Concurrent Cache-Oblivious B-Trees

Michael A. Bender
Department of Computer Science
SUNY Stony Brook
Stony Brook, NY 11794-4400, USA
bender@cs.sunysb.edu

Jeremy T. Fineman
MIT CSAIL
The Stata Center, 32 Vassar Street
Cambridge, MA 02139, USA
jfineman@mit.edu

Seth Gilbert
MIT CSAIL
The Stata Center, 32 Vassar Street
Cambridge, MA 02139, USA
sethg@mit.edu

Bradley C. Kuszmaul
MIT CSAIL
The Stata Center, 32 Vassar Street
Cambridge, MA 02139, USA
bradley@mit.edu

ABSTRACT

This paper presents concurrent cache-oblivious (CO) B-trees. We extend the cache-oblivious model to a parallel or distributed setting and present three concurrent CO B-trees. Our first data structure is a concurrent lock-based exponential CO B-tree. This data structure supports insertions and non-blocking searches/successor queries. The second and third data structures are lock-based and lock-free variations, respectively, on the packed-memory CO B-tree. These data structures support range queries and deletions in addition to the other operations. Each data structure achieves the same serial performance as the original data structure on which it is based. In a concurrent setting, we show that these data structures are linearizable, meaning that completed operations appear to an outside viewer as though they occurred in some serialized order. The lock-based data structures are also deadlock free, and the lock-free data structure guarantees forward progress by at least one process.

Categories and Subject Descriptors: D.1.3 **Programming Techniques** Concurrent Programming—*parallel programming;* E.1 **Data Structures** Distributed Data Structures; E.1 **Data Structures** Trees; G.3 **Probability and Statistics** Probabilistic algorithms;

General Terms: Algorithms, Theory.

Keywords: Cache-Oblivious B-tree, Concurrent B-tree, Non-Blocking, Lock Free, Exponential Tree, Packed-Memory Array.

1. INTRODUCTION

For over three decades, the B-tree [5, 14] has been the data structure of choice for maintaining searchable, ordered data on disk. Traditional B-trees are effective in large part because they minimize the number of disk blocks accessed during a search. Specifically, for block size B, a B-tree containing N elements performs $O(\log_B N+1)$ block transfers per operation. B-trees are designed to achieve good data locality at only one level of the memory hierarchy and for one fixed block size.

This research was supported in part by the Singapore-MIT Alliance, MURI–AFOSR SA2796PO 1-0000243658, USAF–AFRL #FA9550-04-1-0121, DARPA F33615-01-C-1896, and NSF Grants ACI-0324974, CNS-0305606, EIA-0112849, CCR-0208670, CCR-0121277, and CCR-9820879.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPAA'05, July 18–20, 2005, Las Vegas, Nevada, USA.
Copyright 2005 ACM 1-58113-986-1/05/0007 ...$5.00.

In contrast, cache-oblivious (CO) B-trees attain near-optimal memory performance at all levels of the memory hierarchy and for all block sizes (e.g., [7–9, 11, 13]). A CO B-tree performs a search operation with $O(\log_B N+1)$ cache misses for all possible block sizes simultaneously, and even when the block size is unknown. In a complex memory hierarchy consisting of many levels of cache, the tree minimizes the number of memory transfers between each adjacent cache level. Thus CO B-trees perform near optimally in theory, and in recent experiments have shown promise of outperforming traditional B-trees [12, 19].

One shortcoming of previously described CO B-trees is that they do not support concurrent access by different processes, whereas typical applications, such as databases and file systems, need to access and modify their data structures concurrently. This paper describes three concurrent CO B-tree data structures, proves them correct, and presents performance analysis for a few interesting special cases. The rest of this introduction reviews CO B-trees and explains the concurrency problem.

Cache oblivious B-trees

We first review the performance models used to analyze cache-efficient data structures, and then review three variations on serial cache-oblivious B-trees.

External-memory data structures, such as B-trees, are traditionally analyzed in the *disk-access model (DAM)* [1], in which internal memory has size M and is divided into blocks of size B, and external memory (disk) is arbitrarily large. Performance in the DAM model is measured in terms of the number of block transfers. Thus B-trees implement searches asymptotically optimally in the DAM model.

The *cache-oblivious model* [15, 26] is like the DAM model in that the objective is to minimize the number of data transfers between two levels. Unlike the DAM model, however, the parameters B, the block size, and M, the main-memory size, are unknown to the coder or to the algorithm. If an algorithm performs a nearly optimal number of memory transfers in a two-level model with unknown parameters, then the algorithm also performs a nearly optimal number of memory transfers on any unknown, multilevel memory hierarchy.

A CO B-tree [8] achieves nearly optimal locality of reference at every level of the memory hierarchy. It optimizes simultaneously for first- and second-level cache misses, page faults, TLB misses, data prefetching, and locality in the disk subsystem. Although the first CO B-tree is complicated [8], subsequent designs [10, 13, 19, 27] rival B-trees both in simplicity and performance [6, 10, 13, 19,

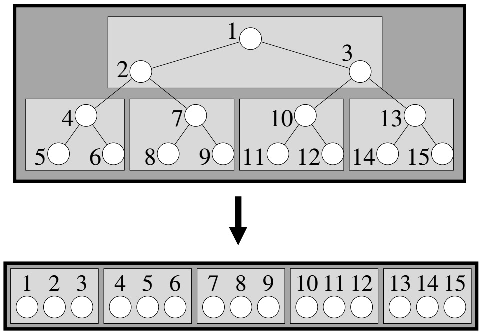

Figure 1: The van Emde Boas layout from [26]. A tree of height h is cut into subtrees at height near $h/2$, so each subtree is $\Theta(\sqrt{N})$ in size. This example shows the layout of a tree with 8 leaves.

21, 27]. Indeed, preliminary experiments have shown, surprisingly, that CO B-trees can outperform traditional B-trees, sometimes by factors of more than 2 [12, 19].

There are two main approaches to implementing *serial* CO B-trees. One approach is based on packed-memory arrays and the other on exponential search trees. Both approaches employ a static CO search tree [26] as a building block.

A static CO search tree contains a set of N ordered elements in a complete binary tree. It executes searches using $O(\log_B N + 1)$ memory transfers. The elements are laid out in an array using the *van Emde Boas layout* (see Figure 1). Each node in the binary tree is assigned to a position in a length-N array. To perform the layout, split the tree at roughly half its height to obtain $\Theta(\sqrt{N})$ subtrees each with $\Theta(\sqrt{N})$ nodes. Each subtree is assigned to a contiguous portion of the array, within which the subtree is recursively laid out. The array is stored in memory or disk contiguously.

A static CO search tree executes searches in $O(\log_B N + 1)$ memory transfers [26]. To understand this bound, consider the decomposition of the tree into subtrees of size between $\sqrt{B}$ and B. The depth of each subtree is at least $\lg\sqrt{B}$, so any root-to-leaf path encounters at most $\lg N/\lg\sqrt{B} = 2\log_B N$ subtrees. Each of these subtrees can cross at most one block boundary, leading to $4\log_B N$ memory transfers in the worst case. This analysis is not tight. In particular, [6] proves a bound of $\left(2 + \frac{6}{\sqrt{B}}\right)\log_B N + O(1)$ expected memory transfers, where the expectation is taken over the random placement of the tree in memory.

The packed-memory array [8, 17] appeared in the earliest serial dynamic CO B-tree [8] and subsequent simplifications [11, 13]. The packed-memory array stores the keys in order, subject to insertions and deletions, and uses a static CO search tree to search the array efficiently. The idea is to store all of the leaves of the tree in order in a single large array. If all the elements were packed into adjacent slots of the array with no spaces, then each insertion would require $O(N)$ elements to be displaced, as in insertion sort. The fix to this problem (known by every librarian) is to leave some gaps in the array so that insertions and deletions require amortized $O(\log^2 N/B + \log_B N + 1)$ memory transfers. The amortized analysis allows that every once in a while the array can be cleaned up, e.g., when space is running out.

