A Locality-Preserving Cache-Oblivious Dynamic Dictionary

Michael A. Bender*† Ziyang Duan*‡ John Iacono§ Jing Wu*¶

Abstract

This paper presents a simple dictionary structure designed for a hierarchical memory. The proposed data structure is *cache oblivious* and *locality preserving*. A cache-oblivious data structure has memory performance optimized for all levels of the memory hierarchy even though it has no memory-hierarchy-specific parameterization. A locality-preserving dictionary maintains elements of similar key values stored close together for fast access to ranges of data with consecutive keys.

The data structure presented here is a simplification of the cache-oblivious B-tree of Bender, Demaine, and Farach-Colton. Like the cache-oblivious B-tree, this structure supports search operations using only $O(\log_B N)$ block operations at a level of the memory hierarchy with block size B. Insertion and deletion operations use $O(\log_B N + \log^2 N/B)$ amortized block transfers. Finally, the data structure returns all k data items in a given search range using $O(\log_B N + \frac{k}{B})$ block operations.

This data structure was implemented and its performance was evaluated on a simulated memory hierarchy. This paper presents the results of this simulation for various combinations of block and memory sizes.

1 Introduction

The B-tree [10, 17, 22, 27] is the classic external-memory search tree, and it is widely used in both theory and practice. The B-tree is designed to support insert, delete, search, and scan on a two-level memory hierarchy consisting of main memory and disk. The basic structure is a balanced tree having a fan-out proportional to the disk-block size B. The B-tree uses linear space and its query and update performance are $O(\log_B N)$ memory transfers. This is a $O(\log B)$-factor improvement over the $O(\log_2 N)$ bound obtained by the RAM-model structures (e.g., [1, 21, 30, 33, 37, 38]). This improvement translates to approximately an order of magnitude speedup, depending on the application.

Although B-trees are in widespread use, they do have several limitations. They depend critically on the block size B and therefore are only optimized for two levels of the memory hierarchy. On the other hand, modern memory hierarchies often have many levels including registers, several levels of cache, main memory, and disk. Furthermore, the disparity in the access times of the levels is growing, and future memory hierarchies are likely to have even more levels each with their own parameters.

Theoretically, it is possible to create a multilevel B-tree, but the resulting structure is significantly more complex than the standard B-tree. On each level of interest, the data structure must be carefully tuned. Furthermore, the amount of wasted space in such an implementation appears exponential in the number of levels.

In many applications, such as database management systems, it is recognized that the classic implementation of a B-tree can be optimized for modern memory hierarchies by improving the data layout. For example, many systems heuristically attempt to group logically close pages physically near each other in memory in order to improve the performance of the scan operation [18, 25, 26, 9, 35].

These methods are still parameterized by the block size B and may not perform as well for suboptimal values of B.

The Cache-Oblivious Model. Traditionally most algorithmic work has been done in the *Random Access Model* (RAM) of computation, which assumes a "flat" memory with uniform access times. Recently, however, research has been performed on developing theoretical models for modern complicated hierarchical memory systems; see e.g. [2, 3, 4, 6, 36, 43, 44].

In order to avoid the complications of multilevel memory models, a body of work has focused on two-level memory hierarchies. In the most successful two-level model, the *Disk Access Model* (DAM) of Aggarwal and Vitter [5], the memory hierarchy consists of an internal memory of size M and an external memory partitioned into B-sized blocks. Motivated by the dominant role disk accesses typically play in an algorithm's total runtime, the performance metric in this model is the number of block transfers.

The main disadvantage of a two-level memory model is that the programmer must focus efforts on a

*Department of Computer Science, State University of New York, Stony Brook, NY 11794-4400, USA.

†*bender@cs.sunysb.edu*. Supported in part by HRL Laboratories, the National Science Foundation, and Sandia National Laboratories.

‡*dziyang@cs.sunysb.edu*.

§Department of Computer and Information Science, Polytechnic University, 5 Metrotech Center, Brooklyn NY 11201, USA. *jiacono@poly.edu*.

¶*jingwu@cs.sunysb.edu*.

particular level of the hierarchy, resulting in programs that are less flexible to different-scale problems and that are not portable.

Recently, a new model that combines the simplicity of the two-level models with the realism of more complicated hierarchical models was introduced by Frigo, Leiserson, Prokop, and Ramachandran [19, 31]. This model, called the *cache-oblivious model*, enables us to *reason* about a simple two-level memory model, but *prove results* about an unknown, multilevel memory model. The idea is to *avoid any memory-specific parameterization*, that is, to design algorithms that do not use any information about memory access times or block sizes.

The theory of cache-oblivious algorithms is based on the *ideal-cache model* [19, 32]. As in the DAM model, there are two levels in the memory hierarchy, which we call *cache* and *memory*, although they could represent any pair of levels. The main difference between the cache-oblivious and the DAM model is that parameters B and M are unknown to a cache-oblivious algorithm. This crucial difference forces cache-oblivious algorithms to be optimized for all values of B and M and therefore for all levels of the memory hierarchy.

The ideal-cache model assumes unlimited associativities and optimal replacement capability. Frigo, Leiserson, Prokop, and Ramachandran [19] showed that the ideal-cache model can be simulated by essentially any memory system with only a small constant-factor overhead.

