Cache-Oblivious Streaming B-trees

Michael A. Bender
Dept. of Computer Science
Stony Brook University
Stony Brook, NY 11794-4400
bender@cs.sunysb.edu

Martin Farach-Colton
Dept. of Computer Science
Rutgers University
Piscataway, NJ 08855
farach@cs.rutgers.edu

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

Yonatan R. Fogel
Dept. of Computer Science
Stony Brook University
Stony Brook, NY 11794-4400
yfogel@cs.sunysb.edu

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

Jelani Nelson
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139
minilek@mit.edu

ABSTRACT

A ***streaming B-tree*** is a dictionary that efficiently implements insertions and range queries. We present two cache-oblivious streaming B-trees, the ***shuttle tree***, and the ***cache-oblivious lookahead array*** (***COLA***).

For block-transfer size B and on N elements, the shuttle tree implements searches in optimal $O\left(\log_{B+1} N\right)$ transfers, range queries of L successive elements in optimal $O\left(\log_{B+1} N + L/B\right)$ transfers, and insertions in $O\left((\log_{B+1} N)/B^{\Theta(1/(\log\log B)^2)} + (\log^2 N)/B\right)$ transfers, which is an asymptotic speedup over traditional B-trees if $B \geq (\log N)^{1+c/\log\log\log^2 N}$ for any constant $c > 1$.

A COLA implements searches in $O(\log N)$ transfers, range queries in $O(\log N + L/B)$ transfers, and insertions in amortized $O((\log N)/B)$ transfers, matching the bounds for a (cache-aware) buffered repository tree. A partially deamortized COLA matches these bounds but reduces the worst-case insertion cost to $O(\log N)$ if memory size $M = \Omega(\log N)$. We also present a cache-aware version of the COLA, the ***lookahead array*** , which achieves the same bounds as Brodal and Fagerberg's (cache-aware) B$^\varepsilon$-tree.

We compare our COLA implementation to a traditional B-tree. Our COLA implementation runs 790 times faster for random insertions, 3.1 times slower for insertions of sorted data, and 3.5 times slower for searches.

Categories and Subject Descriptors

F.2.3 [**Analysis of Algorithms and Problem Complexity**]: Tradeoffs between Complexity Measures—*Machine-independent complexity*; E.1 [**Data Structures**]: Trees

This research was supported by NSF grants CCF-0540897, CCF-0541097, CCF-0541209, CCF-0621439, CCF-0621425, CCF-0621511, CNS-0627645, CCF-0634793, and CCF-0632838; the US Air Force; Google; and Intel.

SPAA'07, June 9–11, 2007, San Diego, California, USA.

General Terms

Algorithms, Performance, Design, Experimentation, Theory

Keywords

Cache-Oblivious B-Tree, Buffered Repository Tree, Cascading Array, Shuttle Tree, Lookahead Array, Deamortized

1. INTRODUCTION

The ***B-tree*** [4,14] is the classic external-memory-dictionary data structure.[1] The B-tree is typically analyzed in a two-level memory model, called the ***Disk Access Machine (DAM)*** model [1]. The DAM model assumes an internal memory of size M organized into blocks of size B and an arbitrarily large external memory. The cost in the model is the number of transfers of blocks between the internal and external memory.

An N-node B-tree supports searches, insertions, and deletions in $O(\log_{B+1} N)$ transfers and supports scans of L contiguous elements in $O(1 + L/B)$ transfers. An important characteristic of the B-tree is that it is provably optimal for searching within the DAM model.

In fact, there is a tradeoff between the cost of searching and inserting in external-memory dictionaries [10], and B-trees achieve only one point on this tradeoff. Another point is achieved by the ***buffered-repository tree (BRT)*** [12]. The BRT supports the same operations as the B-tree, but searches use $O(\log N)$ transfers and insertions use amortized $O((\log N)/B)$ transfers. Thus, searches are slower in the BRT than in the B-tree, whereas insertions are significantly faster.

More generally, Brodal and Fagerberg's data structure from [10], which we call the ***B$^\varepsilon$-tree***, spans a large range of this tradeoff: For $0 \leq \varepsilon \leq 1$, the B$^\varepsilon$-tree supports insertions in amortized $O((\log_{B^\varepsilon+1} N)/B^{1-\varepsilon})$ transfers and searches in $O(\log_{B^\varepsilon+1} N)$ transfers. Thus, when $\varepsilon = 1$ it matches the performance of a B-tree, and when $\varepsilon = 0$, it matches the performance of a BRT. An interesting intermediate point is when $\varepsilon = 1/2$; then searches are slower by a factor of roughly 2, but insertions are faster by a factor of roughly $\sqrt{B}/2$ when compared with a B-tree.

This paper explores this insert/search tradeoff in the ***cache-oblivious*** (***CO***) model [15]. The CO model is similar to the DAM model, except that the block size B is *unknown* to the coder or to the algorithm and therefore cannot be used as a tuning parameter. The

[1]Most B-tree implementations are, in fact, B$^+$-trees [4, 14, 17], in which the full keys are all stored in the leaves, but for convenience we refer to all variations as "B-trees."

B-tree, buffered-repository tree, and B$^\varepsilon$-tree are not cache oblivious; they are parametrized by B.

There already exist several cache-oblivious dictionaries. The most well-studied is the ***cache-oblivious B-tree*** [6, 7, 11]. The CO B-tree supports searches in $O(\log_{B+1} N)$ transfers, insertions in amortized $O(\log_{B+1} N + (\log^2 N)/B)$ transfers, and range queries returning L elements in $O(\log_{B+1} N + L/B)$ transfers.[2] Another cache-oblivious dictionary is a cache-oblivious alternative to the BRT, which we call here the ***lazy-search BRT*** [2]. Although it is useful in some contexts (such as cache-oblivious graph traversal) the lazy-search BRT is unsatisfactory in two crucial ways: keys are assumed to be in the range $[1,N]$, and searches are heavily amortized, so that the whole cost of searching is charged to the cost of previous insertions. Indeed, any given search might involve scanning the entire data structure.

Results

The paper introduces several cache-oblivious dictionaries that illustrate several points in the insertion/search tradeoff.

Shuttle tree

The ***shuttle tree***, our main result, retains the asymptotic search cost of the CO B-tree while improving the insert cost. Specifically, searches still take $O(\log_{B+1} N)$ transfers, whereas insertions are reduced to amortized $O\left((\log_{B+1} N)/B^{\Theta(1/(\log\log B)^2)} + (\log^2 N)/B\right)$ transfers. This bound represents a speedup as long as $B \geq (\log N)^{1+c/(\log\log\log N)^2}$, for any constant $c > 1$; this inequality typically holds for external-memory applications. Range queries returning L elements take $O(\log_{B+1} N + L/B)$ transfers, which is asymptotically optimal.

This relatively complex expression for the cost of inserts can be understood as follows: When the dominant term in the insertion cost is $O\left((\log_{B+1} N)/B^{\Theta(1/(\log\log B)^2)}\right)$, insertions run a factor of $O\left(B^{1/(\log\log B)^2}\right)$ faster in the shuttle tree than in a B-tree or CO B-tree. Observe that this speedup of $B^{\Theta(1/(\log\log B)^2)}$ is superpolylogarithmic and subpolynomial in B. This speedup, while nontrivial, is not as large as the speedup in the B$^\varepsilon$-tree.

Lookahead array

We give another data structure, which we call a ***lookahead array***. The lookahead array is reminiscent of static-to-dynamic transformations [9] and fractional cascading [13]. This data structure is parametrized by a ***growth factor***. If the growth factor is chosen to be B^ε, then the lookahead array is cache aware and achieves the same amortized bounds as the B$^\varepsilon$-tree. If the growth factor is chosen to be a constant such as 2, then the lookahead array is cache-oblivious and matches the performance of the BRT. We call this version the ***cache-oblivious lookahead array*** (***COLA***). Unlike the BRT, the COLA is amortized, and any given insertion may trigger a rearrangement of the entire data structure.

For disk-based storage systems, range queries are likely to be faster for a lookahead array than for a BRT because the data is stored contiguously in arrays, taking advantage of inter-block locality, rather than stored scattered on blocks across disk. This is the same reason why the cache-oblivious B-tree can support range queries nearly an order of magnitude faster than a traditional B-tree; see, e.g., [8].

Deamortized lookahead array

We show how to deamortize the lookahead array and the COLA when $M = \Omega(\log N)$. Thus, we obtain the first cache-oblivious alternative to the BRT. There is no amortization on searches and the worst-case cost for an insert is no more than the cost of a search.

Experiments

We next measure how fast the COLA runs in comparison to a B-tree. We use the B-tree whose performance was described in [8]. For out-of-core data, the COLA was 790 times faster than the B-tree for random inserts, 3.1 times slower for sorted inserts, and 3.5 times slower for searches.

Map

The rest of this paper is organized as follows. In Section 2 we present the shuttle tree. In Section 3 we present the lookahead array, the cache-oblivious lookahead array, and their partial deamortizations. In Section 4 we present results from our implementation study of lookahead arrays.

2. SHUTTLE TREE

This section describes the ***shuttle tree***. We begin by describing the overall pointer structure of the tree. We next give the cache-oblivious layout of the shuttle tree followed by a search and space analysis. We then describe insertions and explain how to maintain our cache-oblivious layout dynamically in a structure called a ***packed-memory array*** [6]. Finally, we give an analysis of insertions and their effects on the dynamic layout.

Shuttle-tree structure

A shuttle tree is a ***strongly weight-balanced search tree*** [3, 18] (SWBST) with enhancements. In addition to the regular tree structure, each non-leaf node also points to a linked list of *buffers*. These buffers are in turn recursively defined to be shuttle trees. All pointers in the structure are bidirectional to help the shuttle tree adjust to changes in the memory layout.