An exponential search tree [2, 3] can be used to transform a static CO search tree into a dynamic CO B-tree [7, 27]. An exponential search tree is similar in structure to a B-tree except that nodes vary dramatically in size, there are only $O(\log\log N)$ nodes on any root-to-leaf path, and the rebalancing scheme (where nodes are split and merged) is based on some weight-balance property, such as *strong weight balance* [4, 8, 23, 25]. Typically, if a node contains M elements, then its parent node contains something near M^2 elements. Each node in the tree is laid out in memory using a van Emde Boas layout, and the balancing scheme allows updates to the tree enough flexibility to maintain the efficient layout despite changes in the tree.

The concurrency problem

Concurrency introduces a number of challenges. A naïve approach would lock segments of the data structure during updates. For example, in a traditional B-tree, each block is locked before being updated. Unfortunately, in the CO model it is difficult to determine the correct granularity at which to acquire locks because the block size is unknown. Locking too small a region may result in deadlock; locking too large a region may result in poor concurrency.

Second, in most database and file-system applications, searches are more common than inserts or deletes, so it is desirable that searches be *non-blocking*, meaning that each search can continue to make progress, even if other operations are stalled. Our solutions in this paper do not require search operations to acquire locks.

Another problem arises with maintaining the "balance" of the data structure. Both main approaches to designing serial CO B-trees require careful weight balance of the trees to ensure efficient operations. With concurrent updates, the balance can be difficult to maintain. For example, the amortized analysis of a packed-memory array allows a process to rewrite the data structure completely once in a while. But if the rewrite acquires locks which prevent other processors from accessing the data structure, then the average concurrency drops: on average only $O(1)$ processes can access the data structure concurrently.

Finally, there is a significant asymmetry between reading and writing the memory. We analyze our data structures under the concurrent read, exclusive write (CREW) model. Two processes can read a location in memory concurrently, since two caches can simultaneously hold the same block in a read-only state. In fact, it may be preferable if parts of the data structure, e.g., the root, are accessed frequently and maintained in cache by all processes. On the other hand, when multiple processes attempt to write to a block, the memory accesses to the block are effectively serialized, and all parallelism is lost.

We respond to these challenges by developing new serial data structures that are more amenable to parallel accesses, and then we apply concurrency techniques to develop our concurrent CO B-trees.

Our results

This paper presents three concurrent CO B-tree data structures: (1) an exponential CO B-tree (lock-based), (2) a packed-memory CO B-tree (lock-based), and (3) a packed-memory CO B-tree (non-blocking). We show that each data structure is *linearizable*, meaning that each completed operation (and a subset of the incomplete operations) can be assigned a *serialization point*; each operation appears, from an external viewer's perspective, as if it occurs exactly at the serialization point (see, e.g., [16, 22]). We also show that the lock-based data structures are *deadlock free*, i.e., that some operation always eventually completes. We show that the non-blocking B-tree is *lock-free*; that is, even if some processes fail, some operation always completes. When only one process is executing, the trees gain all the performance advantages of optimal cache-obliviousness; for example, they execute operations with the same asymptotic performance as the non-concurrent versions and behave well on a multi-level cache hierarchy without any explicit coding for the cache parameters. When more than one process is

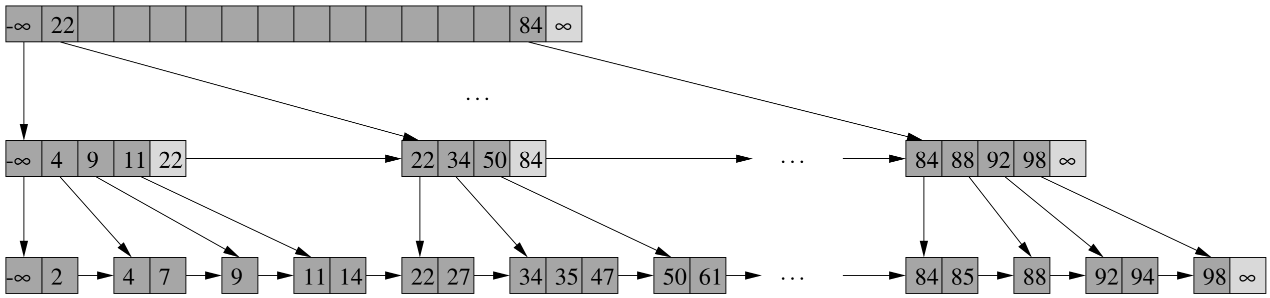

Figure 2: An exponential CO B-Tree. Nodes grow doubly exponentially in the height of the tree. The dark-gray keys of internal nodes point to the smallest key in the corresponding subtree. Data is stored in the leaves. The light-gray keys of internal nodes indicate the *right-key*. Each node has a *right-link* pointer, forming a linked list at each level in the tree.

executing, the B-trees still operate correctly and with little interference between disparate operations.

Roadmap

The rest of this paper is organized as follows. Section 2 describes our concurrent CO model. Section 3 presents a lock-based exponential CO B-tree. Section 4 presents a lock-based packed-memory CO B-tree, whereas Section 5 presents a lock-free packed-memory CO B-tree. Section 6 concludes with a discussion of other ways to build concurrent CO B-trees and of open problems.

2. THE CONCURRENT CO MODEL

In this section we describe the concurrent CO model, and then discuss the concurrency mechanisms used throughout this paper.

The CO model, as introduced in [15,26], models a single level of the memory hierarchy. The model consists of two components: the main memory and a cache of size M. Both are divided into blocks of size B. The values of M and B are unknown to the algorithm, hence the term *cache oblivious*.

In this paper we extend the model to a parallel (or distributed) setting consisting of P processors. We consider the case where each processor has its own cache of size M/P. A block may reside in multiple caches, in which case it is marked in each cache as *shared*. A block that is marked *exclusive* can reside in only a single cache. A processor can perform write operations only on blocks for which it has obtained exclusive access.

If multiple processors concurrently request shared access to a block, the block is placed in each of the caches and marked shared. If a processor requests exclusive access to a block, the block is evicted from all other caches and is marked exclusive; all other concurrent requests for shared or exclusive access fail. If a cache is full, each successful request results in an old block being evicted; we assume that the least recently used (LRU) block is evicted[1].

Each request for a block costs one memory transfer, regardless of whether the block is requested shared or exclusive and regardless of whether the request is successful. There is no fairness guarantee that a processor is eventually successful. A single read or write operation may, in fact, be quite expensive, resulting in a large number of unsuccessful block requests if there are many concurrent write requests.

We make use of two different types of support for concurrency. For much of the paper, we use locks to synchronize access to pieces of memory. When considering non-blocking algorithms, we use *load-linked/store-conditional* (LL/SC) operations. An LL operation reads the memory and sets a link bit. If any other operation modifies the memory, the link bit is cleared. An SC operation writes the memory only if the link bit is still set; otherwise the memory remains unchanged. This approach avoids the well-known ABA problem that arises with *compare and swap* (CAS). There has been much research showing how to implement LL/SC with CAS and vice versa, implying that in some senses they are equivalent (e.g., [18, 24]) and we believe it is not difficult to modify our construction for the somewhat more common CAS operation.

3. EXPONENTIAL CO B-TREE

This section presents our first concurrent CO B-tree. We first describe the data structure. We then prove correctness and give a performance analysis. We conclude by discussing some interesting aspects of this data structure.

