity property of CAS, one will succeed, and the other will fail. The inserter with a successful CAS operation will complete the insertion of the cell to be inserted, and the other unsuccessful concurrent insert operations have to re-do the insert operation.

In the case of concurrent delete operations, two different types of conflicts can occur. In the first type, both delete operations want to delete the same cell, and in the other case, two delete operations want to delete two consecutive cells.

Let us consider the situation where both delete operations want to delete the same cell. Both of their `hp1` pointers point to the same cell (this cell will be referred to as `previous-cell`), and both of their `hp2` pointers point to the same cell (the cell to be deleted, this cell will be referred to as `current-cell`). In the first step of the deletion operation, both delete operations will try to mark `current-cell` as being deleted. A CAS operation is used to add 1 to the `next` field of the `current-cell` to mark that cell (see Section 2.2). Use of CAS operation prevents multiple marking operations (adding 1 to the `next` field) that might change the least significant bit of the `next` field back to 0. Once the cell is marked by one of the concurrent deleter, both deleter processes will try to swing the `next` pointer of `previous-cell` from pointing to `current-cell` to point to the cell immediately following `current-cell`, using a CAS operation. Here also, because of the nature of the CAS primitive, only one delete operation will succeed in swinging the `previous-cell->next` pointer to point to the cell immediately following `current-cell`, and the other will fail. Whichever deleter fails will have to re-do the deletion operation.

The second type of conflict between two concurrent delete operations will occur if they want to delete two consecutive cells from the key list of the same node (Figure 10). From Figure 10, `current-cell-1` and `previous-cell-2` point to the same cell, and `following-cell-1` and `current-cell-2` point to the same cell. Now, if the second deleter can complete the deletion of `current-cell-2` before the first deleter marks `current-cell-1` for deletion, there will be no conflicts among the concurrent delete operations since the deleted cell from the second deleter is not involved in the first deleter's delete operation. However, once the first deleter marks `current-cell-1` (and assuming the second deleter did not delete `current-cell-2` from the `key-list`, yet), the `next` pointer of `current-cell-1` no longer points to `following-cell-1`. For the second deleter, then, `previous-cell-2->next` no longer points to `current-cell-2`. Therefore, the CAS operation performed by the second deleter to remove `current-cell-2` from the key list will fail, and the second deleter will unmark `current-cell-2` and re-do the delete operation.

In addition to the concurrency-handling techniques discussed above, using hazard pointers (see Section 3.4) avoids conflicts related to memory reuse between concurrent operations in a B^{list}-tree. When a delete operation deletes a cell from the `key-list` of a leaf, instead of reclaiming that cell to the `free-list` immediately, the delete process stores the address of the cell in its `retired-cell` stack until no hazard

[2]The inserters' keys do not have to be the same to cause a conflict. The keys just have to both fall between the same two keys already in the tree.

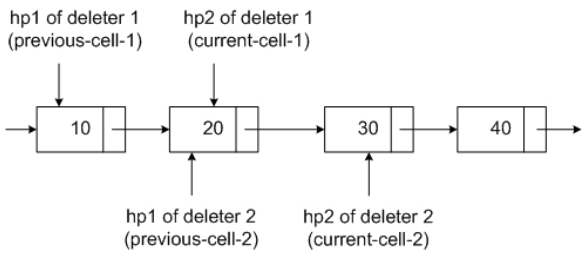

Figure 10: Two concurrent deleter operations want to delete two consecutive cells.

pointers of any process point to that cell (see step 4 of the delete algorithm in Section 4.1.3). Since the address of a deleted cell is kept in the `retired-cell` stack, other concurrent operations can finish their jobs on that cell without facing any conflicts. If there are more than two concurrent update operations in the same node that have potential conflicts, the conflicts are handled in the same way as they are handled between two concurrent update operations in the same node.

Conflicts with Concurrent Split Operations

The B^{list} algorithms do not have any concurrency-control techniques for concurrent update and split since it is impossible to start a split operation in a node (or in a leaf) when there are ongoing concurrent insert or delete operations.