We now introduce terminology useful for presenting SWBSTs and shuttle trees. The ***weight*** of a node u in a tree, denoted $w(u)$, is the total number of descendants of u plus one; that is, $w(u) = \sum_{v \in children(u)} w(v) + 1$, with $w(u) = 1$ if u is a leaf. The depth of a node u is u's distance from the root. SWBSTs have leaves all at the same depth. For such trees, we define the ***height*** of node u, denoted $h(u)$, to be the distance to a leaf plus one, i.e., $h(u) = h(v) + 1$ for every child v of u, and $h(u) = 1$ if u is a leaf.

A ***SWBST*** is a balanced search tree that maintains the following invariant: For fanout parameter $c > 1$ and for any node v, $w(v) = \Theta(c^{h(v)})$.

This invariant determines the balancing routine for insertions and deletions. To insert a new element, first perform a search and then insert the element in the appropriate leaf. (Since each node has size $\Theta(c)$ the leaf node can accommodate several inserts directly into the node before violating the balance condition.) If the leaf is full, then split into two leaves. This split increases the weight of all ancestor nodes in the tree. If ancestor v has weight $w(v)$ that is above threshold, then split v into two separate nodes, v' and v'', and then divide v's children to spread the weight as evenly as possible among v' and v''. Thus, v's ancestors gain in weight by an additional one. This splitting process may trickle up the tree until the balance condition is satisfied at every node along the leaf-to-root path followed. When the root node splits, a new root is added

[2] In fact, by using scanning structures such as [5] and amortizing the cost of range queries, the $\log^2 N$-term can be reduced or removed. However, the details and resulting structure are not directly pertinent to this paper.

above the old root node, thereby increasing the height of the tree by one.

There are several consequences of this invariant and balancing strategy. (See e.g. [3, 6] for full proofs.)

LEMMA 1. *Consider an N-node weight-balanced tree with constant balance parameter c.*
(1) The degree of any node is $\Theta(c)$.
(2) For any node u and constant $d \le h(u)$, the number of descendants of u that have height at least $h(u)-d$ is $\Theta(c^d)$.
(3) Suppose that a node split has cost 1. Then the amortized cost to insert into the tree is the search cost plus $O(1)$.
(4) Suppose that splitting a node u costs $\Theta(c^{h(u)})$. Then the amortized cost to insert into the tree is the search cost plus $O(\log N)$.

The shuttle tree supports inserts efficiently by using the buffers. The buffers work in much the same way as in a BRT—an element being inserted starts at the root, follows the appropriate root-to-leaf path, but pauses at buffers along the way. The element only gets "shuttled" down the tree when buffers overflow (and hence are full enough to amortize the cost of crossing block boundaries). Our shuttle tree differs from the BRT in that it has a linked list of buffers associated with each child pointer (rather than a single buffer as in the BRT). These buffers have doubly-exponentially increasing size.

To insert into a shuttle tree, start by inserting into the root node. To insert into a node in the tree, simply find the appropriate child pointer and insert into the corresponding linked list of buffers by inserting into the smallest buffer. When a buffer "overflows" (i.e., the height of the buffer shuttle tree exceeds the to-be-specified maximum), take each element in the buffer and insert it into the next (larger) buffer in the list.[3] Once the largest buffer in the list overflows, insert these items into the child node in the shuttle tree. (Thus, data items in the shuttle tree live in two possible places, either in some buffer on a root-to-leaf path or in a leaf of the tree.) When an inserted element reaches a leaf in the shuttle tree, insertions work in roughly the same way as in any SWBST with splits trickling up the tree. Note that at the time a node u splits, the buffers in between u and u's parent have just been flushed.

LEMMA 2. *When an element is inserted into a leaf ℓ, all nodes on the path from the root to ℓ can be fetched without increasing the asymptotic complexity, as long as $M = \Omega(B \log N)$.*

Proof. The reason we insert into a leaf is because its parent's buffer has just overflowed, thus the grandparent buffer has just overflowed, and so on up the tree to the root. Thus, we just flushed buffers all the way down. If any subsequent block transfers evict the root-to-leaf path, we can charge the cost of replacing the relevant path block to the cost of evicting it in the first place. □

We base our buffer sizes on Fibonacci numbers. Let F_k be the kth Fibonacci number. Then $F_0 = 0$, $F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$. For all positive integers h, we define the ***Fibonacci factor*** of h, denoted by $\xi(h)$, as follows. If h is a Fibonacci number, then $\xi(h) = h$. Otherwise, let f be the largest Fibonacci number less than h. Then the Fibonacci factor of h is $\xi(h) = \xi(h-f)$. The buffer sizes of a node at height $h+1$ depend upon $\xi(h)$. In particular, consider a node u at height $h+1$ in the tree, and let k be such that $F_k = \xi(h)$. We define the ***buffer-height-index function*** $\mathcal{H}(j) = j - \lceil 2\log_\varphi j \rceil$, where $\varphi \approx 1.618$ is the golden ratio. Then u has buffers with heights $F_{\mathcal{H}(j)}$, for each integer j, $j = \Theta(1), \ldots, k-1, k$.[4] In other words, there are roughly k buffers increasing geometrically in their heights, and the largest buffer has height $F_{\mathcal{H}(k)} = F_{k-2\lceil \log_\varphi k \rceil}$. These settings mean that the parent node of a subtree containing roughly K nodes has the largest buffer of size roughly $K^{1/\Theta((\log\log K)^2)}$.

The shuttle tree as specified thus far cannot yet be analyzed in the cache-oblivious setting. To do so, we must enforce a particular dynamic layout in memory. We must show that the layout permits efficient operations and can be efficiently maintained.

Shuttle-tree layout

We lay out the shuttle tree recursively in a type of "van Emde Boas (vEB) layout" [20] that takes into account the lists of buffers and several additional complications.

We first explain how our vEB layout would proceed on a regular tree of height h. Let F_k be the largest Fibonacci number strictly smaller than h. Then we split the tree at height F_k (roughly h/φ instead of at height $h/2$ as in the traditional vEB layout). That is, if $h = F_k$ is the kth Fibonacci number, then we split the tree into a root subtree of height F_{k-2} and leaf subtrees of height F_{k-1}, which are recursively laid out. It is important that the split is above the halfway point, height $h/2$, unlike in previous cache-oblivious search structures [6–8, 11, 20]. Fibonacci numbers are a convenient way to ensure this requirement because they enforce some integrality while roughly matching $F_k \approx \varphi^k$.

We now give the vEB layout of the shuttle tree, which means also laying out the buffers; see Figure 1. Consider a (sub)tree of height F_{k+1}, with leaves of this tree having buffers of heights (geometrically increasing) up to $F_{\mathcal{H}(k+1)}$. Think of this subtree and these buffers as a single entity, which we call a ***recursive subtree***. When laying out this recursive subtree, we split the subtree at height F_k. In the "left" end of memory, we store the top recursive subtree of height F_{k-1} (which includes leaf buffers of height up to $F_{\mathcal{H}(k-1)}$) recursively. To the right of this subtree, we store the height-$F_{\mathcal{H}(k)}$ leaf buffers, from left-to-right in the same order as the leaves. To the right of these buffers, we lay out each of the bottom recursive subtrees of height F_k (including leaf buffers of height up to $F_{\mathcal{H}(k)}$) recursively. To the right of each of the bottom recursive subtrees, we lay out that subtree's height-$F_{\mathcal{H}(k+1)}$ leaf buffers. We call the (contiguous) height-F_k recursive subtree and height-$F_{\mathcal{H}(k+1)}$ buffers (which appear immediately after the recursive subtree in the layout) a (height-F_k) ***buffered recursive subtree***.

The leaves of the shuttle tree are special in that they do not have any buffers. We call a recursive subtree containing leaves of the top level (i.e., entire) tree a ***leaf recursive subtree***. The recursive layout of a leaf recursive subtree is the same, except that the bottom recursive subtrees do not have any buffers coming out of the leaves. For convenience, we use the terms "recursive subtree" and "buffered recursive subtree" as generalizations of "leaf recursive subtree," even though the leaves do not have buffers.

The buffer heights are carefully chosen using the Fibonacci factors to match this recursive layout. It is always the case that when splitting a tree at the proper height, the leaf nodes of a height-F_k top or bottom recursive subtree have a height-$F_{\mathcal{H}(k+1)}$ buffer that can be stored after the recursive subtree (as indicated in Figure 1). The exception is for buffers that would have a sufficiently small constant height; these buffers are omitted altogether. (The elimination of small buffers helps the analysis in Lemma 16.)

Another way to interpret buffer sizes is that a node has a buffer for every (sufficiently large) recursive subtree in which it is a leaf. Thus, nodes that are roots of height-2 or taller recursive subtrees (i.e., those having Fibonacci factors > 1) have no buffers, because they cannot also be leaves of recursive subtrees. This notion is captured by the following lemma, which can be proved by induction.

[3] These items are inserted in arrival order, not smallest to largest.

[4] We can start j at any sufficiently large constant to help the proofs, in particular Lemma 16.