Data structure description

The data structure uses a strongly weight-balanced exponential tree [2, 3, 7, 27] to support searches and insertions. Each node in the tree contains child pointers and a pointer to the node's right sibling (*right-link*) (see Figure 2). A node maintains the set of keys that partition its children and the key (*right-key*) of the minimum element in the *right-link*'s subtree.

We now describe the protocol for searches and insertions. The tree is parameterized by a constant α, for $1 < \alpha < 2$, which affects the height of the tree. We say that a leaf has height 0, and a node has height one more than its children.

To *search* for a key κ, we begin at the root of the tree and follow child pointers or sibling pointers until we reach the leaf containing the target element. At each intermediate node we examine *right-key*. If *right-key* is smaller than κ, then we follow the *right-link* pointer. Otherwise, we proceed to the appropriate child. On reaching a leaf, the search continues to follow *right-link* pointers until either κ is found, in which case the search returns the value, or a key larger than κ is found, in which case the search fails. We acquire no locks during the search.

To *insert* key κ, we first search for the leaf where κ should be inserted, acquire a lock on the leaf, perform the insertion at the leaf, and release the lock. Next we determine whether the key κ should be *promoted* or whether the insertion is complete. We promote the key κ in the leaf to height 1 with probability $1/2$. When promoting κ, we reacquire the lock and split the leaf u containing κ. Node u keeps all of the keys less than κ, and the new leaf acquires all of the keys greater than or equal to κ. We then release the lock and insert the promoted key κ into the parent of u. More generally, if

[1] The results in this paper hold for any reasonable replacement strategy. Unlike in the original CO model, we do not assume an optimal replacement strategy. One difficulty in the concurrent setting is that it is unclear what "optimal" means because changes to the replacement policy affect the scheduling of the program.

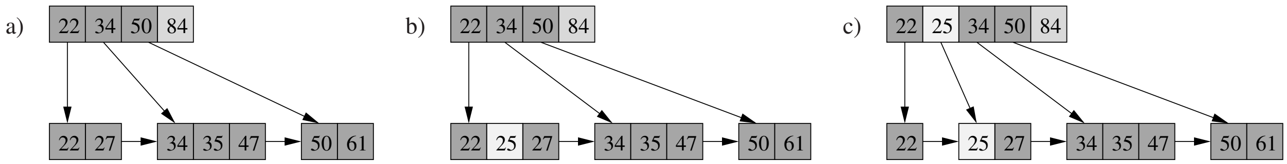

Figure 3: An example of an insert of the key 25 into an exponential CO B-Tree. a) gives the original state of the tree. b) shows the resulting structure if 25 is not promoted. c) gives the resulting structure if 25 is promoted one level.

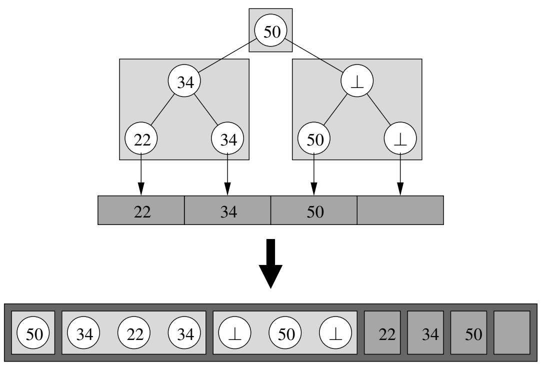

Figure 4: The modified van Emde Boas layout of a node in the exponential CO B-Tree. This figure shows the layout of the node given as the root of (a) in Figure 3

we insert a key κ into a node u' of height h, we promote κ from height h to height $h+1$ with probability $1/2^{\alpha^h}$, which means we reacquire the lock, split u', release the lock, and insert κ into the parent of u'. Figure 3 gives an example of an insert and promotion.

In order for nodes to be searched efficiently, the keys in a node are laid out using a *modified van Emde Boas layout* so that a node of size k can be traversed with $\Theta(\log_B k + 1)$ memory transfers. An example of the modified layout is depicted in Figure 4. The node is divided into two pieces: a size-$\lceil\!\lceil k \rceil\!\rceil$ array[2] filled from the left holding all the keys in a node, and a complete $\lceil\!\lceil k \rceil\!\rceil$-leaf static CO search tree used to efficiently search for an array slot. The i-th leaf of the search tree points to the i-th array slot. In particular, a leaf containing κ in the search tree points to κ in the array. This approach is reminiscent of [10] and reappears in Section 4. The static CO search tree consists of keys laid out in a van Emde Boas layout [26]. This layout consumes (roughly) the first $2/3$ of the memory used by a node. The final $1/3$ of the memory contains the keys stored in order as a vector. A search within a node ends at the location of the key in the vector.

When we rewrite a node (either because the key is inserted into the node or because the node is split), we update the modified van Emde Boas layout as follows. First we rewrite the vector of keys (the right half of the node), proceeding from largest to smallest. Then we update the van Emde Boas layout in the left half. This layout ensures that we can perform concurrent searches even while the node is being updated. If an insert is performed when the vector of keys is already full, we allocate a new node of twice the size.

Correctness

We first argue that the data structure is correct, even under concurrent operations. The most common way of showing that an algorithm implements a linearizable object is to show that in every execution there exists a total ordering of the operations with the following properties: (1) the ordering is consistent with the desired insert/search semantics, and (2) if one operation completes before another begins, then the first operation precedes the second in the ordering. Linearizability follows due to a straightforward extension of Lemmas 13.10 and 13.16 in [22].

Theorem 1. *The exponential CO B-Tree guarantees linearizable insertions and searches and is deadlock-free.*

Proof. In order to show that an appropriate total ordering of the operations exists, we need to show that if an operation completes, all later operations are consistent with it. If an insert operation finishes inserting a key κ at level 0, or a search finds a key κ at level 0, then any later search also finds key κ.

First, notice that at every level of the tree, the keys are stored in order. That is, if the largest key in node u is κ, then the smallest key in node u.*right-link* is larger than κ. This fact follows from the two ways in which a node is modified. First, a node may be split, say at key κ. In this case, the split operation, protected by a lock, preserves this invariant, moving κ and all larger elements into a new node that can be reached by *right-link*. Second, a key κ may be inserted into a node. In this case, the node is locked during the insertion, and some of the elements in the vector half of the node move one position to the right to make room for κ.

This ordering of keys implies that if key κ is in the leaf node of the tree when a search begins, then throughout the search, a leaf containing key κ is reachable by the search. The search begins at the root, and every leaf is reachable from the root. We proceed by induction: at each step of the search, a child pointer is chosen with a minimum key no greater than κ. The only interesting case is when the child node is concurrently split. Even in this case, however, the child node always contains a key no greater than κ, since ever when a node is split it always maintains its minimal key. We conclude that κ is still reachable.

Finally, deadlock-freedom follows immediately, since each process holds only one lock at a time. □

Cache-oblivious performance

We first consider the cost of individual operations when the data structure is accessed sequentially. We then analyze the concurrent performance for the special case when search and insert operations are performed uniformly at random. In both cases, we use the cache-oblivious cost model, counting only cache misses.

Theorem 2. *Assume that operations occur sequentially. A search in an N-node tree takes $O(\log_B N + \log_\alpha \lg B)$ block transfers, with high probability; an insert takes $O(\log_B N + \log_\alpha \lg B)$ block transfers in expectation.*

Proof. First, notice that if a tree contains N keys, then with high probability every node in the tree has height less than $\log_\alpha \lg N + O(1)$. Specifically, the probability that any key is promoted to height $\log_\alpha \lg N + c$ is $O(n^{-\alpha^c+1})$.

[2]The *hyperceiling* of x, denoted $\lceil\!\lceil x \rceil\!\rceil$, is defined to be $2^{\lceil \log x \rceil}$, i.e., the smallest power of 2 greater than x.