Our Results. In this paper we propose a data structure that is both cache oblivious and locality preserving. Our structure is a simplification of the cache-oblivious B-tree of Bender, Demaine, and Farach-Colton [11]. At a level of the memory hierarchy with block size B, the number of block transfers during a search operation is $O(\log_B N)$, which is asymptotically optimal. Insertions and deletions take $O(\log_B N + \frac{\log^2 N}{B})$ amortized memory transfers, while scans of k data items are performed asymptotically optimally with $O(\frac{k}{B} + 1)$ memory transfers. If the scan operations is not supported, our structure can be modified using indirection, as in [11], so that all operations use $O(\log_B N)$ amortized block transfers.

Our data structure is locality preserving. Any range of k consecutive keys is stored in a contiguous region of memory of size $O(k)$. This layout can be advantageous in real-world architectures, where accessing sequential blocks can be an order of magnitude faster than accessing random blocks [20].

Our structure can be easily modified using the method of Brown and Tarjan [15] to achieve $O(\log_B k)$ query times, where k is the difference in rank between the current and previous queries. This property of our structure, known as the *dynamic-finger property*, implies other finger-type results [23]. For example, given a constant-size subset F of the keys in the structure, let $d(x, y)$ be the difference in rank between x and y. The number of page faults to access x is then $O(\log_B \min_{f\in F} d(f, x))$.

Our data structure consists of two arrays. One of the arrays contains the data and a linear number of blank entries, and the other contains an encoding of a tree that indexes the data. The search and update operations involve basic manipulations of these arrays.

We evaluated the algorithm on a simulated memory hierarchy. This paper presents the results of this simulation for various combinations of block and memory sizes.

Related Work. Bender, Demaine, and Farach-Colton developed a cache-oblivious dynamic dictionary [11], which achieves the same asymptotic performance as B-trees on all levels of the memory hierarchy without knowledge of the parameters defining the memory hierarchy. Their structure guarantees the same performance bounds as ours, except for finger-type results. In order to support finger-type results, their update cost increases to $O(\log_B N + \frac{\log^2 N \log B}{\sqrt{B}})$.

Subseqently, Rahman, Cole, and Raman [34] presented and implemented a cache-oblivious dynamic dictionary based on exponential search trees [7, 8, 39] that supports insert and delete in $O(\log_B N + \log\log N)$ memory transfers and supports finger-type results. Their structure does not allow the scan operation, unlike our structure. They provide the first implementation of a cache-oblivious dynamic dictionary, which performs better than an optimized B-tree, because it interacts well with the translation-lookaside buffer.

Brodal, Fagerberg, and Jacob [14] independently developed a simplified cache-oblivious search tree, whose bounds match those presented here. Their tree structure maintains a balanced tree of height $\log n + O(1)$, which they lay out cache-obliviously in an array of size $O(n)$. They also present experimental results comparing memory layouts of binary trees.

Cache-oblivious algorithms originated to solve non-data-structure problems. Frigo, Leiserson, Prokop, and Ramachandran [19] showed how several basic problems—namely matrix multiplication, matrix transpose, Fast Fourier Transform, and sorting—have optimal cache-oblivious algorithms. Optimal cache-oblivious algorithms have also been found for LU decomposition [13, 40]. Yi, Adve, and Kennedy provide compiler support for transforming loop structures into recursive structures, for better performance on hierarchical memories [49]. There has been an increased amount of evidence showing the practicality of cache-oblivious algorithms, most recently in [12, 19, 31, 34, 49].

Another body of related work shows how to keep N elements ordered in $O(N)$ locations of memory, subject to insertions and deletions. Itai, Konheim, and Rodeh [24] examine the problem in the context of priority queues and propose a simple structure using $O(\log^2 N)$ amortized time per update. Similar results were obtained by Melville and Gries [29] and by Willard [45]. Willard [46, 47, 48] examines the problem in the context of dense file maintenance and develops a more complicated structure using $O(\log^2 N)$ worst-case time per update. Rahman, Cole, and Raman [34] independently developed a locality preserving search tree in which all the leaves are stored in order in memory. Bender, Demaine, and Farach-Colton [11] show that a modification to the structure of Itai, Konheim, and Rodeh results in a packed-memory structure running in $O((\log^2 N)/B)$ amortized memory transfers per update and $O(k/B)$ memory transfers per traversal of k elements.

2 Description of the Structure

Our data structure maintains a dynamic set S storing items with key values from a totally ordered universe. It supports the following operations:

- INSERT(x): Adds x to S.
- DELETE(x): Removes x from S.
- PREDECESSOR(x): Returns the item from S that has the largest key value in S that is at most x.
- SCANFORWARD(): Returns the successor of the most recently accessed item in S.
- SCANBACKWARD(): Returns the predecessor of the most recently accessed item in S.

We use two separate cache-oblivious structures, the packed-memory structure of Bender, Demaine, and Farach-Colton [11] (which is closely based upon previous structures of Itai, Kronheim and Rodeh [24] and Willard [45, 46, 47, 48]) and the static B-tree of Prokop [32].

The packed-memory structure maintains N elements in sorted order in an array of size $O(N)$ subject to insertions, deletions and scans. Insertions and deletions require $O(\frac{\log^2 n}{B}+1)$ block transfers and a scan of k elements requires $O(\frac{k}{B}+1)$ block transfers.

The packed-memory structure is used to store the items. However it does not support efficient searches. (A naïve binary search requires $O(\log(\frac{N}{B}))$ memory transfers, which is prohibitively large.) We thus use the static cache-oblivious tree structure as an index into the packed-memory structure, where each leaf in the static cache-oblivious tree corresponds to one item in the array of the packed-memory structure.