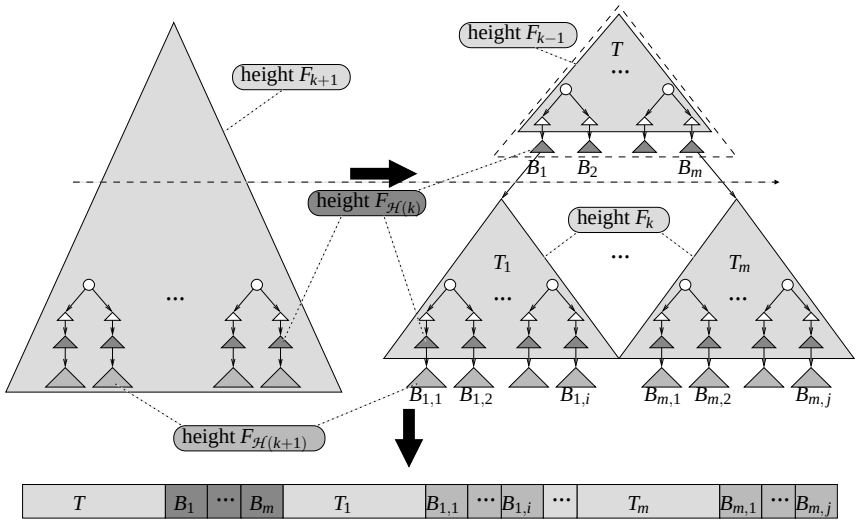

Figure 1: The recursive layout of the shuttle tree. The solid triangles encapsulate "recursive subtrees," where the leaves (drawn as circles) of a height-F_k recursive subtree have buffers up to size $F_{\mathcal{H}(k)}$. The dashed triangle encapsulates a "buffered recursive subtree" (which contains the recursive subtree and an additional, larger buffer for each leaf). The circles are leaf nodes of the tree. They are drawn with degree two here, but in fact they (and all nodes) have degrees that vary between two constants. The rounded rectangles indicate the heights of various recursive subtrees. The recursive definition is given from the top left to the top right figure—the tree is split at the largest Fibonacci number less than the height, and the largest buffers of each resulting subtree's leaves fall out. The array on the bottom gives the layout—first the top recursive subtree, then the largest leaf buffers, followed by each bottom recursive subtree with its largest leaf buffers.

LEMMA 3. *Consider a node at height $h+1>1$ in a shuttle tree (i.e,. a non-leaf node). This node is the leaf of some height-F_{k-1} recursive subtree if and only if $\xi(h) \geq F_k$.*

Search analysis

We now analyze the search cost.

LEMMA 4. *For a shuttle tree of height F_k and fanout c, which contains $N = \Theta(c^{F_k})$ elements, the worst-case search cost is $O(F_k/\log B) = O(\log_B N)$.*

Proof. Without loss of generality, suppose the height of the tree is a Fibonacci number (if not, we can round up). In particular, suppose the height is F_{k+1}, the $(k+1)$st Fibonacci number. Then we recursively lay out the tree by splitting it into recursive subtrees of height F_{k-1} (top) and F_k (bottom).

Let $T(F_{k+1})$ be the cost to search a shuttle tree of height F_{k+1}. Then we have $T(F_{k+1}) = T(F_{k-1}) + T(F_k) + T(F_{\mathcal{H}(k)}) + T(F_{\mathcal{H}(k+1)}) = T(F_{k-1}) + T(F_k) + T(F_{k-\lceil 2\log_\varphi k\rceil}) + T(F_{(k+1)-\lceil 2\log_\varphi(k+1)\rceil})$. The $T(F_{\mathcal{H}(k)})$ and $T(F_{\mathcal{H}(k+1)})$ terms arise because of the cost of recursively searching the (noncontiguous) buffers. This recurrence only deals with the largest buffer at each level, but Lemma 3 implies that the smaller buffers are correctly accounted for at further recursive levels. Once a recursive subtree fits in $O(1)$ blocks, the cost to search the subtree is $O(1)$ memory transfers. Thus, we have $T(F_{\Theta(\log\log B)}) = O(1)$.

We claim that $T(F_k) \leq (1/\log B)(c_1 F_k - c_2 F_k/\log_\varphi F_k)$, for some constants $c_1, c_2 > 0$. This claim can be proved by induction on k. □

Space usage

The following lemma claims that the space used by the shuttle tree and all its recursive buffers is linear in the number of elements stored in the shuttle tree. The main idea is that "most" of the buffers in the tree are very small, so they have little impact on space.

LEMMA 5. *An n-node shuttle tree uses $O(n)$ space.*

Proof. As an inductive hypothesis, we claim that an n-node shuttle tree having height F_k uses at most $d_1 n - d_2 c^{F_{\mathcal{H}(k)}}$ space, for some constants $d_1, d_2 > 0$. We prove this claim by induction on height.

Consider an n-node shuttle tree having height F_{k+1}. Split the tree at height F_k leaving a single top recursive subtree of height F_{k-1} and many bottom recursive subtrees of height F_k. Let ℓ be the number of bottom/leaf recursive subtrees.

The top recursive subtree contains $O(\ell)$ nodes ($\ell = \Theta(c^{F_{k-1}})$ from Lemma 1), each having buffers with heights not exceeding $F_{\mathcal{H}(k)}$ (and hence containing $O(c^{F_{\mathcal{H}(k)}})$ nodes). Thus, the space used by the top recursive subtree is $O(\ell c^{F_{\mathcal{H}(k)}})$.

In total, the ℓ bottom recursive subtrees use $d_1 n - \ell d_2 c^{F_{\mathcal{H}(k)}}$ space by assumption. Making d_2 large enough to dominate the constant (hidden by the big-O) for the space used by the top recursive subtree gives a total space usage of $d_1 n - \ell c^{F_{\mathcal{H}(k)}}$. Since $\ell = \Theta(c^{F_{k-1}})$, we are left with total space $d_1 n - \Theta(c^{F_{k-1}} c^{F_{\mathcal{H}(k)}})$. For k such that $F_{k-1} \geq F_{\mathcal{H}(k)} + \log_c d_2$, which is true for k larger than some constant, we have $c^{F_{k-1}+F_{\mathcal{H}(k)}} \geq c^{(F_{\mathcal{H}(k)}+\log_c d_2)+F_{\mathcal{H}(k)}} = d_2 c^{2F_{\mathcal{H}(k)}} \geq d_2 c^{F_{\mathcal{H}(k+1)}}$. Thus, the total space usage is at most $d_1 n - d_2 c^{F_{\mathcal{H}(k+1)}}$, which proves the claim.

We have proven that an n-node, height-F_{k+1} shuttle tree uses $O(n)$ space. To prove the lemma, we must extend the proof to all heights h. This extension is similar to the above proof—it is simply a matter of recursively dividing the tree at the largest Fibonacci number less than h. The bottom recursive subtrees use linear space, and we amortize the additional space used by the top recursive subtree against the bottom. □

Although a full shuttle tree uses only linear space, a *recursive subtree* uses super-linear space. (A *leaf* recursive subtree, on the other hand, is just a shuttle tree of the same size.) To understand this fact, notice that the number of nodes in the recursive subtree is dominated by the number of leaves. Leaves of recursive subtrees

have superconstant-size buffers. The following corollary places a weak (but sufficient) upper bound on the size of a recursive subtree.

COROLLARY 6. *The total space used by a height-F_{k-1} buffered recursive subtree is $O(c^{F_k})$.*

Proof. A height-F_{k-1} buffered recursive subtree contains at most $O(c^{F_{k-1}})$ nodes (and leaves) according to Lemma 1. Each of these nodes has buffers of size no larger than $F_{\mathcal{H}(k)}$. The space used by buffers is dominated by these largest buffers, which each take $O(c^{F_{\mathcal{H}(k)}})$ space by Lemma 5. Thus, the total space used is $O(c^{F_{k-1}}c^{F_{\mathcal{H}(k)}}) = O(c^{F_{k-1}+F_{k-\lceil 2\log_\varphi k\rceil}}) = O(c^{F_k})$. □

Maintaining layout dynamically

Insertions cause splits, which create new shuttle-tree nodes. Because splits change the topology of the tree, they also affect the vEB layout. The layout described thus far for the shuttle tree gives a total order in memory for nodes (and buffers) of the shuttle tree. Here, we show how to update this total order dynamically. The next subsection deals with how to make space for the new elements.

LEMMA 7. *Consider a split of a node u into two nodes u_1 and u_2 (where u_2 is the newly created node). Let $\mathcal{U}$, $\mathcal{U}_1$, and $\mathcal{U}_2$ be the largest buffered recursive subtrees for which u, u_1, and u_2 are roots, respectively. Then splitting u causes $\mathcal{U}$ to be replaced by $\mathcal{U}_1$ and $\mathcal{U}_2$ in the vEB layout. The new layout has the following properties:*

1. *For any two nodes or buffers $v_1, v_2 \notin \mathcal{U}$, the relationship between v_1 and v_2 does not change in the layout.*
2. *For any two nodes $v_1, v_2 \in \mathcal{U}_1$ (resp. $\mathcal{U}_2$), the relationship between v_1 and v_2 does not change in the layout.*
3. *All nodes and buffers in $\mathcal{U}_1$ immediately precede all those in $\mathcal{U}_2$ after the split.*
4. *Suppose u (and hence u_1 and u_2) is a leaf in a height-F_k recursive subtree, for $k \geq \Theta(1)$.[5] Then u_2 has a height-$F_{\mathcal{H}(k+1)}$ buffer. This newly created buffer appears immediately after u_1's height-$F_{\mathcal{H}(k+1)}$ buffer.*

Proof. These claims follow directly from the recursive definition of the layout. The last is most surprising and follows from Lemma 3 and the fact that u_1 and u_2 are sibling leaves in all relevant recursive subtrees. □

COROLLARY 8. *Consider a split of buffered recursive subtree $\mathcal{U}$ into two buffered recursive subtrees $\mathcal{U}_1$ and $\mathcal{U}_2$. This split can be performed in $O(1)$ linear scans of $\mathcal{U}$.*

Proof. Just do a "stable" (i.e., order-preserving) partition of $\mathcal{U}$ around the root of $\mathcal{U}_2$. □

The following corollary puts a weak upper bound on the cost of updating the layout to include new buffers.

COROLLARY 9. *Consider a split of a node u into two nodes u_1 and u_2. Suppose also that u is the leaf of a height-2 or larger recursive subtree. Let $\mathcal{U}$ be the largest buffered recursive subtree in which u (and hence u_1 and u_2) is a leaf. Then all of u_2's newly created buffers can be inserted with $O(1)$ linear scans of $\mathcal{U}$.*

Proof. All of u_1's buffers reside in $\mathcal{U}$, and u_2's buffers appear immediately after u_1's buffers. □

When a split causes a new node u_2 to be inserted, this node has some buffers associated with it (depending on its child's Fibonacci factor). On node creation, we allocate enough space for full buffers, and insert them in the appropriate place according to Lemma 7. As in Corollary 9, our algorithm need not be more clever than scanning the smallest buffered recursive subtree that includes all of u_2's buffers (i.e., the largest in which u_2 is a leaf).