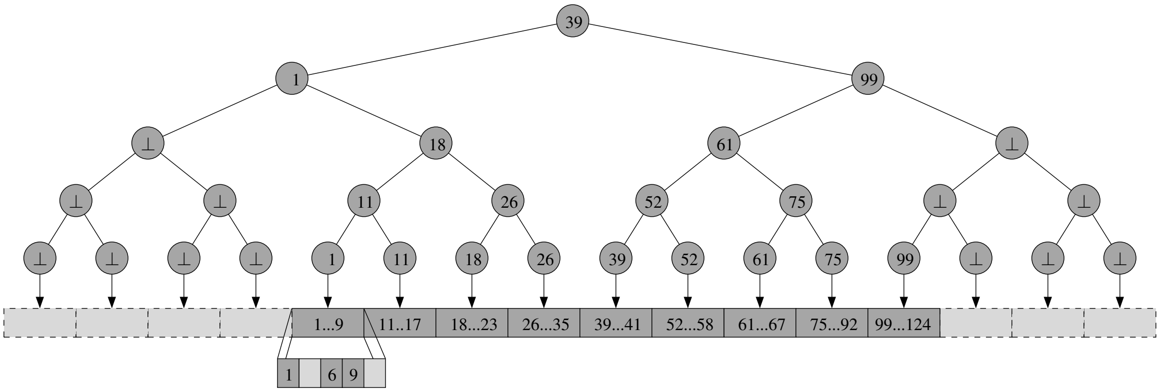

Figure 5: A packed-memory CO B-Tree consists of a static cache-oblivious search tree used to index into the one-way packed-memory array. Each leaf in the tree points to a $\Theta(\lg N)$-sized region of the array. The active region of the array is outlined with a solid border. The active region of the array contains gaps allowing for insertions, and the array can grow to the right or shrink from the left.

Also notice that if u is a node of height h, then the number of keys in u is $O(2^{\alpha^h})$ in expectation and $O(2^{\alpha^h}\lg N)$ with high probability. For $h = \Omega(\log_\alpha \lg\lg N)$, that is, when the expected node size is at least $\Omega(\lg N)$, the number of keys is $O(2^{\alpha^h})$ with high probability.

We now calculate the cost of a search operation. Searching a node of height h, laid out in the modified van Emde Boas layout, for the correct child pointer takes $O(\log_B(2^{\alpha^h}) + \log_B \lg N + 1)$ memory transfers, with high probability. For nodes with expected size at least $\Omega(\lg N)$, the cost is $O(\log_B(2^{\alpha^h}) + 1)$ with high probability. We sum the costs for all levels, ranging from $i = 0$ to $\log_\alpha \lg N + c$ (with high probability):

$$\Theta\left(\sum_{0}^{\log_\alpha \lg N + c} \log_B(2^{\alpha^h}) + \sum_{0}^{\log_\alpha \lg\lg N} \log_B \lg N\right).$$

The right term sums to $O(\log_B N)$. For the left term, we consider two cases, depending on whether or not the size of the node is at least B. For all levels where the nodes are of size at least B, the cost is dominated by the root node (or possibly a node just below the root). For nodes of size less than B, the block transfer cost is at most 2. Therefore, with high probability, the total cost of a search is $O(\alpha^c \log_B N + \log_\alpha \lg B)$.

We next calculate the cost of an insert operation. Consider the expected cost of inserting a key at height h of the tree. Recall that we promote a key from level $h-1$ to level h with probability $2^{-\alpha^{h-1}}$. Thus, the probability of promoting a given key to height h is $2^{-(\alpha^h - 1)/(\alpha-1)}$. The cost of inserting a key into a node at height h, given that the key reaches height h, is the cost of rebuilding the entire node, which is $O(1 + 2^{\alpha^h}\lg N/B)$ with high probability.

Therefore, the expected cost at height h is the probability that the insertion reaches height h times the cost of rewriting a node at height h. The expected cost is therefore $O(2^{-c'(\alpha^h - 1)}\log N/B)$, where $0 < c' < 1$ depends on the value of α. The summation across all $\log_\alpha \lg N + O(1)$ levels, then, is $O(\lg N/B) \le O(\log_B N)$. Hence the total expected cost of an insert is the cost of searching for the right place to do the insertion plus the cost of doing the insertion. That is, the cost is $O(\log_B N + \log_\alpha \lg B)$. $\square$

Notice that if there are a large number of concurrent search operations, but no insert operations, each search operation incurs a cost of $O(\log_B N + \log_\alpha \lg B)$, the same as in the sequential case.

If there are many concurrent searches and insertions, the performance may deteriorate. The performance of concurrent B-trees is difficult to analyze since it depends significantly on the underlying parallelism of the algorithm using the data structure. For example, if a large number of processes all try to modify a single key, then the operations are inherently serialized, and there is no possible parallelism. As a result, for most concurrent B-tree constructions (and in fact, most concurrent data structures), little is stated about their performance. We analyze the "optimally parallel," case where each search and insertion operation targets a randomly chosen key, and show that the data structure still yields good performance.

Theorem 3. *Assume that all processors are synchronous. If there are $O(N^{2/\alpha-1}/\mathrm{polylog}N)$ search and insert operations performed uniformly at random in an N-node tree, then the expected cost for an operation is $O(\log_B N + \log_\alpha \lg N)$ block transfers.*

Proof (sketch). Notice that in the absence of concurrent insert operations, each search operation takes $O(\log_B N + \log_\alpha \lg B)$ block transfers. If an insert operation delays a search, we charge that cost to the insert operation.

An insert operation has four costs: finding the insert point, acquiring the lock, performing the insertion, and delaying other operations. As in the case of searches, finding the appropriate insertion point costs $O(\log_B N + \log_\alpha \lg B)$ plus any delays caused by other insert operations. Again, this delay is charged to the operation causing the delay.

The cost of performing the insertion itself is equivalent to the sequential case: $O(\log_B N + \log_\alpha \lg B)$ block transfers.

It remains to calculate the effects of concurrency: the cost of acquiring the lock and the cost of delaying other operations. Consider the expected cost incurred by a delaying one particular concurrent operation at some level h. There are three components to the cost: (1) the probability that the insert reaches level h, (2) the probability that the insert modifies the same node, and (3) the actual cost incurred by delaying the concurrent operation.

First, as in Theorem 2, an insert operation reaches level h with probability $2^{-(\alpha^h-1)/(\alpha-1)}$.

Next, an insert chooses the same node at level h with probability $2^{(\alpha^{h+1}-1)/(\alpha-1)}/N$, since the expected number of nodes at level h is the number of elements promoted to level $h+1$. This probability can vary by at most a log factor, with high probability, which is enough for the proof.

Finally, we consider the real cost of delaying any one operation at level h. The expected size of a node at level h in the tree is $O(2^{\alpha^h})$. Therefore the concurrent operation can be delayed by at most $O(2^{\alpha^h})$ in expecation, which is the cost of performing one memory transfer for each element in the node. Notice that this worst-case analysis makes no use of the cache: in the case of a search operation, the block may be repeatedly transferred back and forth between the search operation and the insert operation. Moreover, the search may have to read the entire node since the contents of the node are changing during the search. The size of the node also bounds how long the insert may need to wait to acquire the lock.

We now sum over the possible heights in the tree. Consider $h \le h_{max} = \log_\alpha \lg N + \log_\alpha(\alpha-1) - 1$, which is the point at which the second term (i.e., the probability that the insert occurs at the same node) reaches 1. Multiplying the three terms results in an expected cost of $2^{2\alpha^h}/N$. In the worst case, where $h = h_{max}$, this expression is equal to $N^{1-2/\alpha}$; for smaller h, the cost decreases. Since there are at most $N^{2/\alpha-1}$ concurrent operations, the expected cost per level is $O(1)$. Summing over all levels $\le h_{max}$ leads to an expected cost of $O(\log_\alpha \lg N)$.