Every update process in a node of a B^{list}-tree, whether an insert or delete process, gets a cell from the node's `free-list` before it starts the update operation. After inserting the key to be inserted along with the allocated cell, the insert process increments the `key-counter` of the node (or leaf). An insert process calls a split operation if and only if the `key-counter` is incremented to m, and there are no cell in the `free-list` of the node. If more than one concurrent insert operations are happening in the same node so that there are no free cells left in the `free-list`, and other update processes are waiting to get a free cell, none of the ongoing insert processes can request a split operation until each of them succeeds in inserting their new key, and the `key-counter` incremented to m. Once the last successful insert process increments the `key-counter`, and the `key-counter` hits m, only that insert process will invoke a split operation on that node. Therefore, a split operation cannot start when there is another concurrent insert is going on in the same node.

Similarly, if there is a delete process deleting a key from a leaf, no concurrent inserter can perform a split operation. Like the insert processes, a delete process also holds a free cell from the `free-list` before it starts a deletion operation. Suppose a delete process wants to delete a key (and its cell) from a node, and currently the `key-counter` of that node is $m-1$. Since the `key-counter` is $m-1$, we can conclude that there is only one free cell left in the `free-list` (to be allocated for the delete operation), and no concurrent inserter is trying to insert a key in the `key-list`. If there are any concurrent inserters in the same leaf, all of them have already inserted their keys in the `key-list`, and also have finished incrementing the `key-counter` of the leaf. When the delete process grabs the last cell from the `free-list`, the `free-list` becomes empty. Hence, if any update process wants to insert or delete any key in the same node, it has to wait until this deleter process returns that cell to the `free-list`. The deleter process will return the cell only after it decrements the `key-counter` of the node. Whether the deletion takes place or not, when the delete process returns the allocated cell to the free cell in the `free-list`, the `key-counter` of the leaf will be not more than $m-1$. As a result, no inserter will perform a split operation until the deletion is finished. As it is impossible to have concurrent delete and split operations in the same leaf, there is no need to design concurrency-control techniques for concurrent split and delete operations.

4.2.3 Concurrency Control in Split Operations

The B^{list} algorithms do not need any concurrency-control techniques for a split operation, since it gets the highest preference among all operations in a B^{list}-tree, with respect to concurrency. All update and split operations in a node have to wait until an active split operation splits the node completely (see the insert and delete algorithms in Section 4.1.2 and Section 4.1.3, respectively, for details). A search operation can be done concurrently with a split operation, however, since the search operations only read data from the nodes, they do not create any conflict with the split operations. Techniques for handling the concurrency effects of a split operation on search operations were described in Section 4.2.1, and on update operations were described in Section 4.2.2.

5 Assessment

5.1 Experimental Setup

To assess the implementation of our B^{list}-tree, we compared its performance with Lehman and Yao's B^{link}-trees [12]. The B^{list}-tree algorithms are designed for in-memory applications that run efficiently on shared memory machines including Non-Uniform Memory Access-time (NUMA) machines, rather than for the traditional disk-based applications for which Lehman and Yao's B^{link}-tree algorithms were originally designed.

To achieve a fair comparison between B^{link} and B^{list}-trees, we made some modifications to the B^{link}-trees. We implemented an in-memory version of Lehman and Yao's B^{link}-tree algorithms. In the B^{link}-tree implementation, the keys in the nodes are stored in an array, whereas in our lock-free B^{list}-tree the keys in each node are stored in a *localized* linked list. Furthermore, in B^{link}-trees, the keys residing in a node (both leaf and non-leaf) and the child pointers pointing to the children of a node are implemented separately as an array of keys and an array of child pointers, respectively. The child pointers and their associated key separators are distinguished using the array indexes (e.g., `key[0]` separates `child[0]` and `child[1]`). However, in our B^{list}-trees, each key and its associated child pointers are stored in a `cell` object that is a part of the localized linked list of a node (see Section 3). The complex node structure of B^{list}-trees makes the nodes of the trees noticeably larger than the nodes in B^{link}-trees. Because of the larger size and the complex structure of the nodes, the B^{list}-tree algorithms take significantly higher time to access a key in a node compared to the key access time in a node of the B^{link}-tree algorithms.

In B-trees and their variants, major operations like searches, inserts and deletes take place in a node (or a leaf). To more directly compare the lock-based and lock-free performances within a node, we also implemented a variant of Lehman and Yao's B^{link}-tree in which the keys in a node (both leaf and non-leaf) are stored in a localized linked-list instead of being stored in an array. To distinguish the two different variants of Lehman and Yao's B^{link}-tree, we refer to the B^{link}-tree using the array-based node structure as the *B^{link}-array*, and to the B^{link}-tree using the linked-list-based node structure as the *B^{link}-linked-list*. The B^{list}-tree will, of course, be referred to by its own name.

