The Cost of Cache-Oblivious Searching

Michael A. Bender*† Gerth Stølting Brodal‡§ Rolf Fagerberg‡ Dongdong Ge*
Simai He* Haodong Hu* John Iacono¶ Alejandro López-Ortiz‖

Abstract

Tight bounds on the cost of cache-oblivious searching are proved. It is shown that no cache-oblivious search structure can guarantee that a search performs fewer than $\lg e \log_B N$ block transfers between any two levels of the memory hierarchy. This lower bound holds even if all of the block sizes are limited to be powers of 2. A modified version of the van Emde Boas layout is proposed, whose expected block transfers between any two levels of the memory hierarchy arbitrarily close to $[\lg e + O(\lg\lg B/\lg B)] \log_B N + O(1)$. This factor approaches $\lg e \approx 1.443$ as B increases. The expectation is taken over the random placement of the first element of the structure in memory.

As searching in the Disk Access Model (DAM) can be performed in $\log_B N + 1$ block transfers, this result shows a separation between the 2-level DAM and cache-oblivious memory-hierarchy models. By extending the DAM model to k levels, multilevel memory hierarchies can be modelled. It is shown that as k grows, the search costs of the optimal k-level DAM search structure and of the optimal cache-oblivious search structure rapidly converge. This demonstrates that for a multilevel memory hierarchy, a simple cache-oblivious structure almost replicates the performance of an optimal parameterized k-level DAM structure.

*Department of Computer Science, SUNY Stony Brook, Stony Brook, NY 11794, USA. {bender,dge,simaihe, hhaodong}@cs.sunysb.edu.

†Partially supported by Sandia National Laboratories and the National Science Foundation grants EIA-0112849 and CCR-0208670.

‡BRICS, Department of Computer Science, University of Aarhus, Ny Munkegade, 8000 Århus C, Denmark. {gerth,rolf}@brics.dk. Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

§Supported by the Carlsberg Foundation (contract number ANS-0257/20).

¶Department of Computer and Information Science, Polytechnic University, Brooklyn, NY 11530, USA. jiacono@poly.edu.

‖School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada. alopez-o@uwaterloo.ca.

1 Introduction

Hierarchical Memory Models. Traditionally, algorithms were designed to run efficiently in a *random access model (RAM)* of computation, which assumes a flat memory with uniform access times. However, as hierarchical memory systems become steeper and more complicated, algorithms are increasingly designed assuming more accurate memory models; see e.g., [2–5, 7, 8, 10, 22, 31–33, 38–40]. Two of the most successful memory models are the *disk-access model (DAM)* and the *cache-oblivious model*.

The DAM model, developed by Aggarwal and Vitter [4], is a two-level memory model, in which the memory hierarchy consists of an internal memory of size M and an arbitrarily large external memory partitioned into blocks of size B. Algorithms are designed in the DAM model with full knowledge of the values of B and M. Because memory transfers are relatively slow, the performance metric is the number of block transfers.

The cache-oblivious model, developed by Frigo, Leiserson, Prokop, and Ramachandran [21, 29], allows programmers to reason about a two-level memory hierarchy but to prove results about an unknown multilevel memory hierarchy. As in the DAM model, the objective is to minimize the number of block transfers between two levels. The main idea of the cache-oblivious model is that by avoiding any memory-specific parametrization (such as the block sizes) the cache-oblivious algorithm has an asymptotically optimal number of memory transfers between all levels of an unknown, multilevel memory hierarchy.

Optimal cache-oblivious algorithms have memory performance (i.e., number of memory transfers) that is within a constant factor (independent of B and M) of the memory performance of the optimal DAM algorithm, which knows B and M. There exist surprisingly many (asymptotically) optimal cache-oblivious algorithms [1, 9, 11–21, 24, 25, 29, 30, 35].

I/O-Efficient Searching. This paper focuses on the fundamental problem of searching: Given a set S of N comparison-based totally-ordered elements, produce a data structure that can execute searches (or predecessor queries) on items in S.

We provide tight bounds on the cost of cache-oblivious searching. We show that no cache-oblivious search structure can guarantee that a search performs fewer than $\lg e \log_B N$ [1] block transfers between any two levels of the memory hierarchy, even if all of the block sizes are limited to powers of 2. We also prove a search structure in which the expected number of block transfers between any two levels of the memory hierarchy is arbitrarily close to $[\lg e + O(\lg\lg B/\lg B)]\log_B N + O(1)$, which approaches $\lg e \log_B N + O(1)$ for large B. This expectation is taken over the random placement of the first element of the structure in memory.

In contrast, the performance of the B-tree, the classic optimal search tree in the DAM model, is as follows: A B-tree with N elements has nodes with fan-out B, which are designed to fit into one memory block. The B-tree has height $\log_B N + 1$, and a search has $\log_B N + 1$ memory transfers (block cost).

A static cache-oblivious search tree, proposed by Prokop [29], also performs searches in $\Theta(\log_B N)$ memory transfers. The static cache-oblivious search tree is built as follows: Embed a complete binary tree with N nodes in memory, conceptually splitting the tree at half its height, thus obtaining $\Theta(\sqrt{N})$ subtrees each with $\Theta(\sqrt{N})$ nodes. Lay out each of these trees contiguously, storing each recursively in memory. This type of recursive layout is called a *van Emde Boas layout* because it is reminiscent of the recursive structure of the van Emde Boas tree [36, 37]. The static cache-oblivious search tree is a basic building block of essentially all cache-oblivious search structures, including the (dynamic) cache-oblivious B-tree of Bender, Demaine, and Farach-Colton [14], its simplifications and improvements [15, 19, 30], and other cache-oblivious search structures [1, 6, 12, 13, 17, 18]. Any improvements to the static cache-oblivious search structure immediately translate to improvements to these dynamic structures.