Next consider the case where $h \ge h_{max}$. Notice that the second term never exceeds one. The first and third terms decrease geometrically like $O(1/2^{\alpha^h})$, since the tree is unlikely to exceed height h_{max}, so again the sum is bounded by the term $h = h_{max}$, as before.

The polylog in the theorem arises since there can be an $O(\log N)$-factor variation in the size and number of nodes at a level, with high probability. □

Notice that in a concurrent setting, the expected cost is $O(\log_B N + \log_\alpha \lg N)$, instead of $O(\log_B N + \log_\alpha \lg B)$ in the sequential case. The additional cost is incurred when two operations interfere.

Discussion

One interesting aspect of this data structure is parameterizing of the tree by α. By skewing the tree wider (rather than deeper), we reduce the concurrency at the root, thus improving the overall performance. When α is closer to 1, the concurrent performance is better; when α is closer to 2, the sequential performance is better.

A second aspect to note is that the data structure requires relatively minimal memory management, since it does not support delete operations. The only case in which memory must be deallocated (or wasted) is when a node in the exponential tree grows too big (before splitting), and a new contiguous block of memory must be allocated for the node.

4. PACKED-MEMORY CO B-TREE

In this section we present our second concurrent, cache-oblivious B-tree. It supports insertions, deletions, searches, and range queries using lock-based concurrency control. We first describe the lock-based data structure. We then prove the data structure correct and analyze its serial performance.

Data structure description

Here we describe the lock-based packed-memory CO B-Tree. We present a lock-free version of this data structure in Section 5. Our data structure is based on the cache-oblivious B-tree presented in [11] and consists of a static cache-oblivious search tree [26] that is used to index a packed-memory data structure. Instead of using the packed-memory array from [8], we introduce a new "one-way packed-memory structure," which is more amenable to concurrent operations. Each leaf in the static tree points to a $\Theta(\log N)$-size region in the packed-memory array.

The one-way packed-memory structure maintains N elements in order in an array of size $m = \Theta(N)$ (with m a power of 2) subject to element insertion and deletion (see Figure 5). The array consists of three segments: the leftmost and rightmost segments contain extra empty space, and the middle segment, the *active region*, contains the N elements and some gaps between elements. On an insertion, the active region may grow to the right; on a deletion, the active region may shrink from the left. If the active region grows or shrinks too much, then we reallocate the array.

We maintain a near-constant "density" in the active region by *rebalancing* regions of the array—that is, evenly spreading out elements within a region—on insertions or deletions. In the one-way packed-memory structure, the rebalances ensure that elements move only in one direction: to the right (see Lemma 4).

The *density* of a subarray is the number of filled array positions divided by the size of the subarray. Consider a subarray of size k with $i = \lceil \lg k \rceil$. The size of a subarray determines its *upper-bound density threshold*, τ_i, and its *lower-bound density threshold*, ρ_i. For all i and j, $\rho_i < \rho_{i+1}$, $\tau_j > \tau_{j+1}$, and $\rho_i < \tau_j$.

The density thresholds follow an arithmetic progression defined as follows. Let $0 < \rho_{min} < \rho_{max} < \tau_{min} < \tau_{max} = 1$ be arbitrary constants. Let $\delta = \tau_{max} - \tau_{min}$ and $\delta' = \rho_{max} - \rho_{min}$. Then, define density thresholds τ_i and ρ_i to be

$$\tau_i = \tau_{max} - \frac{i-1}{\lg m - 1}\delta \text{ , and } \rho_i = \rho_{min} + \frac{i-1}{\lg m - 1}\delta' \text{ ,}$$

for all i with $1 \le i \le \lg m$.

For example, suppose that $m = 16$, and we fix the threshold $\tau_{min} = 1/2$. There are $\lg 16 = 4$ density thresholds $\tau_1 = \tau_{max}$, τ_2, τ_3, and $\tau_4 = \tau_{min}$. The thresholds increase linearly with $\tau_1 = 1$, $\tau_2 = 5/6$, $\tau_3 = 4/6$, and $\tau_4 = 1/2$. Similarly, if we fix $\rho_{min} = \rho_1 = 1/8$ and $\rho_{max} = \rho_4 = 1/4$, then $\rho_2 = 4/24 = 1/6$ and $\rho_3 = 5/24$.

Search. To search for a key κ, search down the static binary tree as normal, without any locks. When the search reaches the array, scan right until finding the appropriate array slot. Since the search proceeds without locks, we need to perform an ABA test (key, value, key) to make sure the element did not move during the read.

Insertion and deletion. To insert a new element y with key κ, first perform a search to find and then lock the $\Theta(\log N)$-sized leaf where key κ should be inserted. Next, scan the leaf from left to right to find the actual slot s for key κ. (Slot $s-1$ should contain the largest key smaller than κ.) If the scan advances into the next $\Theta(\log N)$-size region (due to a concurrent operation), acquire a lock on the next tree leaf, give up the lock on the current, and continue.

If the slot s is free, place y in the available slot. Otherwise, we must *rebalance* a section of the packed-memory array. Explore right from s in the array, acquiring locks on regions along the way, until finding the smallest region *of any size* that is not too dense. A region of size k, with $\lceil \lg k \rceil = i$, is not "too dense" when the density is no greater than τ_i. We then rebalance the elements evenly within the window by moving elements to the right, starting with the rightmost element and working to the left. We move the elements such that no suffix of the rebalanced region has density exceeding the threshold τ_i.[3] Finally, we release the locks. Unlike the packed-memory array from [8] rebalance windows do not have to have sizes that are powers of 2. Figure 6 gives an example of an insert into the one-way packed-memory array.

Deletions are analogous to insertions, except that when an element is deleted, we always perform a rebalance. A deletion-triggered rebalance explores right until finding the first slot preceding an occupied slot, then explores *left* until finding a region that is

[3]Thus, every prefix of the region not including the last slot has density at least the threshold τ_i. In particular, a prefix of size k has $\lceil k\tau_i \rceil$ elements.

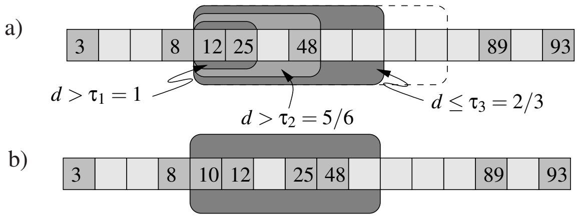

Figure 6: An insertion of the key 10 into the above packed-memory array of size $m = 16$ and thresholds $\tau_{max} = 1$ and $\tau_{min} = 1/2$. The new element 10 should go in the slot containing 12, so we explore right until finding a region that is not too dense, including the new item to insert. Until exploring past a region of size 2 (indicated by the small rounded rectangle in (a)), the density d exceeds the corresponding threshold $\tau_1 = 1$. When exploring between a region of size between 3 and 4, indicated by the medium rounded rectangle, the density is 1 which exceeds the corresponding threshold $\tau_2 = 5/6$. When exploring regions of size between 5 and 8, we use the threshold $\tau_3 = 2/3$. The exploration stops at the sixth slot as there are 4 elements (including the new element 10) to place in 6 slots, giving a density $d = 4/6 = 2/3 = \tau_3$. (b) gives the state of the array after the insert and rebalance. Once the rebalance region is established, we begin at the right and move elements as far right as possible without exceeding the appropriate threshold (i.e., $\tau_3 = 2/3$).

"dense enough." This exploration proceeds without locks. Once the region is established, we acquire locks on the region from left to right. As with the insertion-triggered rebalance, we then spread elements evenly starting with the rightmost element and working left. Finally, we release all the locks.