5.2 Test-bed Environment

We did a complete set of experiments using the Sun Fire x4600 shared memory machines available in the Department of Computer Science at the University of Manitoba. Each test-bed machine, *Helium-01* through *Helium-05*, has eight processors with dual cores so that, in total, the machine can work with 16 different threads concurrently. Due to the memory architecture (where each processor has its own DIMMS) the Sun Fire has NUMA behavior but is not optimized for this. In addition to the Sun Fire machines, we also conducted some of our experiments on an SGI origin 3000 SMP machine, *helios*, provided by Westgrid [21]. Helios has 32 MIPS R14000 CPUs and offers highly optimized cache-coherent (CC-NUMA) memory system.

5.3 Programming Language

To implement the two variants of B^{link}-tree (the B^{link}-array and the B^{link}-linked list) and the B^{list}-tree, we used C++ as our programming language. Additionally, we used the POSIX threads (pthreads) libraries to create multiple simultaneous threads so that our program can concurrently run on the different processors of the Sun and SGI machines.

5.4 Parameters Varied

In our experiments, we recorded the running time of the B^{link}-array, the B^{link}-linked-list, and the B^{list}-tree algorithms for different mixes of operations (search, insert, and delete), different values of m, different key ranges, and different numbers of concurrent processes[3].

5.4.1 Different Operation Types

To begin each experiment on the B^{link}-array, the B^{link}-linked-list, and the B^{list}-tree, we populated the tree with keys inserted by one million uniformly distributed random sequential insert operations. Then, over multiple identical runs, we measured the average execution time for one million concurrent operations consisting of different proportions of search, insert and delete operations. We divided our experiments into five categories according to the ratio of the search, insert, and the delete operations. The first experiment considered only concurrent search operations. In the second, third, and fourth experiments, we included concurrent insert operations along with the search operations. In those three experiments, we decreased the number of the concurrent search operations, and increased the number of concurrent insert operations gradually to see the effect of concurrent insertions in the trees. In the fifth experiment, we examined the running time for equal numbers of concurrent search, insert, and delete operations in all types of trees.

5.4.2 Different Values for m

For each of our five experiments, we also assessed the performance for different values of m for both B^{link}-trees and the B^{list}-tree. In any B-tree variant, using a smaller m, like 100, creates a taller tree than using a larger m, like 1000, assuming the number of keys in the tree is the same. If the tree is taller, then the algorithms should take more time to find the correct leaf to perform the search, insert and delete operations. Conversely, in a shorter tree, with larger m, the algorithms should take more time to find the correct place within a leaf (or node) for the various operations. Moreover, a larger m might introduce more data contention in the same node. To see the effect of the value of m in the trees, we experimented with 100, 200, 500, 700, and 1000 as the values of m.

5.4.3 Different Key Ranges

The key range affects the number of nodes in the initial tree constructed by the serial insertions and also by any concurrent update operations performed after the initial tree is constructed. If the key range is small compared to the number of serial insertions, for example, a key range of 1–250,000 and 1,000,000 serial insertions, then the initial tree will likely contain nearly all the keys in the range. Since the range is 1–250,000, the tree will never be very large because there are only 250,000 keys. Furthermore, concurrent insertions performed after initial construction will fail because the keys are all already in the tree, so the insertions degenerate to searches. Therefore, a mix of searches and insertions with no deletions will be similar to 100% searches except that the insertions allocate a free cell in the appropriate leaf, whereas searches do not. Most concurrent deletions performed after the initial construction will succeed because all keys are in the tree initially. Of course, as the number of deletions increases, more insertions will succeed.

Alternatively, if the key range is large compared to the number of serial insertions, for example, a key range of 1–10,000,000 and 1,000,000 serial insertions, then the number of keys in the tree will be close to the

[3]Although we used pthreads in our implementation, we will continue to use the term *process*.

number of serial insertions. Furthermore, concurrent insertions after the initial construction are more likely to succeed and concurrent deletions are more likely to fail.