The difficulty with this fusion of structures is that when we insert or delete, the positions of the elements in the packed-memory structure may be adjusted; thus invalidating the keys in the static B-tree. Thus, the static B-tree must be updated to reflect the changes in the packed-memory structure. We show that the cost of updating the static B-tree does not dominate the insertion cost.

Before describing our main structure, we present the packed-memory structure and the static cache-oblivious layout.

2.1 Packed-Memory Maintenance.

In a *packed-memory structure* [11], we have N totally ordered elements $x_1, x_2, \ldots, x_N$ to be stored in an array A of size $O(N)$. Two update operations are supported: a new element may be inserted between two existing elements, and an existing element may be deleted. This structure maintains the following invariants:

- *Order Constraint :* Element x_i precedes x_j in array A iff $x_i \prec x_j$.
- *Density constraint:* The elements are evenly distributed in the array A. That is, any set of k contiguous elements $x_{i_1}, \ldots, x_{i_k}$ is stored in a contiguous subarray of size $O(k)$.

The packed-memory structure of [11] has the following performance guarantees: Scanning any set of k contiguous elements $x_{i_1}, \ldots, x_{i_k}$ uses $O(k/B+1)$ memory transfers. Inserting or deleting a new element uses $O(\frac{\log^2 N}{B}+1)$ amortized memory transfers.

Described roughly, the packed-memory structure works as follows: When a window of the array becomes too full or too empty, we evenly spread out the elements within a larger window. The window sizes range from $O(\log N)$ to $O(N)$ and are powers of two. A window of size 2^i is a contiguous block of 2^i elements whose left element's position is a multiple of 2^i.

Associated with window sizes are *density thresholds*, which are guidelines to determine the acceptable densities of windows. The *upper-bound density threshold* of a window of size 2^k is denoted τ_k, where

$$\tau_{\log\log N} > \tau_{\log\log N+1} > \ldots > \tau_{\log N},$$

and the *lower-bound density threshold* of a window of size 2^k is ρ_k, where

$$\rho_{\log\log N} < \rho_{\log\log N+1} < \ldots < \rho_{\log N},$$

and $\tau_{\log N} > \rho_{\log N}$. The density of any particular window of size 2^k may deviate beyond its threshold, but as soon as the deviation is "discovered," the densities of the window of the array is adjusted to be within the threshold.

The values of the densities are determined according to an arithmetic progression. Specifically, let $\tau_{\log N}$ be any positive constant less than 1, and let $\tau_{\log\log N} = 1$. Let

$$\delta = (\tau_{\log\log N} - \tau_{\log N})/(\log N - \log\log N)\,.$$

Then, define density threshold τ_k to be

$$\tau_k = \tau_{\log\log N} - (k - \log\log N)\cdot\delta.$$

Similarly, let $\rho_{\log N}$ be any constant less than $\tau_{\log N}$, and let $\rho_{\log\log N}$ be any constant less than $\rho_{\log N}$. Let

$$\delta' = (\rho_{\log N} - \rho_{\log\log N})/(\log N - \log\log N)\,.$$

Then, define density threshold ρ_k to be

$$\rho_k = \rho_{\log\log N} + (k - \log\log N)\cdot\delta'.$$

Rebalancing. We say that a window of the array of size 2^k is *overflowing* if the number of data elements in the region exceeds $\tau_k 2^k$. We say that a window of the array of size 2^k is *underflowing* if the number of data elements in the region is less than $\rho_k 2^k$. To *rebalance* a window in the array we simply take the elements of the window and rearrange them to be spread out as evenly as possible.

Insertions and Deletions. To *insert* (*delete*) an element x at location $A[j]$, proceed as follows: Examine the windows containing $A[j]$ of size 2^k, for $k = \log\log N, \ldots, \log N$ until the first window is found that is not overflowing (underflowing). Insert (delete) the element x and rebalance this window.

2.2 Static structure. We review a cache-oblivious *static* tree structure due to Prokop [32]. Given a complete binary tree, we describe a mapping from the nodes of the tree to positions of an array in memory. This mapping, called *van Emde Boas layout*, resembles the recursive structure in the van Emde Boas data structure [41, 42]. The cache oblivious structure can perform any traversal from the root to a leaf in an N node tree in $\Theta(\log_B N)$ memory transfers, which is asymptotically optimal.

We now describe the van Emde Boas layout: Suppose the tree contains N items and has height $h = \log(N+1)$. Conceptually split the tree at the middle level of edges, below the nodes of height $h/2$. This split breaks the tree into the *top recursive subtree* A of height $\lfloor h/2 \rfloor$, and several *bottom recursive subtrees* $B_1, \ldots, B_k$ of height $\lceil h/2 \rceil$. Thus there are $k = O(\sqrt{N})$ bottom recursive subtrees and all subtrees contain $O(\sqrt{N})$ nodes. Mapping of the nodes of the tree into positions in the array is done by partitioning the array into $k = O(\sqrt{N})$ regions, one for each of $A, B_1, \ldots, B_k$. The mapping of the nodes in each subtree is determined by recursively applying this mapping. The base case is reached when the trees have one node.

We now introduce the notion of *levels of detail*, which partition the tree into disjoint recursive subtrees. In the finest level of detail, 0, each node is its own recursive subtree. The coarsest level of detail, $\lceil \log_2 h \rceil$, is just the tree itself. In general, level of detail k is derived by starting with the entire tree, recursively partitioning it as described above, and exiting the recursion whenever we reach a recursive subtree of height $\leq 2^k$. Note that according to the van Emde Boas layout, each recursive subtree is stored in a contiguous block of memory. At level of detail k, all recursive subtrees have heights between 2^{k-1} and 2^k.