Once the insert/delete or rebalance is complete, we update the keys in the static search tree. For each node in the subtree defined by the updated region, if the key at a node in the tree is too small, we lock the node, write the key, and release the lock. This strategy ensures that concurrent searches are not blocked.

Resizing the array. When the right boundary of the active region hits the array boundary, or the active region becomes too small, we *resize* the array, copying the elements into a new array, spreading the elements evenly. The "ideal" density of the active region is $(\tau_{min} - \rho_{max})/2$. If the old array has N elements, the new array has size $\left\lceil\left\lceil c\frac{N}{(\tau_{min}-\rho_{max})/2}\right\rceil\right\rceil$, for a constant c. The active region starts at the left portion of the new array.

We use a randomized strategy that allows the resizing to run quickly and concurrently. The resize proceeds in three phases: (1) We randomly choose leaves of size $\Theta(\log N)$ and count the elements in the leaf. We lock the node, write the count in the node, and release the lock. If the sibling has been counted, we write the sum in the parent and proceed up the tree. When the root node is counted, the first phase ends. (2) We allocate a new array based on the count in the root. Note that the count in any node is exact—the randomization in the first phase randomizes only the order in which nodes are counted. (3) We randomly walk down the original tree to find a leaf, keeping a count of the number of elements to the left of this leaf. Then, we copy the $\Theta(\log N)$ elements in the leaf to their correct location in the new array (spreading the elements evenly). For phases (1) and (3), we mark nodes to ensure that leaves are not counted or copied multiple times and to discover when the phases complete.

Correctness

We show that our data structure is correct under concurrent operations: the data structure guarantees linearizable operations and is deadlock-free.

The first lemma shows that the one-way rebalance does in fact move elements in only one direction. This lemma directly implies that searches return the correct elements.

Lemma 4. *The one-way rebalance moves elements only to the right. That is, the rebalance operation maintains the property that the key at a given memory location is nonincreasing.*

Proof. Consider a rebalance region r ranging from array slots s_1 to s_2. Let $i = \lceil \lg(s_2 - s_1 + 1) \rceil$. Then the elements in r are spread evenly to a density of τ_i.

Assume for the sake of contradiction that an element in r moves left during the rebalance. Without loss of generality, let s_2' be the slot of the leftmost element that moves left. Consider the region r' ranging from s_1 to $s_2' - 1$. Let d' be the density of region r'. Since spreading the elements evenly in r moves an element into r', we have $d' < \tau_i$. However, since r' is contained in r, we have $j = \lceil \lg(s_2' - s_1) \rceil$ for some $j \leq i$. Thus, $\tau_j \geq \tau_i \geq d'$, generating a contradiction as the rebalance region would be r'.

The proof is similar for deletion-triggered rebalances. □

Theorem 5. *The packed-memory CO B-Tree is deadlock-free and guarantees linearizable insertions, deletions, and searches.*

Proof. The deadlock-free proof follows directly from the fact that when an operation holds more than one lock, it always acquires these locks from left to right.

Next, we look at the claim of linearizability. The array of the packed-memory CO B-Tree is similar to a single node in the exponential CO B-Tree, except the array is slightly more complicated. The important property about updates to the array is given in Lemma 4. Thus, we can use a similar proof to Theorem 1. □

Cache-Oblivious Performance

We next analyze the cost of sequential operations in the packed-memory CO B-Tree in terms of block transfers. We first bound the cost of the static-search-tree update and of resizing the array. We then apply an accounting argument to conclude that the packed-memory CO B-Tree achieves the same serial costs as in [8, 10, 13].

First, we bound the cost of the static tree update.

Lemma 6. *Assume that operations occur sequentially. If a range of k memory locations are modified during an update in the packed-memory array, then updating the search tree costs $O(\log_B N + k/B + 1)$ memory transfers.*

Proof. We can think of subtrees of the search tree as corresponding to ranges in the packed memory array. Updating the tree requires updating every node in a set of subtrees that constitute the range of k memory locations. The combined number of tree nodes in these subtrees is less than $2k$, but it remains to be shown that these nodes fit in $O(k/B + 1)$ blocks. Consider all the subtrees $\{T_1, \ldots, T_r\}$ to update from left to right. Then there exists an i such that the subtree rooted at $parent(T_i)$ contains the subtrees $T_1, \ldots, T_i$, and $parent(T_{i+1})$ contains $T_{i+1}, \ldots, T_r$. Thus, the tree nodes to update are contained in two subtrees, with total size at most $4k$. All of these subtrees, therefore, are laid out in $O(k/B + 1)$ blocks.

Additionally, some nodes, not in these subtrees, along the path to the root must be updated. We note that we only update nodes with a right child that changes. Thus, we update only a single path to the root, which requires at most $O(\log_B N + 1)$ blocks. □

The insertion cost includes not only the cost of rebalancing the array and updating the static tree, but also the cost of resizing the array. The following lemma implies that the array is not resized very frequently.

Lemma 7. *Consider a packed-memory array of size m. Then there must be $\Omega(m)$ insertions or deletions before the array is resized.*

Proof. Consider a resizing triggered by an insertion. We use an accounting argument to prove the lemma. We give 1 dollar to each filled slot in the array at the time of the last resizing. Whenever an item is inserted into some slot, we give that slot 1 dollar. Whenever an item is deleted, we leave the dollar in the slot. Whenever we rebalance a region of the array, we move 1 dollar with each item moved. All excess dollars (i.e., associated with items the have been removed) are moved to the first slot of the rebalanced region. Clearly every nonempty slot has at least 1 dollar associated with it.

Let $d = (\tau_{min} - \rho_{max})/2$—the density to which the array is resized. Then we claim that any prefix of the array, starting at the left boundary (slot 0) and ending at a slot s preceding the right boundary of the active region, contains at least sd dollars. This invariant is clearly true at the time of the last resizing. It also trivially holds across insertions or deletions that do not trigger rebalances. It remains to be shown that the invariant holds across rebalances.

Consider an insertion-triggered rebalance that ranges from slot s_1 to s_2. The invariant is trivially unaffected for any slot s with $s < s_1$ or $s \geq s_2$ since no money moves into or out of the rebalanced region. It remains to show that the invariant holds for a slot s with $s_1 \leq s < s_2$. By assumption, we have at least $(s_1 - 1)d$ preceding slot s_1. The rebalance algorithm guarantees that the density of every prefix of the rebalanced region is at least τ_{min}. Thus the array contains at least $(s_1 - 1)d + (s - s_1 + 1)\tau_{min} \geq sd$ dollars by slot s.

For completeness, we also need to consider a deletion-triggered rebalance that ranges from slot s_1 to s_2. Consider a slot s with $s_1 \leq s < s_2$. The rebalance algorithm again guarantees that the density of every suffix of the rebalanced region is at most $\rho_{max} < d$. Thus, since the invariant holds at slot s_2 before the rebalance, and all extra dollars are moved to slot s_1, we have that the invariant holds for all slots after the rebalance.

Now we just apply the invariant to complete the proof. When an insertion triggers a rebalance, the active region includes the entire array. Thus, the array contains at least md dollars. At the last resizing, there were md/c dollars, where both $c > 1$ and $0 < d < 1$ are constants. Thus, there must have been $\Omega(m)$ insertions.

The proof for deletion-triggered resizings is similar. □

Now we bound the cost of resizing. The main idea is that $\Theta(N)$ random choices is enough to count $\Theta(N/\lg N)$ leaves.

Lemma 8. *Assume that operations occur sequentially. Consider a packed-memory CO B-Tree containing N elements. Then the cost of resizing the packed-memory array is $O(N(\log_B N + 1))$ memory transfers.*

Proof. There are $\Theta(m/\lg m)$ leaves. We make only $O(m)$ random leaf selections, with high probability, before selecting every leaf in phases (1) and (3). In phase (3), finding a leaf follows a root-to-leaf path in the tree with a cost of $O(\log_B m + 1)$. The total cost of selecting all the leaves is, therefore, $O(m(\log_B m + 1))$.