When the root r of the entire shuttle tree is split, a new root r' is inserted above the previous root r. This new root r' may be the leaf of (possibly large) recursive subtrees. These recursive subtrees are conceptual and have not been filled out yet, but we still allocate all buffers. Consider, for example, if the height of r is F_k. Then r' has height $F_k + 1$ and includes buffers of height up to $F_{\mathcal{H}(k)}$. The new root r' and all its buffers simply precede all other nodes and buffers in the shuttle tree.

LEMMA 10. *The amortized cost of dynamically updating the layout due to an insertion at a leaf in a shuttle tree containing N items is $O((\log N)/B)$, assuming that the leaf-to-root path (and $\Theta(1)$ blocks adjacent to each block on the path) are already in memory.*

Proof. There are two components to the cost. First, we examine scans caused by splitting the appropriate subtrees. Then we analyze scans caused by inserting buffers.

Suppose that a node u at height h is split. Corollary 8 states that the cost of splitting this node is just that of scanning $\mathcal{U}$, where $\mathcal{U}$ is the largest recursive subtree for which u is a root. If $\mathcal{U}$ is a leaf recursive subtree (i.e., h is a Fibonacci number), then the size of $\mathcal{U}$ is $O(c^h)$ from Lemma 5. Otherwise, $\mathcal{U}$ has height geometrically smaller than h, and Corollary 6 states that $\mathcal{U}$ has size $O(c^h)$. (Note that $\mathcal{U}$ may be much smaller than $O(c^h)$, but we can afford to round up.) Thus, we pay for a scan costing $O(\lfloor c^h/B \rfloor)$ every $\Omega(c^h)$ inserts, for an amortized cost of $O(1/B)$ to insert below u.[6] The floor function is added because a constant number of blocks in this region are already in memory.

When a node u at height h has a split that creates new buffers, we insert the buffers in the correct location according to Lemma 7. Corollary 9 states that inserting these buffers involves at most a linear scan of the largest buffered recursive subtree in which u is a leaf. The size of this subtree is again $O(c^h)$ from Corollary 6. We have, therefore, that the scan from inserting buffers costs $O(\lfloor c^h/B \rfloor)$ every $\Omega(c^h)$ inserts for an amortized cost of $O(1/B)$ to insert below u.

We complete the proof by observing that an insert into a leaf inserts below $\Theta(\log N)$ nodes in the shuttle tree, which means we have to multiply all costs by $\Theta(\log N)$. □

Making space for insertions

The shuttle-tree layout suggests a total order in memory, and we maintain this layout dynamically by embedding the shuttle tree in a packed-memory array (PMA) [6]. The PMA is simply an array that allows for efficient insertions (amortized $O(\log^2 N/B)$ block transfers) by leaving gaps between elements. A nice property of the PMA is that any n consecutive elements use only $\Theta(n)$ space, so we can maintain our data structure compactly in memory.

When there is no more space available for an insert, a section of the array must be ***rebalanced***, or evenly spread out. In particular, when there is no room for an insert, we must search for a region of the array that is not "too dense." Details of the density thresholds can be found in [6].

This idea of maintaining a weight-balanced tree embedded in a PMA was described in the original cache-oblivious B-tree (in the conference version of [6]). The difficulty with bidirectional pointers is that when nodes shift around in the PMA, we have to update

[5] For small k, there are no buffers.

[6] This sort of analysis is similar to that for Lemma 1.

all pointers that point to these nodes. This updating may lose data locality. In particular, when a node moves, it must tell its children to update their parent pointers, and the children may not be nearby and may not be near each other.

To minimize the parent-pointer-update cost, we take advantage of the fact that we can be flexible in choosing the region to rebalance. In particular, Katriel [16] shows that the region can grow in either direction as long as we perform a density-threshold test whenever the region size roughly doubles. The cache-oblivious string B-tree [8] also leverages the flexible rebalance to deal with a tree embedded in a PMA.

A buffer is allocated as a single chunk $\mathcal{C}$ in the PMA $\mathcal{A}$, i.e., all space that the buffer will ever need is allocated at the outset. A rebalance of a shuttle tree moves this enclosed buffer chunk $\mathcal{C}$ as a single unit.

Thus, the PMA $\mathcal{A}$ can be thought of as a PMA containing variable-size elements (chunks). A PMA with variable-size elements also occurs in the string B-tree [8], and the PMA bounds are asymptotically unchanged.

The chunk $\mathcal{C}$ is itself a (recursive) PMA, imposing density thresholds and rebalance regions on the elements in $\mathcal{C}$. Specifically, when inserting into a buffer shuttle tree, the ensuing rebalance touches only memory within that tree's preallocated chunk $\mathcal{C}$, and thus never moves any nodes in the enclosing PMA $\mathcal{A}$. This property that insertions into a buffer do not affect nodes in the enclosing tree is necessary for the analysis and is the reason for preallocating buffers.

There are two costs to bound when analyzing the PMA insert. The first is the normal PMA cost for the amount of space inserted. The second is the cost of parent/child pointer updates due to node movement. We now bound the first cost by bounding the (amortized) amount of space inserted due to a single leaf insert. Note that until an item reaches the leaf, it does not increase the amount of space used.

LEMMA 11. *Each insert into the leaf of a shuttle tree increases the amount of space used by the shuttle tree by amortized $O(1)$.*

Proof. Consider a split of a node u at height $h+1$, let k be such that $F_k \le h < F_{k+1}$, and let j be such that $F_j = \xi(h)$. Then the split creates new buffers having heights up to $F_{\mathcal{H}(j)} \le F_{\mathcal{H}(k)}$. Since u's buffers have geometrically increasing heights, the size of the buffers is dominated by the largest buffer, which has size $O(c^{F_{\mathcal{H}(k)}})$ from Lemma 5. We amortize against the $\Omega(c^{h+1})$ items that must be inserted before u is split to get an amortized space increase of $O(c^{F_{\mathcal{H}(F_k)}}/c^{h+1}) = O(c^{F_{k-2}-h}) = O(c^{-h/2})$. Taking the sum over all heights gives a geometric sum, bounded by a constant. □

We now give more information about inserting extra space. A split determines how much space to insert into the PMA, specifically the size of one node plus the size of the extra buffers inserted (which on average is constant from Lemma 11). We first insert the extra space into the PMA so that the changes in the dynamic layout (a new node and buffers) have space available to them. Once there is space available, we can just do the split. It is convenient to insert a large block of space in one spot in the PMA and spread it out with a scan. Recall that when inserting new buffers, our analysis already accounts for a scan of the buffered recursive subtree containing those new buffers (Corollary 9 and Lemma 10), so this "spreading out" does not asymptotically increase the cost.

The following lemma states that an algorithm for choosing a "good" rebalance region exists. A good region is one for which there are not too many nodes outside the region having parent pointers into the region. (That is, the number of pointers into the region is polynomially smaller than the size of the region.) Nodes having child pointers into the region have some locality, so we can update those easily, and we do not consider them further in the lemma argument.

LEMMA 12. *There exists an algorithm for growing the rebalance region such that when inserting after a node at height h, with k such that $F_k < h \le F_{k+1}$, we have the following properties:*
(1) The initial size of the region is $O(c^h)$.
(2) The initial region contains all nodes and buffers involved in the dynamic layout changes (i.e., from Lemma 7).
(3) There are $O(c^{F_{k-1}})$ nodes outside the region having parent pointers to nodes inside the region.
(4) Each time we increase the region size by a multiplicative constant, we find another feasible rebalance region.

Proof. We give an algorithm for choosing a rebalance region. Property (3) is the difficult property, so we concentrate on how to satisfy it in particular.

Suppose that we need space next to the node u in the shuttle tree (because u splits into u_1 and u_2). Suppose that height $h(u)$ is not a Fibonacci number, and let k be such that $F_k < h(u) < F_{k+1}$. (Thus, u is not the root of a leaf recursive subtree—the case when u is the root is the easier case, which we handle later.) Let $\mathcal{U}$ be the height-F_{k-1} buffered recursive subtree containing u_1 and u_2. (This subtree contains nodes of height between F_k+1 and F_{k+1}). Let $\mathcal{L}_1, \mathcal{L}_2, \ldots$ be $\mathcal{U}$'s children leaf recursive subtrees. (These subtrees $\mathcal{U}$ and $\mathcal{L}_1, \mathcal{L}_2, \ldots$ are contiguous in the layout.) Then we initially select our rebalance region to include $\mathcal{U}$ and $\mathcal{L}_1$.

This initial rebalance region satisfies properties (1) through (3). In particular, Corollary 6 states that $\mathcal{U}$ uses $O(c^{F_k}) = O(c^h)$ space, and Lemma 5 states that $\mathcal{L}_1$ uses $O(c^h)$ space. Thus, the size of the region is $O(c^h)$, satisfying property (1).

Property (2) is satisfied because we have selected a buffered recursive subtree $\mathcal{U}$ containing u_1 and u_2 for which these nodes are not the root. This region, therefore, includes the buffered recursive subtrees involved in the dynamic layout changes (i.e., in Lemma 7).[7]

The only nodes outside the region having parent pointers to nodes inside the region are the roots of $\mathcal{L}_2, \mathcal{L}_3, \ldots$. All of these pointers point to nodes in $\mathcal{U}$. Thus, property (3) is satisfied because $\mathcal{U}$ has $O(c^{F_{k-1}})$ nodes.