Thus, the following lemma describes the performance of the Van Emde Boas layout.

Lemma 2.1. ([31]) *Consider an N-node complete binary search tree T that is stored in a van Emde Boas layout. Then a traversal in T from the root to a leaf uses at most $O(\log_B N)$ memory transfers.*

Proof. Let k be the coarsest level of detail such that every recursive subtree contains at most B nodes. Thus, every recursive subtree is stored in at most 2 memory blocks. Since tree T has height $\log(N+1)$, and the height of the subtrees will range from $\frac{\log B}{2}$ to $\log B$, the number of subtrees traversed from the root to a leaf will be at most $\frac{2\log N}{\log B} = 2\log_B N$. Since each subtree can be in at most 2 memory blocks, traversing a path from the root to a leaf uses at most $4\log_B N$ memory transfers. ∎

2.3 Dynamic Cache-Oblivious Structure. Our dynamic cache-oblivious locality-preserving dictionary uses the packed-memory structure to store its data, and uses the static structure as an index into the packed-memory structure. From this point on we use the term "array" to refer to the packed memory structure storing the data, and "tree" to refer to the static cache-oblivious structure that serves as an index. We use this terminology even though the "tree" is actually stored as an array.

The N data items are stored in a packed-memory structure, which is an array A of size $O(N)$. Recall that the items appear in the array in sorted order but some of the array elements are kept blank.

Recall that the static cache-oblivious structure consists of a complete tree. Let T_i denote the ith leftmost leaf in the tree. In our structure there are pointers between array position $A[i]$ and leaf T_i, for all values of i. We maintain the invariant that $A[i]$ and T_i store the same key value. All internal nodes of T store the maximum of the non-blank key value(s) of its children. If a node has two children with blank key values, it also has a blank key value.

We now describe the supported operations:

PREDECESSOR(x): Predecessor is carried out by traversing the tree from the root to a leaf. Since each internal node stores the maximum key value of the leaves in its induced subtree, this search is similar to the standard predecessor search on a binary search tree. When the search has reached a node u, it decides whether to branch left or right by comparing the search key to the key of u's left child.

THEOREM 2.1. *The operation* PREDECESSOR(x) *uses* $O(\log_B N)$ *block transfers.*

Proof. Search in our structure is similar to the $O(\log_B N)$ root-to-leaf traversal described in Lemma 2.1, except that the process of examining the key value of the current node's left child at every step may cause an additional $O(\log_B N)$ block transfers. ∎

SCANFORWARD(), SCANBACKWARD(): These operations are implemented by scanning forward or backwards to the next non-blank item in the array from the last item accessed. Because of the density constraint on the array, we are guaranteed that we only scan $O(1)$ elements to find a non-blank element.

THEOREM 2.2. *A sequence of k* SCANFORWARD *or* SCANBACKWARD *operations uses $O(\frac{k}{B}+1)$ block transfers.*

Proof. A sequence of k SCANFORWARD or SCANBACKWARD operations only accesses $O(k)$ consecutive elements in an array in order. Thus scan takes $O(\frac{k}{B}+1)$ block transfers. ∎

INSERT(x), DELETE(x): We describe INSERT(x); DELETE(x) proceeds in the same manner. We insert in three stages: First, we perform a predecessor query to find the location in the array to insert x. Then, we insert x into the array using the insertion algorithm of the packed-memory structure. Finally, we update the key values in the tree to reflect the changes in the array.

The first two steps are straightforward; we now describe the third step in more detail: First, we copy the updated keys from the array into the corresponding locations in the tree. We then update all of the ancestors of the updated leaves. We proceed through this subtree according to the postorder traversal. The updating process changes the key value of a node to reflect the maximum of the key values of its children. By updating using the postorder traversal, we can guarantee that when we reach a given node the values of its children have been updated already.

LEMMA 2.2. *To perform a postorder traversal on k leaves in the tree and their ancestors requires $O(\log_B N + \frac{k}{B})$ block transfers.*

Proof. We consider the largest level of detail in the tree where recursive subtrees are smaller than B. Consider the horizontal stripes formed by the subtrees at this level of detail. On any root-to-leaf path we pass through $O(\log_B N)$ stripes. We number the stripes from the bottom of the tree starting at 1. Each stripe consists of a forest of subtrees of the original tree. If the root of a tree T_a in stripe i is a child (in the full tree) of a leaf of a tree T_b in stripe $i+1$, we say that T_a is a tree-child of T_b.

Accessing all of the items in one tree in any stripe uses at most two memory transfers, since the subtree is stored in a consecutive region of memory of size at most B. We now analyze the cost of accessing one of the stripe-2 trees T and all of its stripe-1 tree-children. The size of all of these trees will be in the range $\sqrt{B}$ to B. In the postorder traversal, all of the stripe-1 trees will be accessed in the order that they are stored in the array. Since all of the stripe-1 tree-children of T are stored consecutively in memory, the number of page faults caused by accessing l consecutive items in the stripe-1 trees of T in postorder is at most $1+\frac{2l}{B}$, provided memory can hold 2 blocks. Note that accessing all of the items in T takes 2 memory transfers provided that memory can hold 2 blocks. Accessing any k consecutive items in T and to its tree-children involves interleaving accesses to items in T and tree-children. Interleaving operations takes no more block transfers than doing operations separately, provided sufficient cache is available. Thus at most $2+\frac{2k}{B}$ block transfers are performed if memory can hold 4 blocks. In general, to access k consecutive items in the tree in postorder mostly consists of accessing level-1 and level-2 subtrees. In addition, there will be at most $\log_B N + \frac{k}{B}$ items accessed at stripes 3 and higher. Each of these accesses causes at most 1 memory transfer, provided 1 block of memory is available. Thus at most $2+\log_B N+3\frac{k}{B}$ block transfers are performed to access k consecutive items in the tree in postorder, given a cache of 5 blocks. ∎

THEOREM 2.3. *The number of block transfers caused by the* INSERT(x) *and* DELETE(x) *operations is* $O(\log_B N + \frac{\log^2 N}{B})$.