Copying or counting a $\Theta(\lg m)$-sized region (corresponding to a tree leaf) of the old array takes $\Theta(\lg m/B + 1)$ block transfers. Since each leaf is counted and copied once in phases (1) and (3), respectively, the total cost of counting and copying the elements is $\Theta((m/\lg m)(\lg m/B + 1)) = O(m(\log_B m + 1))$.

Copying a $\lg m$ sized region from the old array to the new array takes $\Theta(\lg m/B)$ block transfers (since the array is kept near a constant density), for a total cost of $O(m/B)$ across the entire resize.

Updating the tree for the new array costs $O(\log_B m' + 1)$ for each element copied where m' is the size of the new array, for a total of $O(m(\log_B m' + 1))$ block transfers.

Since m and m' are $\Theta(N)$, the lemma follows. □

The following theorem states that we achieve the desired serial performance.

Theorem 9. *Assume that operations occur sequentially. The amortized cost of insertions and deletions is $O(\log_B N + \lg^2 N/B + 1)$.*

Proof. This proof is similar to the one in [20] that analyzes the cost of rebalance regions extending only one direction (but in which elements can move in both directions). This argument uses the accounting method, placing $\Theta(\lg^2 m/B)$ dollars in an array slot on an insertion/deletion. In particular, $\Theta(\lg m/B)$ dollars are associated with each of $\lg m$ accounts, corresponding to $\lg m$ density thresholds. Whenever a region of size $k > \lg m$ is rebalanced[4], the region contains $\Omega(k/B)$ dollars in an appropriate account with which to pay for the rebalance. Our scenario is different because regions are not necessarily powers of 2, and our resizing algorithm is different. Whereas [20] uses an accounting argument that charges rebalances against the left half of the region of size k, we charge against the left subregion of size $\lceil\!\lceil k \rceil\!\rceil/2$.

Moreover, we incur a cost for updating the static tree on top of the packed-memory data structure during a rebalance. Since we have $\Theta(k/B)$ potential saved up at the time of a rebalance of size k, and a tree update costs $O(\log_B N + k/B + 1)$ (from Lemma 6) we can afford the tree update.

Finally, given that there must be $\Omega(N)$ insertions between resizings (see Lemma 7), and a resizings costs $O(N(\log_B N + 1))$ (see Lemma 8), we conclude that the cost of resizing the array is amortized to $O(\log_B N + 1)$ per insertion. □

5. LOCK-FREE CO B-TREE

We now show how to transform the lock-based data structure in Section 4 into a non-blocking, lock-free CO B-tree. Instead of using locks, we use load-linked/store-conditional (LL/SC) operations. For the sake of clarity, we will assume that keys and values are each a single word, and can be read and written by a single LL/SC operation; the data structure is easily extensible to multi-word keys and values.

Data structure description

Recall that locks are used only in updating the packed-memory array; the static search tree is already non-blocking. Instead of acquiring locks before modifying the packed-memory array, we show how to use four basic non-blocking primitives to update the data structure: (1) *move*, which moves an element atomically from one slot in the packed-memory array to another, (2) *cell-insertion*, which inserts a new key/value pair into a given cell in the packed-memory array, (3) *cell-deletion*, which deletes an existing key/value pair from a cell in the packed-memory array, and (4) *read*, which returns the key and value of a given cell in the packed-memory array. Each of these primitives is non-blocking, and may fail when other operations interrupt it. For example, a move operation may fail if a cell-insertion is simultaneously performing an insertion at the target.

Markers. Each cell in the array is augmented with a *marker* which indicates whether an ongoing operation is attempting to modify the cell. The marker contains all the information necessary to complete the operation. For example, a move marker indicates the source and the destination of the move. Any processor that is performing an operation and discovers a marked cell helps to complete the operation indicated by the marker. For example, consider a move

[4] We can trivially pay for any small rebalances with the cost of inserting the new element.

operation that is attempting to move an element from cell 14 to cell 15, while concurrently a cell-insert is attempting insert an element at cell 15. First, the cell-insert updates the marker at cell 15. Then, the move attempts to update the marker, and discovers the concurrent insertion. The move operation then performs the insertion, before proceeding with the move. In this way, the move can eventually complete, even if the processor performing the cell-insert has failed, or been swapped out of memory.

The primitive operations which modify the data structure (i.e., move, cell-insert, and cell-delete) are all initiated by marking an appropriate cell. Once a cell has been marked, any processor can complete the operation by simply processing the marker. In order to begin a move operation, the source of the move is updated. For a cell-delete operation, the cell containing the element to be deleted is marked. A cell-insert operation marks the cell immediately *preceding* the cell where the new element is being inserted. A requirement of a cell-insert is that this preceding cell not be empty. By marking the preceding cell, we prevent a concurrent move operation from moving an element "over" an ongoing cell-insertion. For example, if a cell-insert is happening at cell 15, we must prevent the element from cell 14 (or any smaller cell) from being moved to cell 16 (or any larger cell); otherwise, the new element might not be ordered correctly.

We believe that it is possible to implement a lock-free packed-memory CO B-Tree using compare-and-swap, instead of LL/SC, by adding version tags to the markers.

Implementing the nonblocking primitives. For a move operation, once the source has been successfully marked, an LL operation is performed on the following items: (1) the marker, (2) the source key, (3) the source value, (4) the destination key, and (5) the destination value. Then, an SC is performed on the marker, rewriting the marker. This technique ensures that no concurrent process has modified the marker in an attempt to help complete the move. The move then completes by using SC to update the keys and values at the destination, and then at the source. If an SC fails during this final stage, it is ignored; some concurrent process has already helped to complete the move. Finally, the marker is cleared.

For a cell-insert, once the preceding cell has been marked, the insert performs a LL on the marker, and then on the keys and values at the new cell. If the key is already in the tree, then an error is returned. If the cell is not empty, an error is returned. The insert then rewrites the marker with an SC, ensuring that it has not changed in the interim. Finally, the key and value are updated, and the marker is cleared. A cell-delete is essentially identical to a cell-insert.

Finally, a read operation simply examines a cell, helps out if the cell is marked, and returns the appropriate values.

Insertions and deletions. An insertion proceeds as before, first using the static search tree to find and mark the appropriate cell in the array, that is, the cell containing the largest key in the tree that is less than or equal to the key being inserted. Once the cell is marked, a cell-insert begins. If the insertion succeeds, the element is successfully inserted. If the cell-insertion fails, however, then either the cell is not empty or a move caused interference. In this case, we need to rebalance the array.

A rebalance begins by exploring to the right, as before. In this case, however, it remembers which cells were filled and which were empty. It performs a load-link (LL) operation on each non-empty cell; each of these marked cells may need to be moved, and the move is initiated by performing an SC on the marker. In this way, the rebalance operation can detect when the array has changed during the rebalance.

When an appropriately sparse region is found, the operation can calculate the appropriate spacing of the elements in the array, as before. The rebalance then proceeds from right to left using move operations to spread elements evenly. (Elements are only moved from left to right, as before.) If any move fails, then the rebalance restarts. In particular, a move may fail if any of the markers has changed since they were initially linked during the scanning phase of the rebalance. If at any time during the rebalance, we otherwise detect that the array has changed, then the rebalance restarts.

Deletions are similar to insertions. We first search for the item to be deleted, and then perform a cell-deletion. Finally, we scan to the right until discovering a non-empty cell, and then perform a rebalance. Unlike the rebalance on insertions, however, this operation scans to the left, looking for a dense region (as is described in Section 4).