We now show how to increase the region size (property (4)) while still respecting property (3). Recall that the region initially includes $\mathcal{U}$ and $\mathcal{L}_1$; we grow to the right to include $\mathcal{U}$'s children leaf recursive subtrees $\mathcal{L}_2, \mathcal{L}_3, \ldots$. Each time the region extends into the next leaf recursive subtree $\mathcal{L}_i$, it consumes all of $\mathcal{L}_i$. Since these subtrees all have size $\Theta(c^{F_k})$, we can select enough of them to increase the size of the rebalance region by a constant factor.

Observe that each time the rebalance region grows to include more leaf recursive subtrees $\mathcal{L}_i$, the number of nodes having parent pointers into the rebalance region actually reduces. We, therefore, maintain property (3).

If the rebalance region grows to include all such children subtrees $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3, \ldots$, then our rebalance region is itself a height-F_{k+1} leaf recursive subtree $\mathcal{L}'_j$. We now grow the region to include a neighboring height-F_{k+1} leaf recursive subtree $\mathcal{L}'_{j-1}$ or $\mathcal{L}'_{j+1}$, etc. Once the region consumes all such leaf recursive subtrees $\mathcal{L}'_1, \mathcal{L}'_2, \ldots$, then we grow to include the parent height-F_k buffered recursive subtree $\mathcal{U}'$. This process continues until a region meeting the proper density thresholds is discovered.

[7] Notice that this initial region may be much larger than the subtrees involved in the layout change. We make the initial region this large because it is simpler, and we can afford to do so.

Once the rebalance region consists of only leaf recursive subtrees, there are no nodes outside the region having parent pointers into the region, and thus property (3) is trivially satisfied. As the region grows, notice that it always consists of leaf recursive subtrees.

We now handle the case when $h(u) = F_k$ for some k. We initially choose a rebalance region to include the height-F_k leaf recursive subtrees rooted at u_1 and u_2. When this region needs to grow, it grows similarly, and there are no parent pointers into the region. □

We now analyze the cost of updating pointers in the shuttle tree. Note that pointer updates in the contiguous region being rebalanced can be resolved in a constant number of scans, and hence are amortized against the cost of making space in the PMA.

Instead, we examine the cost of updating parent pointers outside the rebalance region (pointing into it).

LEMMA 13. *Consider a rebalance of a region in the PMA due to an insert at the leaf of an N-element shuttle tree. Assume that the leaf-to-root path (and $\Theta(1)$ blocks adjacent to each block on the path) are already in memory. Then the amortized cost per insert of updating parent pointers from nodes outside the region pointing to nodes inside the region is $O((\log N)/B^{1/3})$.*

Proof. Lemma 12 states that the maximum number of parent pointers (outside the region) to update when splitting a node at height $F_k < h \leq F_{k+1}$ is $O(c^{F_{k-1}})$. We amortize these updates against the $\Omega(c^h)$ inserts that must be performed below a height-h node between splits, which gives an amortized cost of $O(c^{F_{k-1}-h})$ parent-pointer updates per insert.

Suppose that $c^{F_k} = \Omega(B)$. Then the leaf recursive subtrees do not fit in a block, and we must pay a block transfer for each parent-pointer update. This case results in an amortized memory-transfer cost of $O(c^{F_{k-1}-h}) = O(c^{F_{k-1}-F_k}) = O(c^{(2/3)F_k-F_k}) = O(c^{-F_k/3}) = O(B^{-1/3})$ per insert (for $k \geq 4$).

Suppose that $c^{F_k} = O(B)$. Thus, a leaf recursive subtree whose parent pointer must be updated fits in a block. In particular, $\Theta(B/c^{F_k})$ such subtrees fit in $\Theta(1)$ blocks, and we get $B/c^{F_k} \geq B/c^h$ parent-pointer updates per block transfer. We thus have an amortized cost of $O(c^{F_{k-1}-h}/(B/c^h)) = O(c^{F_{k-1}}/B) = O(c^{(2/3)F_k}/B) = O(B^{2/3}/B) = O(B^{-1/3})$ per insert (for $k \geq 4$).

Since an insert into a leaf inserts below $\Theta(\log N)$ nodes, we multiply our bound by $\log N$ to complete the proof. □

Insert analysis

We first bound the total amortized cost of an insert into a leaf node of the shuttle tree (i.e., the split cost and PMA inserts). Then we bound cost to recursively insert into the leaves of all (shuttle-tree) buffers. Finally, we bound insert cost to "shuttle" down a root-to-leaf path. We conclude the analysis with the full insert cost.

The following lemma bounds the cost to insert at a leaf. We assume that the memory is large enough to store a root-to-leaf path (i.e., $M = \Omega(B \log N)$).

LEMMA 14. *The amortized cost of inserting into a leaf of a size-N shuttle tree is $O((\log N)/B^{1/3} + (\log^2 N)/B)$.*

Proof. Inserting into a leaf that has room for more keys is essentially free (i.e., only the block containing the leaf must be in memory). There is only an extra cost when nodes split. Note that at the time of an insert into the leaf, Lemma 2 states that the leaf-to-root path (and a constant number of neighboring blocks) is in memory.

The costs incurred by an insert are the costs of updating the layout (i.e, splitting nodes and inserting new buffers), finding space in the PMA, and updating pointers.

We first bound the cost of making space in the PMA. Lemma 11 says that the average amount of space inserted is $O(1)$, so making space amortizes to $O(\log^2 N/B)$ per insert. In fact, we choose a larger initial rebalance region than the PMA dictates. This additional cost is amortized against the layout update.

We now bound the cost updating the dynamic layout. Lemma 10 states that the cost of splitting nodes and inserting buffers amortizes to $O(\log N/B)$.

Finally, we bound the cost of parent-pointer updates. Lemma 13 states that the cost of updating parent pointers outside the rebalanced region amortizes to $O(\log N/B^{1/3})$. Updating parent pointers within a rebalance region comes for free with a constant number of scans.

Adding all these costs together gives the lemma. □

The next lemma bounds the number of buffers of each size along a root-to-leaf path in the shuttle tree. We then apply this lemma to show that the cost of recursively inserting into all of these buffers is small.

LEMMA 15. *Consider the nodes along a root-to-leaf path of a height-F_k shuttle tree. At most F_{k-j+2} of these nodes have height-$F_{\mathcal{H}(j)}$ (or larger) buffers.*

Proof. We count the number of nodes u along a root-to-leaf path with $\xi(h(u)-1) \geq F_j$.

A node only has a height-$F_{\mathcal{H}(j)}$ buffer if its child has Fibonacci factor at least F_j. Thus, the number of nodes with Fibonacci factor at least F_j is an upper bound on the number of nodes with height-$F_{\mathcal{H}(j)}$ buffers.

Let $N(k, j)$ be the number of nodes along the root-to-leaf path of a height-F_k shuttle tree such that the nodes have Fibonacci factor at least F_j. We claim that $N(k, j) = F_{k-j+2}$, and we prove this claim by induction on k.

The claim is true for $k = 3$ and j with $1 < j \leq k$—the tree has height $F_3 = 2$. For $j = 3$, $F_{3-j+2} = F_2 = 1$ indicates that one node (the root) has Fibonacci factor at least $F_3 = 2$. For $j = 2$, $F_{3-j+2} = F_3 = 2$ indicates that two nodes (the root and leaf) have Fibonacci factor $F_2 = 1$.

Suppose that the claim is true for all $k' < k$ and all $j \leq k'$. Notice that $N(k,k) = 1 = F_{k-k+2}$ because only the root of a height-F_k tree has Fibonacci factor F_k. For $j = k-1$, only $F_{k-(k-1)+2} = 2$ nodes have Fibonacci factor at least F_{k-1}. For $j \leq k-2$, observe that for any node u with height $h(u) > F_{k-1}$, $\xi(h(u)) = \xi(h(u) - F_{k-1})$ (by the definition of Fibonacci factor). Thus, the nodes of these height have exactly the same Fibonacci factors as the nodes in a height-F_{k-2} subtree.

Thus, by inductive assumption on the values of k and $j \leq k-2$, we obtain the recurrence $N(k, j) = N(k-1, j) + N(k-2, j) = F_{(k-1)-j+2} + F_{(k-2)-j+2} = F_{k-j+2}$. □

We now bound the cost to insert into *all* leaves, including those in the (recursive) shuttle-tree buffers. Note that this lemma does not account for bringing the root-to-leaf path of the shuttle tree (or the recursive shuttle-tree buffers) into memory. The cost of transferring these blocks will be accounted for in Theorem 17.

LEMMA 16. *The amortized cost of (recursively) inserting into the leaves of all shuttle-tree buffers along the root-to-leaf path of a size-N shuttle tree is $O((\log N)/B^{1/3} + (\log^2 N)/B)$.*

Proof. Without loss of generality, suppose that the height of the tree is the Fibonacci number F_k, giving $N = O(c^{F_k})$.

Let $T(F_k)$ be the amortized cost of inserting into the leaf of a height-F_k shuttle-tree and all leaves of shuttle-tree buffers (recursively) along the root-to-leaf-path. Let $C(F_k) \leq c_1(F_k/B^{1/3} +$

F_k^2/B) be the cost of inserting into the leaves of a particular height-F_k shuttle tree (not counting the cost to recursively insert into buffers—this cost is bounded in Lemma 14).

Applying Lemma 15 gives the recurrence $T(F_k) \leq C(F_k) + \sum_{j=b}^{k} F_{k-j+2} T(F_{\mathcal{H}(j)})$, where $b = \Theta(1)$ is the smallest buffer-height index. Thus, we have $T(F_k) \leq C(F_k) + \sum_{j=b}^{k} F_{k-j+2} T(F_{\mathcal{H}(j)}) = C(F_k) + \sum_{j=0}^{k-b} F_{k-j+2-b} T(F_{\mathcal{H}(j+b)}) \leq C(F_k) + \sum_{j=0}^{k-b} (F_{k-1}/F_{j-3+b}) T(F_{\mathcal{H}(j+b)})$.