To assess the effect of key range on algorithm performance, we used the following different key ranges: 1–250,000, 1–1,000,000, 1–2,000,000, and 1–10,000,000. In all cases, we did 1,000,000 initial serial insertions to populate the tree and then 1,000,000 concurrent operations with mixes, as described earlier.

5.4.4 Different Numbers of Processes

In addition to assessing the effect of operation mix, m and the key range on the algorithms, we also measured the running time of one million concurrent operations on each tree performed by different numbers of processes. As the number of concurrent processes increases, chances of conflicts also increase. Since the one million concurrent operations are divided among the processes, the number of operations per process decreases as the number of processes increases. Consequently, the overall running time is expected to decrease, subject to concurrency overhead.

To determine the trade-off between the number of processes and the performance gain, we ran all of our experiments for 1, 2, 4, 6, 8, 12, 14, and 16 processes on the Sun machines, and for 1, 2, 4, 8, 16 and 32 processes on the SGI machines. These experiments used all values of m, all key ranges, and all concurrent operation mixes.

5.4.5 Memory Affinity

Instead of sequentially building the tree, we also built all three types of trees using eight concurrent processes. Thus, each process should have built its own part of the tree in its associated memory. We expected that, during the concurrent operations, accessing the leaves (or nodes) in each process's associated memory would be less time consuming than accessing nodes in another process's associated memory. However, the experimental results did not show any significant effect of exploiting memory affinity. This lack of effect was likely due to efficient caching because of the design for locality in our algorithm. Therefore, we decided to simply build the tree with a single process running on a single core for the reported experiments.

5.5 Results

In our experiments, we recorded the running times of the B^{link}-array, B^{link}-linked-list, and B^{list}-tree for 10 runs per experiment, and then calculated their average running time to compare their performances. In the next sections, we describe the results of the various experiments on the Sun machines first. Then we describe the results of a subset of the experiments run on the SGI machines[4]. To have a fair comparison in all experiments, we maintained the same scale for all result graphs.

5.6 Sun Machine with Eight Processors

The Sun machines are NUMA-style machines with eight dual-core processors. In our experiments, we created threads only on the second core of each processor to ensure that no processes shared the same data bus to access the shared memory until the number of processes was more than eight. (When there are more than eight processes, two processes simultaneously share the second core of each processor of the Sun machines.) More precisely, if there are 10 processes, process 1 and 9 share the second core of the first processor, and process 2 and 10 share the second core of the second processor. We chose this creation pattern because, during early experimentation on the Sun Fire machines, we discovered that the memory access path shared by the cores on a single processor quickly became saturated and resulted in a performance bottleneck. This phenomenon was not observed on the SGI machines.

[4]Not all experiments could be run on the shared SGI machines due to long waiting times to access them. Hence, a representative subset of the experiments were chosen.

Table 1: Percentage of performance gain in B^{list}-trees compared to the B^{link}-linked-list trees with eight concurrent processes.

m	**Key Range**	**Experiment 2**	**Experiment 3**	**Experiment 4**	**Experiment 5**
500	1–250,000	25.09	21.93	8.17	17.54
	1–1,000,000	25.11	23.51	3.14	16.57
	1–2,000,000	25.96	20.39	0.98	16.59
	1–10,000,000	25.65	22.45	2.79	16.73
700	1–250,000	0.01	22.90	3.42	22.46
	1–1,000,000	0.14	20.28	6.03	21.79
	1–2,000,000	0.64	24.17	5.83	6.18
	1–10,000,000	1.44	22.34	4.73	12.57
1000	1–250,000	0.64	23.99	2.53	22.90
	1–1,000,000	0.65	24.51	4.35	23.56
	1–2,000,000	0.67	21.25	1.08	13.67
	1–10,000,000	0.02	22.38	4.74	7.36

5.6.1 Summary of the Experiments on the Sun Fire Machines

After conducting our experiments, we observed that the B^{link}-array tree always performs the best compared to the B^{link}-linked-list and B^{list}-tree. Furthermore, depending on the values of our tested parameters, the B^{list}-tree has noticeable to significant performance gain compared to the B^{link}-linked-list trees. We summarized the results from experiment 2 to experiment 5 in table 1. We did not include the results of the experiment with 100% search operations, since no lock-free or lock-based concurrency-control techniques were used in this experiment. Furthermore, with smaller m like 100 or 200, there were not much performance gain in the B^{list}-trees. Therefore, we did not include the summary of the running time, when m is 100 or 200.