All other data structure operations proceed as before; after the array has been updated using the non-blocking primitives, the search tree is updated to reflect the changes. When resizing the array, load-link/store-conditional is used to atomically write the count of a leaf, instead of a lock.

Correctness

Theorem 10. *The packed-memory CO B-Tree guarantees linearizable search, insert, and delete operations.*

Proof. If a key is in the data structure and the data structure remains in sorted order, then a search will find it: the static tree is updated after the packed-memory array, and elements are only moved to the right in the packed-memory array; it is therefore easy to see that a search always exits the static tree to the left of the key in question.

The key property, then, is that the elements in the array remain in sorted order. If they always remain in sorted order, then a search will correctly find any previously inserted element or any element returned by a prior search and it fail to find a previously deleted element, as is required.

In order to show that elements remain in order, we examine how elements are inserted. An important invariant is that if a cell is marked for a cell-insert, then the cell contains the largest key that is less than or equal to the key being inserted. Initially, on acquiring a marker, this invariant is ensured by the correctness of the search which locates the insertion cell, and the use of LL/SC to acquire the marker atomically.

Throughout the insertion, this invariant is maintained by the way in which a rebalance moves elements. In particular, a rebalance never moves an element from one side of a marked cell to another. In particular, whenever a rebalance moves an element, all the intervening cells are empty. This property is checked while the rebalance scans to the right, determining the region to rebalance and simultaneously performing a load-link (LL) on each non-empty cell in the region. The only way this property of a rebalance can be violated is if an element is inserted or moved between when the region is scanned and the rebalance moves an element. In this case, however, the SC which acquires the move marker fails.

As a result, elements are always inserted in order, and rebalances never move elements out of order. Combined with the fact that searches terminate correctly, we have the main result. □

Theorem 11. *The packed-memory CO B-Tree is a lock-free data structure.*

Proof (sketch). We need to show that if there is at least one ongoing operation, then eventually some operation completes. An operation can only be delayed by pausing to help a concurrent operation, or by working on a rebalance. If an operation is delayed by helping an insertion or deletion to complete, then some operation has completed, as desired.

Therefore the key requirement is to show that rebalances cannot prevent operations from making progress. If a rebalance is forced to restart because of a successful insertion or deletion, then some other operation has completed.

A rebalance may also restart because of interference by a concurrent rebalance. In this case, however, since elements are moved only to the right, when the rebalance restarts it has (strictly) less work to do than in the aborted rebalance. In particular, the concurrent rebalance that forced a restart must have moved at least one of the items in the region to the right, thus reducing the amount of rabalancing necessary. Since every time a rebalance restarts it has less work to do, eventually the rebalance completes. □

6. CONCLUSIONS AND FUTURE WORK

This paper explores a range of issues for making cache-oblivious search structures concurrent. We consider both locking and lock-free solutions. Each of these approaches has practical and theoretical merits, and we make no judgement about which approach is best. A third approach, which we do not consider here, is to use transactional memory to support concurrency. This approach may lead to simpler coding and several new data-locality issues.

Like many previous concurrency studies, this paper analyzes uniformly random insertions. This analysis does not reveal all the design principles that went into our data structures. In particular, in the exponential trees, the parameter α tunes the tradeoff between low-concurrency and high-concurrency performance of the tree. When α is larger, serial operations in the tree are faster; when α is smaller, the tree supports increased concurrency and reduced contention. Moreover, the randomized nature of the tree reduces contention at high levels in the tree, by temporally spacing out updates, even when insertions are adversarial.

This paper focuses on correctness issues more than performance issues. In particular, in order to get performance guarantees, there are different insertion patterns (such as adversarial), different process models (such as different speeds and changing speeds), and different models of memory (such as queuing on memory locations for both reads and writes). Some techniques from the design of overlay networks may to carry over to these models.

A natural extension of the one-way packed-memory structure from Section 4 is to implement the structure as circular array. Using the circular array may lead to less memory allocation. While a circular implementation is straightforward for the serial and locking cases, it is more complex in the lock-free setting.

Acknowledgments

The authors gratefully acknowledge Maurice Herlihy, Victor Luchangco, and Mark Moir for suggesting this problem.

7. REFERENCES

[1] A. Aggarwal, J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[2] A. Andersson. Faster deterministic sorting and searching in linear space. In *FOCS'96*, pp. 135–141, 1996.

[3] A. Andersson, M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *STOC'2000*, pp. 335–342, 2000.

[4] L. Arge, J. S. Vitter. Optimal dynamic interval management in external memory. In *FOCS'96*, pp. 560–569, 1996.

[5] R. Bayer, E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[6] M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, A. López-Ortiz. The cost of cache-oblivious searching. In *FOCS'2003*, pp. 271–282, 2003.

[7] M. A. Bender, R. Cole, R. Raman. Exponential structures for efficient cache-oblivious algorithms. In *ICALP'2002*, pp. 195–207, 2002.

[8] M. A. Bender, E. Demaine, M. Farach-Colton. Cache-oblivious B-trees. In *FOCS'2000*, pp. 399–409, 2000.

[9] M. A. Bender, E. Demaine, M. Farach-Colton. Cache-oblivious B-trees. *SIAM J. Comput.*, 2005. To appear.

[10] M. A. Bender, Z. Duan, J. Iacono, J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *SODA 2002*, pp. 29–38, 2002.

[11] M. A. Bender, Z. Duan, J. Iacono, J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *J. of Alg.*, 3(2):115–136, 2004.

[12] M. A. Bender, M. Farach-Colton, B. C. Kuszmaul, J. Sukha. Cache-oblivious b-trees for optimizing disk performance. Manuscript., 2005.

[13] G. S. Brodal, R. Fagerberg, R. Jacob. Cache oblivious search trees via binary trees of small height. In *SODA 2002*, pp. 39–48, 2002.

[14] D. Comer. The ubiquitous B-Tree. *Computing Surveys*, 11:121–137, 1979.

[15] M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran. Cache-oblivious algorithms. In *FOCS'99*, pp. 285–297, 1999.

[16] M. P. Herlihy, J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, 1990.

[17] A. Itai, A. G. Konheim, M. Rodeh. A sparse table implementation of priority queues. In S. Even, O. Kariv, editors, *ICALP'81*, vol. 115 of *LNCS*, pp. 417–431, 1981.

[18] P. Jayanti. A complete and constant time wait-free implementation of CAS from LL/SC and vice versa. In *DISC'98*, vol. 1499 of *LNCS*, 1998.

[19] Z. Kasheff. Cache-oblivious dynamic search trees. MS thesis, Massachusetts Institute of Technology, Cambridge, MA, 2004.

[20] I. Katriel. Implicit data structures based on local reorganizations. MS thesis, Technion – Isreal Inst. of Tech., Haifa, 2002.

[21] R. E. Ladner, R. Fortna, B.-H. Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *Experimental Algorithmics: From Algorithm Design to Robust and Efficient Software*, vol. 2547 of *LNCS*, pp. 78–92, 2002.

[22] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[23] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, theorem 5, pp. 198–199. Springer-Verlag, 1984.

[24] M. M. Michael. Practical lock-free and wait-free ll/sc/vl implementations using 64-bit cas. In *DISC 2004*, vol. 3274 of *LNCS*, 2004.

[25] J. I. Munro, T. Papadakis, R. Sedgewick. Deterministic skip lists. In *SODA'92*, pp. 367–375, 1992.

[26] H. Prokop. Cache-oblivious algorithms. MS thesis, Massachusetts Institute of Technology, Cambridge, MA, 1999.

[27] N. Rahman, R. Cole, R. Raman. Optimised predecessor data structures for internal memory. In *Proc. of the 5th Intnl. Workshop on Algorithm Engineering*, vol. 2141 of *LNCS*, pp. 67–78, 2001.