Proof. We describe the proof for INSERT(x); DELETE(x) proceeds in the same manner. The predecessor query costs $O(\log_B N)$. The cost of inserting into the packed memory structure is $O(\frac{\log^2 N}{B} + 1)$ amortized memory transfers. Let w be the actual number of items changed in the array by the packed-memory structure. By Lemma 2.2 updating the internal nodes uses $O(\log_B N + \frac{w}{B})$ memory transfers. Since $O(\frac{w}{B} + 1)$ is asymptotically the same as the actual number of block transfers performed by the packed-memory structure's insertion into the array, it may be restated as the $O(\frac{\log^2 N}{B} + 1)$ amortized cost of insertion into the packed-memory structure. Therefore the entire INSERT operation uses $O(\frac{\log^2 N}{B} + \log_B N)$ amortized memory transfers. ∎

3 Simulation Results

Our objective is to understand how the block size B and the cache size M affect the performance of our data structure, and how it compares to a standard B-tree. In our simulations we began with an empty structure and inserted up to 1,000,000 elements. Each data entry is a 32-bit integer. Whenever the array becomes too full we recopy the elements into a larger array.

We tested our structure using the following input patterns:

- *Insert at Head* — The elements are inserted at the beginning of the array. This insertion pattern models close to worst-case behavior, where the inserts "hammer" on one region of the array.
- *Random Inserts* — Each insertion is before a random element in the array. We implement the random insert by assigning each new element a random 32-bit key.
- *Bulk Inserts* — This insertion pattern is a middle ground between random inserts and insert at head. In this strategy we pick a random element to insert before and insert before it a sequence of elements. (We perform the packed-memory structure modification after each element is inserted.) We run the simulations with block sizes of 1, 10, 100, 1000, 10,000, 100,000 and 1,000,000. Observe that random inserts and insert at head are special cases of bulk insert with bulk size 1 and 1,000,000, respectively.

Our experiments have three parts. First, we test the packed-memory structure to measure the amortized number of moved elements or array positions scanned per insertion. We consider different density thresholds as well as different density patterns. We next built a memory simulator, which models which blocks are in memory and which blocks are on disk. We adopt the standard Least Recently Used (LRU) replacement strategy and assume full associativity. Thus, whenever we touch a block that is not in memory, we increment the page-fault count, bring the block into memory, and eject the least-recently-used block. We separately measure the number of page faults caused by packed-memory-structure manipulations and index-structure manipulations. Finally, we compare our structure with a standard B-tree.

3.1 Scans and Moves. We measure both the number of elements moved and the number of array positions scanned, that is, touched during a rebalance. Note that each moved element is moved approximately twice. This is because when we rebalance an interval, first we compress the elements at one end of the interval, and then we space the elements evenly in the interval, moving each element twice. Figures 1-4 show our results for the insert-at-head insertion strategy. We consider density parameters of 50%, 60%, 70%, 80%, and 90%. With a 60% density threshold the average number of moves is only 320 when inserting 1,000,000 elements, and only 350 when inserting 2,000,000 elements. Even with a 90% density threshold the number of moves is only 1100 when inserting 2,000,000 elements.

Figures 5 and 6 show the number of moves and scanned array positions for random inserts. There are only a small constant number of element moves or scanned elements per random insertion. Figures 7 and 8 depict bulk inserts ranging from best case (random) to worst case (insert-at-head); the number of moves increases with the bulk size.

3.2 Page Faults. We first focus on the page faults caused by the packed-memory structure using the insert-at-head insertion strategy. As expected, the number of page faults is more sensitive to B than to N. The number of page faults behaves roughly linear in B, as depicted in Figures 9 and 10. Specifically, Figure 9 suggests that the number of page faults decreases linearly when B increases and M remains fixed. Figure 10 indicates that the number of page faults decreases roughly linearly when B is fixed and M increases exponentially.

Overall, these results are promising. Even for insertions at the head, the number of page faults per insertion is less that 1 for most reasonable values of B and M. There are few page faults because we are attempting to insert into one portion of the array which can be kept in cache unless a rebalance window is larger then the cache size, which is a relatively infrequent event.

Although random insertions are close to the best case in terms of number of elements moved, they are close to the worst case in terms of number of page faults! This is because since the location of the insert in the array is chosen randomly, it is unlikely to be in the cache

bulk size	array pagefault	index pagefault	total
10^0	1.0	2.2	3.2
10^1	0.28	0.23	0.51
10^2	0.076	0.028	0.10
10^3	0.083	0.0095	0.093
10^4	0.37	0.024	0.39
10^5	0.65	0.031	0.69
10^6	0.83	0.030	0.86

Table 1: Average page faults in our data structure.

node size	96	816	1616	3216
bulk size				
10^0	4.6	3.4	6.3	12
10^1	0.47	0.35	0.66	1.2
10^2	0.060	0.046	0.074	0.14
10^3	0.020	0.019	0.02163	0.030
10^4	0.016	0.016	0.016	0.017
10^5	0.016	0.016	0.016	0.016
10^6	0.016	0.016	0.016	0.016

Table 2: Average page faults in the B-tree.

already. Thus, we expect approximately one page fault per insertion, which is supported by Figure 11. The situation for bulk insertions as illustrated in Figure 12 is dramatically better than for random insertions because the cost of the page fault is amortized over the number of elements inserted in a region.