From the experimental results on the Sun machines, it can be concluded that, regardless of the operation type, value of m, and key range, the B^{link}-array trees perform the best compared to the B^{link}-linked-list trees and the B^{list}-trees. This result is expected because of the structural difference between the nodes among the array-based B^{link}-array trees, and the localized linked-list-based B^{link}-linked-list trees, and B^{list}-trees (as described in Section 5.1).

When we compare the lock-based B^{link}-linked-list trees and the B^{list}-trees, there are no structural differences between these two types of tree. The only difference between them is the concurrency-control techniques used—more precisely, the difference between lock-based techniques and the lock-free techniques used in the update operations.

In the first experiment, when only search operations were conducted on both types of trees, none of the trees use any concurrency-control techniques since there were no updates on the trees. The results of this experiment show that when there is no conflict in the trees, the B^{list}-tree algorithms perform slightly better than the B^{link}-linked-list tree algorithms.

When update operations were introduced in both types of trees (experiments 2–5), the B^{link}-linked-list tree algorithms had to acquire locks to handle concurrency among concurrent update operations, whereas the B^{list}-tree algorithm used lock-free CAS as a concurrency-control technique to enable concurrent update operations. In these experiments where there was no thread-switching overhead (the experiments conducted with eight or fewer threads), the B^{list}-trees always performed better than the B^{link}-linked-list trees. Thus, in these experiments, the lock-free concurrency-control techniques outperformed the lock-based concurrency-control techniques. Because of the lock-based properties of the B^{link}-linked-list trees, a process acquires the lock of a node for an update operation when the process finds the appropriate node to be updated. Once the node is locked, the rest of the processes must wait (typically be blocked) if they want to update the same

node. Conversely, in lock-free B^{list}-trees, there is no need of such waiting. No matter how high the data contention is in a node, all processes can simultaneously work on the same node, if necessary. A process in a B^{list}-tree needs to re-do its computation only if more than one process wants to update the same cell in that node. As long as the concurrent operations do not manipulate the same cell, greater parallelism is achieved. Therefore, as we see the performance analysis of experiment 2–5, the ratio of the running time of the B^{link}-linked-list trees and the B^{list}-trees goes higher when m grows larger. The ratio increases because, when m is 500 or more, the overhead to acquire the lock of a node becomes higher (in the B^{link}-linked-list trees) than the overhead of re-doing the work of updating a cell in the node (in the B^{list}-trees). From the analysis of the results, it can be concluded that when m is relatively larger, the lock-free B^{list}-trees perform noticeably better than the lock-based B^{link}-linked-list trees.

5.7 SGI Machines

In our experiments on the Sun Fire machines, up to sixteen processes performed their operations on eight processors. Therefore, when there were more than eight processes used on the Sun Fire machines simultaneously, more than one process had to share the same processor. As a result of thread-sharing by the same processors, the running time for the experiments with update operations (experiments 2–5 in Section 5.6) on B^{list}-trees were sometimes significantly higher than the running time for the B^{link}-array and the B^{link}-linked-list tree algorithms. To see if this problem is specific to the Sun Fire machines, we did some of our experiments on the SGI machines hosted by WestGrid.

The SGI machines are also NUMA-style Symmetric Multiprocessors (SMP) machines. In our experiments on the SGI machines, we recorded the average running times of ten runs for the B^{link}-array, B^{link}-linked-list, and B^{list}-trees with 1, 2, 4, 8, 16, and 32 parallel processors and with m values of 100, 500, and 1000. With no thread sharing in the SGI machines, the running time of the B^{list}-trees was never greater than the running time of the B^{link}-linked-list trees, suggesting a limitation in the memory system of the Sun Fire machines.

Like the experiments on the Sun Fire machines, in the experiments on the SGI machines, we also populated the tree with one million random keys from different key ranges (as in Section 5.6), and then we process different mixes of search, insert and delete operations on the trees. We tested with 100% search operations (experiment 1 of Section 5.6), 50% search and 50% insert operations (experiment 3 of Section 5.6) and equal mix of search, insert and delete operations (experiment 5 of Section 5.6) on the SGI machines.