We make the inductive hypothesis that $T(F_i) \leq c_2(F_i/B^{1/3} + F_i^2/B)$ for all $i < k$, which can be proved by substitution into the summation. □

The next theorem bounds the total cost of an insert into the shuttle tree. The main idea of this theorem is that whenever an item moves from one block to another, at least $B^{\Theta(1/(\log\log B)^2)}$ items also move, and we can amortize against these to get a small cost of moving down the tree.

THEOREM 17. *The amortized cost of an insert into the shuttle tree of size N is $O((\log_B N)/B^{\Theta((1/\log\log B)^2)} + (\log^2 N)/B)$.*

Proof. The full insert cost has two contributors: the cost of letting an item move toward the leaves in the tree(s), and the cost of inserting into the leaves. The latter is bounded by Lemma 16.

Here we analyze the cost of moving down the tree. This cost follows a similar recurrence to that in Lemma 4's proof. In particular, we let $T(F_{k+1})$ be the cost to move an item through a height-F_{k+1} shuttle tree. Then we have $T(F_{k+1}) = T(F_{k-1}) + T(F_k) + T(F_{\mathcal{H}(k)}) + T(F_{\mathcal{H}(k+1)})$.

This proof differs from that in Lemma 4 because of a different base case. Once we recurse down to buffered recursive subtrees of size roughly B, we have a subtree with leaves containing buffers of size roughly $B^{\Theta(1/(\log\log B)^2)}$. The only reason an item being inserted enters a new block is if it overflows from one of these buffers in a previous block. Thus, we can amortize the cost of this move to get a base case of $T(F_{\Theta(\log\log B)}) = O(1/B^{\Theta(1/(\log\log B)^2)})$. The recurrence solves in the same way as in the proof of Lemma 4. □

Scanning

We conclude this section by analyzing the scan cost.

THEOREM 18. *Suppose that $M = \Omega(B\log N)$. Suppose also that we just performed a search on an element x. Then a range query starting at x that returns L elements in sorted order makes $O(L/B)$ memory transfers.*

Proof. Finding the successor of an element x entails finding x's successor in the shuttle tree and recursively in all the buffers along the root-to-leaf path in the shuttle tree. Assuming all (recursive) root-to-leaf paths are in memory, a successor query is free. A new block must only be brought in every $\Omega(B)$ successor queries, for an amortized cost of $O(1/B)$.

Lemma 4 states that a search costs $O(\log_B N)$ block transfers. Thus, assuming memory has size $M = \Omega(B\log_B N)$, all (recursive) root-to-leaf paths are in memory. Finding a successor is, therefore, amortized to $O(1/B)$. □

3. CACHE-OBLIVIOUS LOOKAHEAD ARRAY (COLA)

This section describes the ***cache-oblivious lookahead array (COLA).*** We begin by describing the structure of the COLA with an amortized analysis. Then we show how to deamortize the COLA to provide better worst-case bounds. We explain the deamortization in two steps, showing first the basic idea, and then dealing with the full complexity of the data structure. We conclude this section with a discussion of the tradeoff between updates and queries.

COLA structure and amortized analysis

The cache-oblivious lookahead array (COLA) is similar to the binomial list structure [9] of Bentley and Saxe. It consists of $\lceil \log_2 N \rceil$ arrays, or ***levels***, each of which is either completely full or completely empty. The kth array is of size 2^k and the arrays are stored contiguously in memory. The COLA maintains the following invariants:

1. The kth array contains items if and only if the kth least significant bit of the binary representation of N is a 1.
2. Each array contains its items in ascending order by key.

To maintain these invariants, when a new item is inserted we effectively perform a carry. That is, we create a list of length one with the new item, and as long as there are two lists of the same length, we merge them into the next bigger size.

LEMMA 19. *Insertion into the COLA incurs an amortized $O((\log N)/B)$ block transfers.*

Proof. Once an item has been involved in $O(\log N)$ merges, it is in the last array. The first $\log_2 B$ moves incur no cost because these arrays fit into a constant number of blocks, which we can assume are always kept in memory. After this, the arrays being merged have length at least B. Thus, the cost of merging two such arrays of length k is $O(k/B)$. The amortized cost of one merge is $O(1/B)$ per item, and so the total amortized merge cost is $O((\log N)/B)$ block transfers. □

Naïvely, searches can be implemented by binary searching separately in each of the $O(\log N)$ levels for a total complexity of $O(\log^2 N)$. We call this first data structure the ***basic COLA***.

We speed up searches by fractional cascading [13]. Every eighth element in the $(k+1)$st array also appears in the kth array, with a ***real lookahead pointer*** to its location in the $(k+1)$st array. Each fourth cell in the kth array is reserved for a ***duplicate lookahead pointer***, which holds pointers to the nearest real lookahead pointer to its left and right. Thus, every level uses half its space for actual items and half for lookahead pointers.

LEMMA 20. *COLA searches incur $O(\log N)$ block transfers.*

Proof. To simplify the proof, on each level store $-\infty$ and $+\infty$ in the first and last cell and give them real lookahead pointers.

We prove inductively that a search for key r looks at at most eight contiguous items in each level and that r is greater than or equal to the smallest of these and less than or equal to the largest of these. This induction will establish both the time bound and correctness of the search procedure.

The claim is true in the first three levels, because each has size at most eight (even with the $-\infty$ and $+\infty$ pointers added). Suppose the claim is true at the kth level, where $k \geq 3$, and the search in this level looked at contiguous items with keys $r_1 < r_2 < \ldots < r_8$ and that $r_1 \leq r \leq r_8$.

Let ℓ be such that $r_\ell \leq r < r_{\ell+1}$. If $r_\ell = r$ then we have found the target element or lookahead pointers that lead to the target element. In this first case the induction goes through trivially for all remaining levels. Otherwise, $r_\ell < r$. In this second case we restrict our search on the next level to those keys between the elements pointed to by two lookahead pointers, the two lookahead pointers whose key values are the maximum below r and the minimum above r.

We can find both lookahead pointers quickly using the duplicate lookahead pointers. □

Lookahead pointers can also be used to achieve $O(\log N)$ block transfers for predecessor and successor queries and $O(\log N + L/B)$ block transfers for range queries, where L is the number of items reported.

Deamortization of basic COLA

As a first step, we show how to partially deamortize the basic COLA (which includes no lookahead pointers) when $M = \Omega(\log N)$. We improve the worst-case bound from $O(N/B)$ to $O(\log N)$. We ignore the issue of deamortizing global rebuilding after the data structure doubles in size because this can be handled using standard methods [19, Ch. 5].

The idea is as follows. In level k of the lookahead array we maintain two arrays each of size 2^k. Whenever a level contains items in both of its arrays, we begin merging those two arrays into an empty array in the next level. Each level is said to be either ***safe*** or ***unsafe***. Informally, a level is unsafe during merging. More formally, initially all levels are safe, level k become unsafe once it contains exactly 2^{k+1} items, and an unsafe level becomes safe when both of its arrays become empty.

We place newly inserted items into level 0. We then scan the levels from left to right, merging from unsafe levels to the next level, stopping when we have either run out out of unsafe levels or moved a total of m items, whichever comes first (m to be chosen later). We keep three arrays of size $\lceil \log_2 N \rceil$. One array keeps track of which levels are safe. The other two arrays' kth entries hold the indices we are at in the merge of the two arrays at level k.

The bigger m, the more aggressively we merge. We need to make sure we merge aggressively enough to guarantee that whenever we merge from a level there is at least one free array in the next level, which would be implied by the next level being safe. The insertion worst-case bound is clearly $O(m)$, so we want m as small as possible while still keeping us aggressive enough. Lemma 21 shows that a fairly low setting of m suffices.

LEMMA 21. *If a lookahead array contains k levels, setting $m = 2k+2$ guarantees that two adjacent levels are never simultaneously unsafe.*

Proof. By induction on levels we show that levels i and $i+1$ are never simultaneously unsafe. For $i = 0$ there are at least 2 levels, so $m \geq 6$ and we can afford to perform an entire merge from level 1 to 2 whenever level 1 becomes unsafe. Thus level 1 never remains unsafe, showing the base case.

We show that if level ℓ is unsafe, it becomes safe before level $\ell - 1$ becomes unsafe. When level ℓ becomes unsafe, level $\ell - 1$ is empty. Furthermore, for us to have moved items from level $\ell - 1$ to level ℓ, all previous levels had to have been safe since we process unsafe levels from left to right. Therefore, levels 0 to $\ell - 2$ hold a total of at most $\sum_{j=0}^{\ell-2} 2^j = 2^{\ell-1} - 1$ items, and for level $\ell - 1$ to become unsafe again there must be at least $2^{\ell-1} + 1$ inserts. We show that level ℓ becomes safe after at most $2^{\ell-1}$ inserts. After $2^{\ell-1}$ inserts, we have accumulated at least $m\ell 2^{\ell-1}$ moves, and no move will go unused as long as level ℓ is unsafe.

After $2^{\ell-1}$ insertions there can be at most $2^\ell - 1$ items in levels $\ell - 1$ and below, and each one could have used at most $\ell - 2$ moves to get to its level. Thus, items below level ℓ could have used a total of at most $(\ell - 2)(2^\ell - 1)$ moves. We want at least $2^{\ell+1}$ moves available to move all the items from level ℓ to the next level, so we want $m2^{\ell-1} \geq (\ell - 2)(2^\ell - 1) + 2^{\ell+1}$, which holds for $m \geq 2\ell$. □

THEOREM 22. *The basic COLA can be deamortized to perform $O(\log N)$ block transfers per insertion in the worst case and $O((\log N)/B)$ amortized block transfers per insertion as long as $M = \Omega(\log N)$.*

Proof. The worst-case bound follows immediately from Lemma 21 since $k = O(\log N)$.