We next measure the number of page faults caused by the entire structure, separately considering the contribution from the searching the index and the scanning the array. We consider bulk sizes of 1, 10, 100, 1000, 10,000, 100,000, and 1,000,000; see Figures 13 and 14 and Tables 1 and 2. Interestingly, the worst case is for random inserts where there are typically two page faults caused by the index and one caused by the scanning structure. As the bulk size increases to 10 and then 100 we obtain almost order-of-magnitude improvements in efficiency. For larger bulk sizes the rebalancing in the packed-memory structure hurts the cache performance, increasing the number of page faults. However, the performance is never worse than for random insertions.

We also simulated a B-tree. The data structure is a modification of David Carlson's implementation [16] based on the one found in [28]. The data entries in the B-tree are also 32 bit integers. Each node of the B-tree contains at most B data entries and $B + 1$ branches. The page fault rate of insertion into the B-tree is tested with different B and different bulk sizes.

Insert at head is by far the best case for B-trees. Interestingly, this is not even the case B-trees are optimized for because there is little advantage to the B-sized fan-out since an entire root-to-leaf path fits in cache. When we increase B but keep the node size within the size of a page, the performance improves. However, if the node size gets larger than the page size, the performance gets much worse, especially when inserting randomly. We measured the search efficiency when searching for a random key from 1,000,000 indexed elements. When the page size is 1024 bytes, the average page fault per search of the B-tree with node size 816 is 3.77, whereas the average page fault per search for our structure is 3.69.

4 Conclusion

We have developed and simulated a new cache-oblivious locality-preserving dictionary, which supports Insert, Delete, and Scan operations. Our structure has two advantages not held by the standard B-tree. First, it is cache oblivious, that is, it is not parameterized by the block size or any other characteristics of the memory hierarchy. Second, it locality preserving. That is, unlike any other dynamic dictionary (except for [11], which seems too complicated too implement), the structure keeps data stored compactly in memory in order.

Interestingly, although our structure is algorithmically more sophisticated than the B-tree, it may be of comparable difficulty to implement. Unlike the B-tree structure, which requires dynamic memory allocation and pointer manipulation, our structure is just two static arrays.

Different insertion patterns have different costs in our structure and in the standard B-tree. Our simulations indicate that our worst-case performance is at least as good as the worst-case performance of the B-tree for typical block and memory sizes. Indeed, when the B-tree is not optimized for the block size then our structure outperforms the B-tree. This worst-case performance is exhibited during random insertions. On the other hand, because we must keep data in order, we can not match the B-tree performance when all insertions are to the same location. However, even in the adversarial case, we still perform better than when data is evenly distributed. More research needs to be done to test our data structure on actual input distributions.

For the special case where we know the block size and where the two-level DAM model is an accurate cost model of the system, then the B-tree is of course the best option since it is optimized for the DAM model. However, It is becoming increasingly important to optimize for multilevel memories. Furthermore, the research effort in clustering B-tree blocks and keeping data in order suggests that the DAM model can be improved, even on two-level memory hierarchies. More work should be performed on developing more realistic cost models and testing our structure on these models.

If we do not need scans then we can use one level of indirection to performs searches and updates in amortized $O(\log_B N)$ memory transfers (see [11] for details). We can also use our data structure to keep data ordered within superblocks that are arbitrarily placed in memory. Thus, practitioners can benefit from the cache-

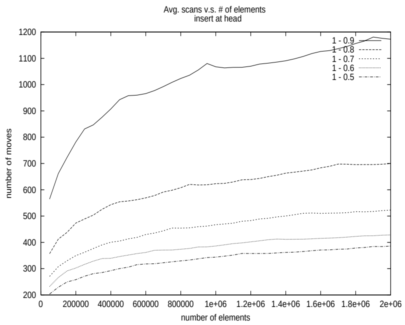

Figure 1: Insert-at-head, average number of scans versus number of elements.

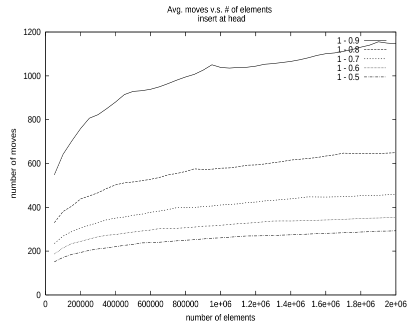

Figure 2: Insert-at-head, average number of moves versus number of elements.

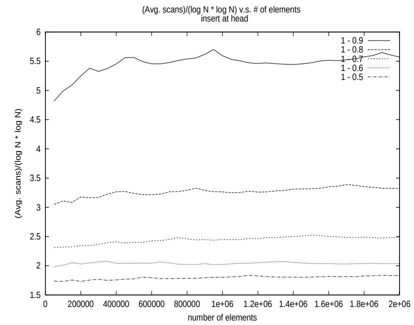

Figure 3: Insert-at-head, (number of scans)/$(\log^2 N)$ versus number of elements.