5.7.1 Summary of the Experiments on the SGI Machines

With the experiments on the SGI machines, we observed due to the cache-coherent nature of the NUMA-style parallel architecture in the SGI machines, sometime our B^{list}-tree outperforms the B^{link}-array tree. With 50% search and 50% insert operations, the performance gain is observed highes. We summarized the result of the performance gain in B^{list}-tree compared to the B^{link}-array and B^{link}–linked-list tree in table 2.

6 Conclusion and Future Work

6.1 Conclusions

B-trees are ideal for large-scale searching since their search, insert and delete operations take only logarithmic time. If a B-tree can be used concurrently by many users, the efficiency of the system increases. Many lock-based algorithms were designed for concurrent B-trees on disk. These algorithms lock some portion of the tree to allow some concurrency while maintaining consistency. However, these lock-based algorithms may offer limited concurrency in certain cases and have negative side-effects such as deadlock, convoying and priority inversion.

We presented algorithms for a B^{list}-tree that attempt to provide more concurrency for in-memory applications using lock-free techniques, applying Michael's hazard-pointer techniques to efficiently manage intra-node updates in such a way that memory locality is maintained. Furthermore, after designing the

Table 2: Percentage of performance gain in B^{list}-trees compared to the B^{link}-array B^{link}-linked-list trees with 50% search and 50% insert operations on 32 concurrent processes.

m	**Key Range**	**B^{link}-array tree VS B^{list}-trees**	**B^{link}-linked-list tree VS B^{list}-trees**
100	1–250,000	0	0
	1–1,000,000	10.47	26.69
	1–2,000,000	0	50.48
	1–10,000,000	9.82	44.39
500	1–250,000	16.02	11.706
	1–1,000,000	7.33	64.68
	1–2,000,000	37.52	56.05
	1–10,000,000	36.83	64.10
1000	1–250,000	36.09	41.16
	1–1,000,000	11.84	44.31
	1–2,000,000	6.00	17.20
	1–10,000,000	30.71	57.57

algorithms, we assessed the performance relative to two variants of Lehman and Yao's B^{link}-tree (where one variant stores the keys in an array and another variant stores the keys in a linked list). We performed all of our assessments on the NUMA-style Sun Fire and SGI machines. The results of our experiments show that B^{link}-trees with simple array-based key lists perform better than both B^{link} and B^{list}-trees with complex linked-list-based key list in their nodes. The lock-free concurrency-control techniques achieve better performance when the node structure of the trees being compared is similar (i.e., object-oriented linked-list-based key list structure), and when thread-switching overhead is comparable. However, some of the experimental results on the SGI machines show that lock-free B^{list}-tree concurrent insertions are more efficient than insertions in both types of lock-based B^{link}-trees.

Our contributions include:

- Introducing a lock-free locality-of-reference-oriented linked list to store keys in the nodes of B^{list}-trees.
- Implementing atomic CAS operations for both Sun Fire and SGI machines to perform an update operation.
- Introducing a split bit to each node to avoid unsafe concurrent update and split operations.
- Introducing a cell allocation in the beginning of an update operation to avoid unsafe concurrent update and split operations.
- Introducing a finally-linked-in-tree bit to each node to allow search processes to search in a newly-created node that is not currently accessible from the parent node.
- Designing a please-scan array to send requests to other concurrent processes to release some free cells for reuse.
- Revising the scan algorithm to return the retired cells to their associated nodes for reuse.
- Identifying a threshold level R as $m/5$ to trigger a scan operation.
- Presenting a comparative assessment on B^{link} and B^{list}-trees experimenting with different parameters.

6.2 Future Work

Some additional work could improve the performance of the B^{list}-tree. For example, a lock-free array-based B^{list}-tree on a non object-oriented key-list implementation could be designed. Also, maintaining more empty cells in the nodes might limit delays during updates. Furthermore, avoiding unnecessary rollbacks due to concurrent deletions could improve experiment results. Finally, careful work assignment on the processors might improve locality.

Lehman and Yao's array-based B^{link}-trees give the best performance compared to the linked-list-based B^{link}-trees and lock-free B^{list}-trees. The complex linked-list-based node structure has some overhead (as described in Section 5.1) on intra-node operations (i.e., search, insert and delete operations within a node) and causes delay. We could build B^{list}-trees with lock-free arrays and without using objects, to assess their performance.