For the amortized bound, the argument is similar to that for the amortized COLA. The difference is that now when we incur a block transfer at a level beyond the $(\log_2 B)$th level, that block may be evicted due to previous levels becoming unsafe before we get a chance to move B items. We charge such an eviction to the block that is brought in. Since unsafe levels are processed from left to right, there can be no cycles in blocks charging one another for evictions. Furthermore, if a block C evicts a block D from a later level then is evicted itself before completing the merge it was brought in for, when C is later brought back into memory it must be because a block at an earlier level finished participation in its merge; C can take that block's space in memory. Thus, a block brought into memory during a merge is only charged for evicting at most one other block. This guarantees that the amortized cost of moving one item is still $1/B$.

Furthermore, since $M = \Omega(\log N)$, we can assume the three arrays keeping track of merge information are always in memory, so the insertion algorithm can determine which merges need to take place without incurring extra transfers. □

Deamortization of COLA

The introduction of lookahead pointers complicates COLA deamortization since we must maintain the pointers in the middle of merges to keep queries fast. Since the deamortized structure is only allowed to perform a small amount of work per update, it is not clear how this can be done. We show how to extend the deamortization technique for the basic COLA to the COLA in such a way as to completely hide the details of the deamortization from the queries; from the viewpoint of a query, no level will appear to be in the middle of a merge. The precise description and proofs follow.

At level k we now maintain three arrays, each of size 2^k. At least one array is a ***shadow array*** and at least one is a ***visible array*** (unless level k is beyond the levels being used by the data structure yet, in which case all three arrays are considered shadow). Query algorithms ignore the shadow arrays. This labeling of shadow versus visible is non-constant throughout time; during updates a shadow array may become visible, and vice versa. Items in visible arrays never appear to be in the middle of a merge from a query's viewpoint, so the data structure induced by only looking at visible arrays appears almost exactly as the amortized COLA with lookahead pointers. The key difference from the deamortization method of the previous section is that when level k becomes unsafe, we do not move items from level k to $k+1$; we instead *copy* them to a shadow array of level $k+1$. Thus, from the point of view of a query, no arrays appear to be in the middle of a merge.

All levels start as being safe and all arrays start as being shadow arrays. Level k becomes unsafe once two of its arrays become full. Two full arrays at an unsafe level must be merged into any shadow array A of level $k+1$. If there is more than one shadow array in level $k+1$, preference is given to one already containing lookahead pointers. After the merge, lookahead pointers are copied from A into an empty shadow array at level k. We restrict ourselves to having a total combined number of copied lookahead pointers and merged items to being at most m during each insertion, m to be chosen later. Level k becomes safe once both the merge into A and the placement of lookahead pointers from A into level k are

complete. At this point we say the array in level k that received lookahead pointers from A becomes ***linked*** to A.

We now discuss the transition from shadow to visible and vice versa. Level 0 only contains two arrays, both of which are always visible. A shadow array becomes visible when there is a sequence of linked arrays beginning at level 0 to that array. When a shadow array becomes visible there are two cases. If there are two other visible arrays in that level, their statuses are both changed to shadow and both arrays are then considered empty. Otherwise, if there are 0 or 1 other visible arrays, they remain visible.

LEMMA 23. *Suppose level k becomes unsafe and merges into a shadow array A at level $k+1$. Then, A becomes visible before another merge into level $k+1$ occurs.*

Proof. We prove the claim by induction on level. If we merge from level 0 to an array A at level 1, the array A becomes immediately visible since an array at level 0 then becomes linked to A. Now, suppose we just merged two arrays from level k into array A at level $k+1$. At the end of the merge, level k has two arrays with items (the arrays that engaged in the merge) and one shadow array B that is linked to A. Since there are never two adjacent unsafe levels, B has received all its lookahead pointers from A before a merge into level k can occur. When a merge into level k does occur, it is into B, and the other two arrays at level k then become shadow. For another merge into level $k+1$ to occur, there must be another merge into level k after the merge into B in order to make level k unsafe. By our induction hypothesis, B becomes visible by this time, and thus so does A since B is linked to A. □

THEOREM 24. *The COLA with lookahead pointers can be deamortized to perform $O(\log N)$ block transfers per insertion in the worst case and $O((\log N)/B)$ amortized block transfers per insertion as long as $M = \Omega(\log N)$.*

Proof. Safeness of a level is similar to safeness in the basic COLA. In both scenarios, if level k is unsafe then we must scan $\Theta(2^k)$ items before level k becomes safe again (note that placing lookahead pointers in the previous level after a merge can be done by two simultaneous scans). Thus, we can use the proof idea of Lemma 21 to state that two adjacent levels are never unsafe if we choose to move $\Theta(\log N)$ items per insertion. By Lemma 23, three arrays per level suffice to guarantee that for any unsafe level there is at least one shadow array in the next level to merge into. The rest of the argument follows that of the proof of Theorem 22. □

We discuss how to slightly modify the query algorithms to work when the COLA has two arrays at each level. As with the basic COLA, each level k will actually have at most two arrays in use at any given time. At least one of these arrays will be the ***main array***. Should there only be one array in use at level k, we consider that array to be the main array. Otherwise, the array merged into least recently is the main array and the other is ***secondary***.

The main array contains lookahead pointers into both the main and secondary arrays of the next level, and the secondary array, if it exists, contains no lookahead pointers. We only use half the space of the secondary array to maintain that every array is exactly half-full with real items. When merging into what will become the main array of level $k+1$, every eighth element in this main array appears as a lookahead pointer in an array of level k. When merging into what will become the secondary array of level $k+1$, every sixteenth element of each of the main and secondary arrays of level $k+1$ appears as a lookahead pointer in an array at level k. Thus, main arrays always contain exactly half real items and half lookahead pointers. Duplicate lookahead pointers at level k point to the nearest real lookahead pointers to their left and right in each of the main and secondary arrays of level $k+1$. Queries then search both arrays at a level in parallel.

Cache-aware update/query tradeoff

With a few changes, it is possible to make the lookahead array cache-aware and achieve $O(\log_{B^\varepsilon+1} N)$ block transfers per query and $O((\log_{B^\varepsilon+1} N)/B^{1-\varepsilon})$ block transfers per insertion for any $\varepsilon \in [0,1]$, matching the bound of the B$^\varepsilon$-tree [10]. Instead of having the array at level k be twice as big as the array at level $k-1$, we can have it grow in size by some arbitrary multiplicative factor g which we call the ***growth factor***, e.g. $g = 2$ yields the COLA. We discuss setting $g = \Theta(B^\varepsilon)$. Instead of every eighth element of level k also appearing in level $k-1$, every $\Theta(B^\varepsilon)$th element will appear as a lookahead pointer in the previous level. During queries we must look through $\Theta(B^\varepsilon)$ instead of $\Theta(1)$ cells of each array, but $\Theta(B^\varepsilon)$ cells still fit in at most 2 blocks implying a constant number of block transfers per level. When performing an insertion, the level being merged into may not be empty and we thus have to merge the pre-existing items with the ones from the previous level. Since the sum of the sizes of the first $k-1$ levels is at least an $\Omega(1/B^\varepsilon)$ fraction of the size of the kth level, a level is merged into at most B^ε times before its items participate in a merge into a future level. This fact, together with there being at most $O(\log_{B^\varepsilon+1} N)$ levels, gives the amortized $O((\log_{B^\varepsilon+1} N)/B^{1-\varepsilon})$ insertion bound.

The deamortization technique of the last section can be adjusted to reduce the worst-case insertion complexity of this cache-aware lookahead array to $O(\log_{B^\varepsilon+1} N)$. The main differences are that we need to move $\Theta(B^\varepsilon \log_{B^\varepsilon+1} N)$ items per insertion instead of $\Theta(\log N)$ items, and that we need an extra array per level to provide space for holding the result of merges into non-empty arrays.

4. COLA: EXPERIMENTAL RESULTS

This section presents an experimental evaluation of the COLA. We compared COLAs with B-trees as well as the impact of insertion order on performance. First we describe our implementation, then the experiments, and present our results showing that the COLA can be orders of magnitude faster than traditional B-trees without sacrificing much performance on searches.

Implementation

Here we describe the implementation details of the amortized COLA from, parametrized by growth factor g (***g-COLA*** for short) and ***pointer density*** p. For pointer density p, each level ℓ includes an additional $\lfloor 2p(g-1)g^{\ell-1} \rfloor$ redundant elements (real lookahead pointers). We set $p = 0.1$ in all our experiments. For a g-COLA, a level receives $g-1$ merges before being merged into a higher level. Therefore level ℓ has size 1 for $\ell = 0$ and size $2(g-1)g^{\ell-1}$ for $\ell > 0$. When a level is not completely full (e.g., when it contains only redundant elements or has received fewer than $g-1$ merges) we maintain the elements right justified in their array. Elements comprise key/value pairs, where keys and values each are of size 64 bits. We pad the elements to a total size of 32 bytes. Instead using of duplicate lookahead pointers, each real element uses 64 of its padding bits to hold a copy of the closest real lookahead pointer to its left. Redundant elements use 64 of their padding bits to hold the real lookahead pointer. When performing merges, we merge the 2 smallest levels at a time as in the last paragraph of the proof of Lemma 19. We alternate placing the result of the merge at the beginning of the target level and at the newly freed space at the beginning of the data structure, thus requiring space for only 1 additional element during merges.

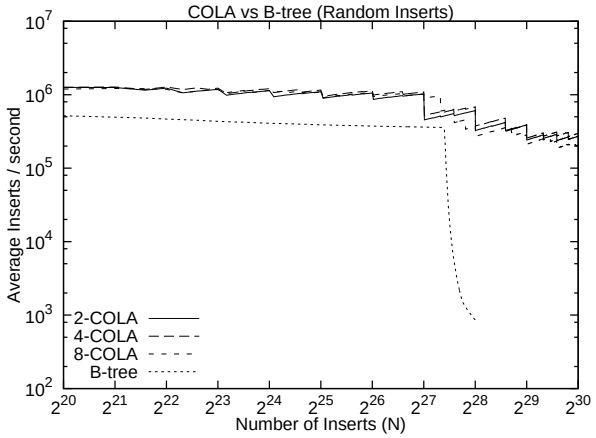

Figure 2: Data is inserted in random order. The 2-COLA is 790 times faster than the B-tree for $N = (256 \times 2^{20}) - 1$ (the largest N tested). Structures no longer fit in main memory when $N \approx 2^{27}$.