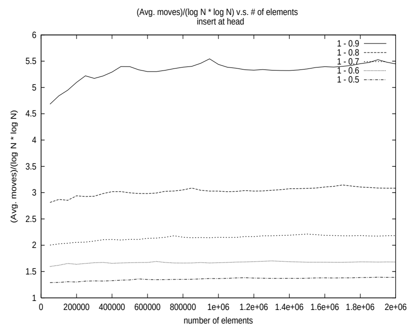

Figure 4: Insert-at-head, (number of moves)/$(\log^2 N)$ versus number of elements.

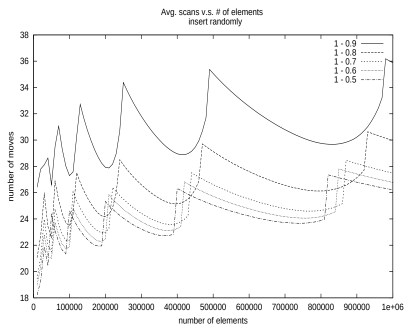

Figure 5: Random insertion, Average number of scans versus number of elements.

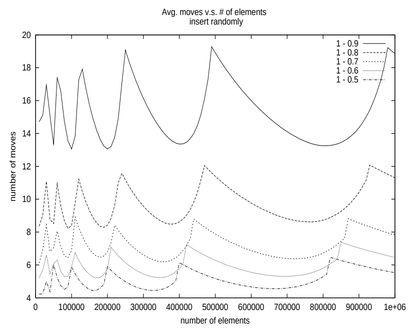

Figure 6: Random insertion, Average number of moves versus number of elements.

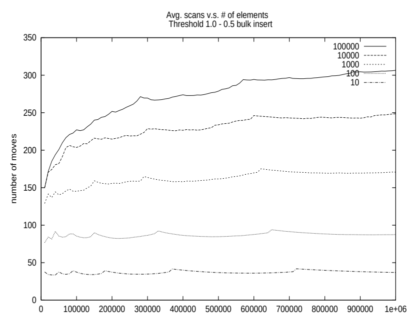

Figure 7: Average scans and moves for different bulk insert sizes for density threshold 0.5, average number of scans versus number of elements.

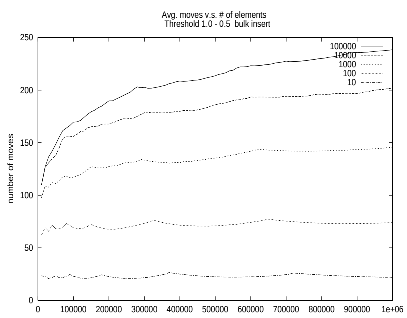

Figure 8: Average scans and moves for different bulk insert sizes for density threshold 0.5, average number of moves versus number of elements.

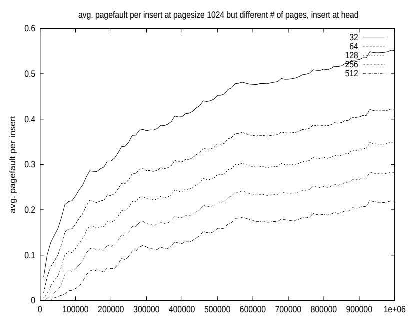

Figure 9: Average page faults per insert, at page size 1024 but with different numbers of pages.

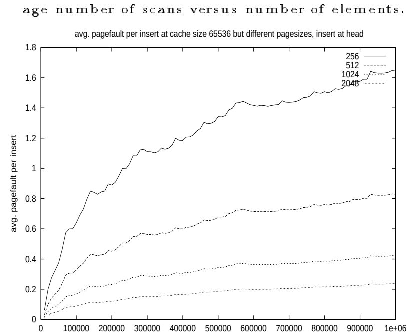

Figure 10: Average page faults per insert, at cache size 65536 but with different page sizes.

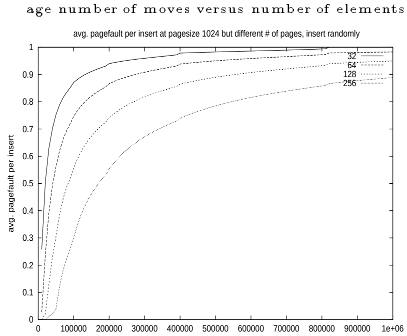

Figure 11: Average page faults per insert, at page size 1024 but with different numbers of pages.

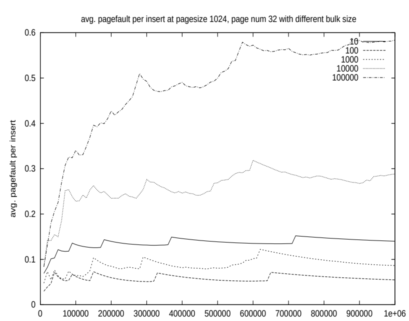

Figure 12: Average page faults per insert, at cache size 32K, page size 1024, but with different bulk sizes.

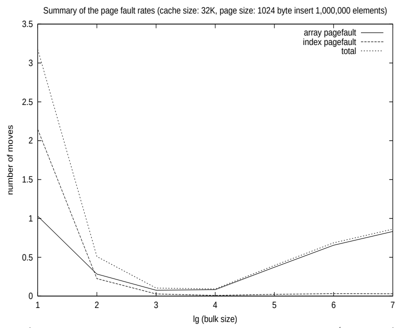

Figure 13: Page fault rate versus $\log_{10}$(bulk size) (insert 1,000,000 elements, cache size 32767 bytes, page size 1024 bytes), our data structure.