Another possibility for short-term future work might be to add extra free-space in each node of the B^{list}-trees. In the experiments on the Sun Fire machines, when more than one thread share the same core, if the nodes are nearly full, the processes require more time to complete the update operations. In our B^{list}-trees, we designed the nodes to have exactly m cells to store the keys. With high data contention, the free list of a node might be empty and any process that wants to do an update operation on the node has to wait until there is a free cell in the node's free list. If a node is split after the node is 75% full, instead of 100% full, the probability of getting an empty cell will increase. Since memory is cheap, this might be an attractive option. The threshold level for node splitting might also be altered to see the effect of the fullness of nodes on overall performance.

On SGI machines, in particular, performance degradation due to deletes was obvious. This performance degradation was likely due to processes being rolled back when they find a cell that is marked for deletion. To minimize the overhead of such rollbacks, we might maintain pointers to a number of preceding cells while searching. Rather than rolling back to the beginning of the key list, we could then restart our search from a nearby safe position.

In NUMA-style machines, updating a piece of memory is most efficient when the update is performed by the processor associated with the memory segment containing the piece of memory to be updated. Therefore, when a node of a B^{list}-tree needs to be updated, the work could be assigned to the process associated with the piece of memory where the node resides. It would be very interesting to see whether the overhead of this work assignment could be made small enough to benefit from improved performance due to better locality.

References

[1] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[2] A. Biliris. Operation specific locking in B-trees. In *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 159–169, San Diego, CA, U.S.A., 23–25 March 1987.

[3] S. K. Das and M. A. Demuynck. B^{mad}-tree: An efficient data structure for parallel processing. In *Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing*, pages 384–391, New Orleans, LA, U.S.A., 23–26 October 1996.

[4] W. de Jonge and A. Schijf. Concurrent access to B-trees. In *Proceedings of the International Conference on Databases, Parallel Architectures and Their Applications (PARBASE-90)*, pages 312–320, Miami Beach, Florida, U.S.A, 7–9 March 1990.

[5] C. S. Ellis. Concurrent search and insertion in 2-3 trees. Technical report, Department of Computer Science, University of Washington, Seattle, U.S.A., May 1978.

[6] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium of Foundation of Computer Science (FOCS), IEEE*, pages 8–21, October 1978.

[7] Per Brinch Hansen, editor. *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, chapter Cooperating sequential processes, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, June 2002.

[8] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[9] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing*. The McGraw-Hill Companies, Boston, MA, U.S.A., first edition, 1998.

[10] Jong Ho Kim, Helen Cameron, and Peter Graham. Lock-free Red-Black trees using CAS. *Concurrency and Computation: Practice and experience*, pages 1–40, 2006.

[11] V. Lanin and D. Shasha. A symmetric concurrent B-tree algorithm. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, pages 380–389, Dallas, TX, U.S.A., November 1986.

[12] Philip L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, December 1981.

[13] Jianwen Ma. Concurrent, lock-free insertion in red-black trees. Master's thesis, University of Manitoba, Winnipeg, MB, Canada, June 2004.

[14] Maged M. Michael. Safe memory recalmation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 21–30, Monterey, California, U.S.A., 21–30 July 2002.

[15] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[16] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of Principles of Distributed Computing (PODC)*, pages 267–275, Philadelphia, PA, U.S.A., 23–26 May 1996.

[17] R. Miller and L. Snyder. Multiple access to B-trees. In *Proceedings of the Conference on Information Sciences and Systems*, John Hopkins University, Baltimore, U.S.A., March 1978.

[18] Yehoshua Sagiv. Concurrent operations on B-trees with overtaking. In *Fourth Annual ACM SIGACT/SIGMOD Symposium on Principles of Database System (PODS)*, pages 28–37, Portland, OR, U.S.A., 25–27 March 1985. ACM.

[19] B. Samadi. B-trees in a system with multiple users. *Information Processing Letters*, 5(4):107–112, 1976.

[20] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[21] WestGrid. `http://www.westgrid.ca`.

[22] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. Spinning versus blocking in parallel systems with uncertainty. In *Proceedings of the International Symposium on Performance of Distributed and Parallel Systems*, pages 455–472, Kyoto, Japan, December 1988.