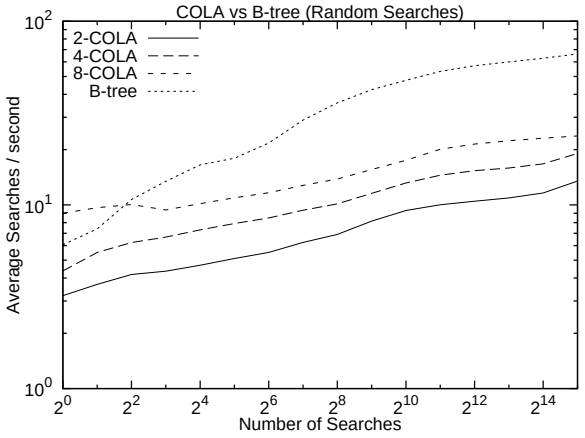

Figure 4: When $N = 2^{30} - 1$, the 4-COLA performs 2^{15} searches 3.5 times slower than the B-tree. Initial searches are slow due to the cache being empty. The source data was created from the test in Figure 3.

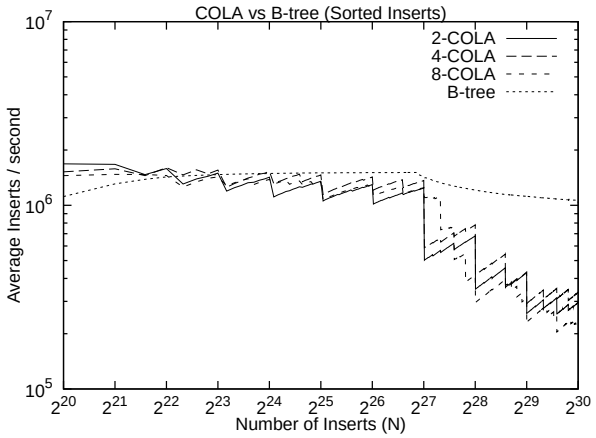

Figure 3: Data is inserted in sorted order, which gives best-case performance for the B-tree. The 4-COLA is 3.1 times slower than the B-tree for $N = 2^{30} - 1$.

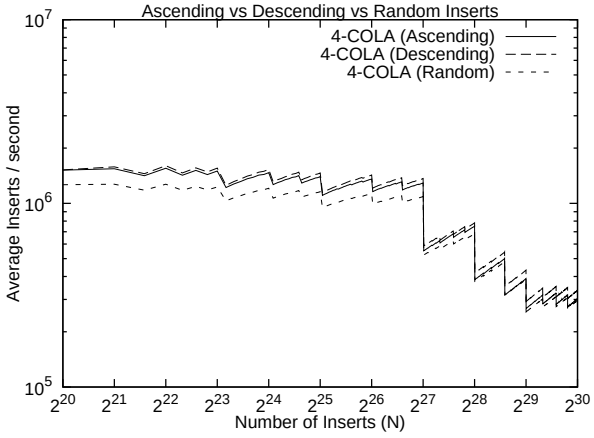

Figure 5: We measured the time to insert into COLAs for ascending, descending, and random keys. Inserting $2^{30} - 1$ keys sorted in descending order is 1.1 times faster than inserting in ascending order, and 1.1 times faster than inserting in random order.

This merge pattern requires $O(k)$ CPU time and $O(k/B)$ memory transfers to merge a total of k items across any number of levels. When distributing lookahead pointers after a merge we proceed level by level. The target level is scanned to copy pointers down one level, the next largest level is scanned to copy pointers down to the next level, and so on. We perform searches as in the proof of Lemma 20, except that we compute right-hand lookahead pointers on the fly by scanning subsequent levels.

Our B-tree implementation employs blocks of size 4KiB. Key and value sizes were each 64 bits to match our COLA implementation. Both data structures access external memory via memory mapping. As a sanity check, we compared the performance of our traditional B-Tree to the Berkeley DB [21], a high-quality commercially available B-tree. Berkeley DB supports variable-sized keys, crash recovery, and very large databases, none of which our implementation supports. The Berkeley DB with the default buffer-pool allocation is much slower than our implementation, but is comparable once the parameters are tuned, and logging is turned off, suggesting that we did a reasonable job implementing our B-tree.

All experiments were performed on a dual Xeon 3.2GHz machine with 2MiB of L2 Cache, 4GiB RAM and two 250GB Maxtor 7*L*250*S*0 SATA drives using software RAID-0 with 64KiB stripe width, running Linux 2.6.12-10-amd64-xeon in 64-bit mode.

Experiments

When performing the experiments we measured the time once every 2^{20} inserts. We measured user CPU time u, kernel CPU time k, and elapsed time since start of test w. We estimated disk time d as $d = w - u - k$. The RAID array contains only the memory-mapped file. Before measuring any times, we first created a large enough file to hold the entire experiment. We measured raw disk bandwidth of 120MiB/s by timing the write of 2^{37} bytes to the RAID array. We remounted the RAID array's file system before every insertion test to clear the file cache.

In all figures resulting from experiments, insert performance of the COLA appears to be periodic when plotted on a logarithmic scale because there is a merge of 2^k elements after every 2^k inserts.

We compared the B-tree to the COLA as follows. For $N = 2^{30} - 1$ we inserted keys $[N-1,\ldots,0]$ into a B-tree. We next attempted to insert $(256 \times 2^{20}) - 1$ random elements into a new B-tree, stopping after 87 hours at about 2^{28} insertions. We repeated the tests for a 2-, 4-, and 8-COLA, and completed the random insertion test of N elements. We were able to complete the insertions for the COLAs. After remounting the RAID array, we next searched in the g-COLAs and B-tree for 2^{15} random elements, measuring the time after search number 2^x, $0 \le x \le 15$. (To construct the complete B-tree we first sorted the N random elements then inserted them, since directly inserting the elements would have taken too long.)

As shown in Figure 2, the 4-COLA is 790 times faster than the B-tree for random inserts. Figure 3 shows that the 4-cola is 3.1 times slower for insertions where the insertions are performed in descending, rather than random, order. Figure 4 shows that the 4-cola is 3.5 times slower for searches.

We believe that the B-tree performs faster for sorted data because it only uses the leftmost root-to-leaf path, which can stay in memory. For random inserts, the B-tree loses the advantage of keeping its insertion path in memory. For random inserts, sequential prefetching of more than one block does not significantly help B-trees, but significantly helps COLAs. The 4-COLA is 1.1 times faster than the 2-COLA for random inserts, 1.1 times faster for sorted inserts, and 1.4 times faster for searches. The 4-COLA is 1.4 times faster than the 8-COLA for random inserts, 1.5 times faster for sorted inserts, and 1.2 times slower for searches. Given the superior tradeoff of the 4-COLAs, we use them for all subsequent experiments.

We next performed an experiment to measure COLA performance for different insertion patterns. For $N = 2^{30} - 1$ we inserted $[N-1,\ldots,0]$ into a COLA, $[0,\ldots,N-1]$ into a second COLA, and N random elements into a third COLA. Figure 5 shows that inserting keys sorted in descending order is 1.1 times faster than inserting keys sorted in ascending order, and 1.1 times faster than inserting random keys. Inserting keys sorted in ascending order was 1.02 times faster than inserting random keys.

We believe inserting keys in descending order is faster due to the final merge. The number of elements we merge at each level grows geometrically. The last merge, into the target level, is the largest. When inserting in descending order, elements already in the target level do not move, whereas when inserting in ascending order, all elements in the target level move to make room for elements from smaller levels.

Acknowledgments

We gratefully acknowledge Christopher Wright for his help in implementing, testing, and running experiments on the *g*-COLA.

5. REFERENCES

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

[2] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, pages 268–276, Montréal, Canada, Québec, Canada, May 2002.

[3] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6):1488–1508, 2003.

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

[5] M. A. Bender, R. Cole, E. D. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proc. 10th Annual European Symp. on Algorithms (ESA)*, pages 139–151, Rome, Italy, Sept. 2002.

[6] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM J. Comput.*, 35(2):341–358, 2005. An earlier version of this paper appeared in Proc. 41st Annual Symp. on Foundations of Computer Science (FOCS), pages 399-409, Redondo Beach, California, 2000.

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

[8] M. A. Bender, M. Farach-Colton, and B. Kuszmaul. Cache-oblivious string B-trees. In *Proc. 25th Symposium on Principles of Database Systems (PODS)*, pages 233–242, Chicago, Illinois, June 2006.

[9] J. L. Bentley and J. B. Saxe. Decomposable searching problems i: Static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[10] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 546–554, Baltimore, Maryland, May 2003.

[11] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 39–48, San Francisco, California, Jan. 2002.

[12] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 859–860, San Francisco, California, Jan. 2000.

[13] B. Chazelle and L. J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[14] D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[15] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symp. on Foundations of Computer Science (FOCS)*, pages 285–297, New York, New York, Oct. 1999.

[16] I. Katriel. Implicit data structures based on local reorganizations. Master's thesis, Technion, Israel Inst. of Tech., Haifa, May 2002.

[17] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1973.

[18] J. I. Munro, T. Papadakis, and R. Sedgewick. Deterministic skip lists. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 367–375, Orlando, Florida, January 1992.

[19] M. H. Overmars. *The Design of Dynamic Data Structures*. Springer, 1983.

[20] H. Prokop. Cache-oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Inst. of Tech., June 1999.

[21] Sleepycat Software. The Berkeley Database. `http://www.sleepycat.com`, 2005.