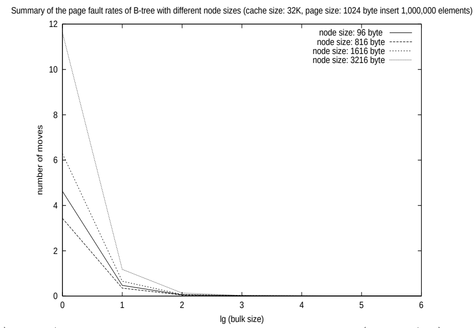

Figure 14: Page fault rate versus $\log_{10}$(bulk size) (insert 1,000,000 elements, cache size 32767 bytes, page size 1024 bytes), B-tree.

oblivious index structure and modify the superblocks according to need.

Acknowledgments

The authors gratefully acknowledge Jon Bentley, Erik Demaine, Martín Farach-Colton, Petr Konečný, and Torsten Suel for useful discussions.

References

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for organization of information (in Russian). *Doklady Akademii Nauk SSSR*, 146:263–266, 1962.

[2] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. ACM Symp. on Theory of Computing*, pages 305–314, 1987.

[3] A. Aggarwal and A. K. Chandra. Virtual memory algorithms. In *Proc. ACM Symp. on Theory of Computation*, pages 173–185, 1988.

[4] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. IEEE Symp. on Foundations of Computer Science*, pages 204–216, 1987.

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

[6] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2–3):72–109, 1994.

[7] A. Andersson. Faster deterministic sorting and searching in linear space. In *Proc. Symp. Foundations of Computer Science (FOCS)*, pages 135–141, 1996.

[8] A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proc. ACM Symp, on Theory of Computing*, pages 335–342, 2000.

[9] L. Arge, K. Hinrichs, and J. Vitter. Efficient bulk operations on dynamic r-trees. In *ALENEX*, pages 328–348, 1999.

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

[11] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. Symp. on Foundations of Computer Science*, pages 399–409, 2000.

[12] G. Bilardi, P. D'Alberto, and A. Nicolau. Fractal matrix multiplications: a case study on portability of cache performance. In *Proc. Workshop on Algorithm Engineering*, pages 26–38, 2001.

[13] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 297–308, 1996.

[14] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height (extended abstract). In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 2002.

[15] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM J. Comput.*, 9:594–614, 1980.

[16] D. Carlson. Software design using c++. *http://cis.stvincent.edu/carlsond/swdesign/swd.html*, 2001.

[17] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[18] J. V. den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, editors, *Proc. International Conference on Very Large Data Bases, 1997,* , pages 406–415, 1997.

[19] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. Symp. on Foundations of Computer Science*, pages 285–297, 1999.

[20] J. Gray and G. Graefe. The five minute rule ten years later. *SIGMOD Record*, 26(4), 1997.

[21] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. Symp. on Foundations of Computer Science*, pages 8–21, 1978.

[22] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[23] J. Iacono. Alternatives to splay trees with $O(\log n)$

worst-case access times. In *Proc. Symp. on Discrete Algorithms*, pages 516–522, 2001.

[24] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. Annual International Colloquium on Automata, Languages, and Programming, LNCS 115*, pages 417–431, 1981.

[25] I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. International Conference on Information and Knowledge Management*, pages 490–499, 1993.

[26] K. Kim and S. K. Cha. Sibling clustering of tree-based spatial indexes for efficient spatial query processing. In *Proc. ACM Intl. Conf. Information and Knowledge Management*, pages 398–405, 1998.

[27] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.

[28] R. L. Kruse and A. J. Ryba. *Data strcutres and program design in C++*. Prentice-Hall, Upper Saddle River, New Jersey, 1998.

[29] R. Meville and D. Gries. Controlled density sorting. *Infor. Process. Lett.*, 10:169–172, 1980.

[30] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2:33–43, 1973.

[31] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.

[32] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.

[33] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Proc. Workshop on Algorithms and Data Structures*, volume 382 of *LNCS*, pages 437–449, 1989.

[34] N. Rahman, R. Cole, and R. Raman. Optimized predecessor data structures for internal memory. In *Proc. Workshop on Algorithms Engineering* , pages 67–78, August 2001.

[35] V. Raman. Locality preserving dictionaries: theory and application to clustering in databases. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1999.

[36] J. E. Savage. Extending the Hong-Kung model to memory hierachies. In *Proc. International Conference on Computing and Combinatorics*, volume 959 of *LNCS*, pages 270–281, 1995.

[37] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4–5):464–497, 1996.

[38] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[39] M. Thorup. Faster deterministic sorting and priority queues in linear space. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 550–555, 1998.

[40] S. Toledo. Locality of reference in *LU* decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4):1065–1081, Oct. 1997.

[41] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proc. Symp. on Foundations of Computer Science*, pages 75–84, 1975.

[42] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(2):99–127, 1977.

[43] J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 1–38. American Mathematical Society, DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1999.

[44] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994.

[45] D. E. Willard. Inserting and deleting records in blocked sequential files. Technical Report TM81-45193-5, Bell Laboratories, 1981.

[46] D. E. Willard. Maintaining dense sequential files in a dynamic environment. In *Proc. ACM Symp. on Theory of Computation*, pages 114–121, 1982.

[47] D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 251–260, 1986.

[48] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *Information and Computation*, 97(2):150–204, 1992.

[49] Q. Yi, V. Advi, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 169–181, 2000.