[1] Throughout the paper $\lg N$ means $\log_2 N$

Results. We present the following results:

- We give an analysis of Prokop's static cache-oblivious search tree [29], proving that searches perform at most $2\left(1 + \frac{3}{\sqrt{B}}\right)\log_B N + O(1)$ expected memory transfers; the expectation is taken only over the random placement of the data structure in memory. This analysis is tight to within a $o(1)$ factor.
- We then present a class of generalized van Emde Boas layouts that optimizes performance through the use of uneven splits on the height of the tree. For any constant $\epsilon > 0$, we optimize the layout achieving a performance of $[\lg e + \epsilon + O(\lg\lg B/\lg B)]\log_B N + O(1)$ expected memory transfers. As before, the expectation is taken over the random placement of the data structure in memory. We prove that a numerical analysis within a limited range of values of N can bound this constant over an infinite range of values of N.
- Finally, we demonstrate that it is harder to search in the cache-oblivious model than in the DAM model. Previously the only lower bound for searching in the cache oblivious model was the $\log_B N$ lower bound from the DAM model. We prove a lower bound of $\lg e \log_B N$ memory transfers for searching in the average case in the cache-oblivious model, which also proves that our layout is optimal in this model to within an additive constant as B increases.

Interpretation. We present a cache-oblivious search structure that takes 44% more block transfers than the optimal DAM structure, and we prove that we cannot do any better. However, this result does not mean that our cache-oblivious structure is 44% slower than an optimal algorithm for a multilevel memory hierarchy. To the contrary, this worst-case behavior only occurs on a two-level memory hierarchy. To design a structure for a k-level memory hierarchy, one can extend the DAM model to k levels. A data structure for a k-DAM is designed with full knowledge of the size and block size of each level of the memory hierarchy. Thus, the 2-DAM is the standard DAM where searches cost $\log_B N + 1$ block transfers (using a B-tree). Surprisingly, in the 3-DAM this performance cannot be replicated in general.

We show in Corollary 2.3 that a 3-DAM algorithm cannot achieve less than $1.207 \log_B N$ block transfers on all levels simultaneously. Thus, the performance gap between a 3-DAM and the optimal cache-oblivious structure is about half that of the 2-DAM and the optimal cache-oblivious structure; naturally, a modern memory hierarchy has more than three levels. Furthermore, we show that as the number k of levels in the memory hierarchy grows, the performance loss of our cache-oblivious structure relative to an optimal k-DAM structure tends to zero. Thus, for a modern memory hierarchy, our cache-oblivious structure combines simplicity and near-optimal performance.

Our cache-oblivious search trees also provide new insight into the optimal design strategy for divide-and-conquer algorithms. More generally, it has been known for several decades that divide-and-conquer algorithms frequently have good data locality [34]. The cache-oblivious model provides a mechanism to understand why divide-and-conquer is advantageous.

When there is a choice, the splitting in a divide-and-conquer algorithm is traditionally done evenly. The unquestioned assumption is that splitting evenly is best. Our new search structure serves to disprove the myth that even splits yield the best results. This paper suggests the contrary: uneven splits can yield better worst-case performance.

2 Lower Bound

In this section, we prove lower bounds for the I/O cost of cache-oblivious comparison based searching. The problem we consider is the average cost of successful searches among N distinct elements, where the average is over a uniform distribution of the search key y on the N input elements. For lower bounds, average case complexity is stronger than worst case complexity, so our bounds also apply to the worst case cost. We note that our bounds hold even if the block sizes are known to the algorithm, and that they hold for any memory layout of data, including any specific placement of a single data structure.

Formally, our model is as follows. Given a set S of N elements $x_1 < \cdots < x_N$ from a totally ordered universe, a *search structure* for S is an array M containing elements from S, possibly with several copies of each. A *search algorithm* for M is a binary decision tree where each internal node is labeled with either $y < M[i]$ or $y \leq M[i]$ for some array index i, and each leaf is labeled with a number $1 \leq j \leq N$. A search on a key y proceeds in a top-down fashion in the tree, and at each internal node advances to the left child if the comparison given by the label is true, otherwise it advances to the right. A binary decision tree is a correct search algorithm if for any $x_i \in S$, the path taken by a search on key $y = x_i$ ends in a leaf labeled i. Any such tree must have at least N leaves, and by pruning paths not taken by any search for $x_1, \ldots, x_N$, we may assume that it has exactly N leaves.

To add I/Os to the model, we divide the array M into contiguous *blocks* of size B. An internal node of a search algorithm is said to *access* the block containing the array index i in the label of the node. We define the I/O cost of a search to be the number of distinct blocks of M accessed on the path taken by the search.

The main idea of our proof is to analyze the I/O cost of a given search algorithm with respect to several block sizes simultaneously. We first describe our method for the case of two block sizes. This will lead to a lower bound of $1.207 \log_B N$. We then generalize this proof to a larger number k of block sizes, and prove that in the limit as k grows, this gives a lower bound of $\lg e \log_B N \approx 1.443 \log_B N$.

Throughout this section, we assume that block sizes are powers of two and that blocks start at memory addresses divisible by the block size. This reflects the situation on actual machines, and entails no loss of generality, as any cache-oblivious algorithm at least should work for this case. The assumption implies that for two block sizes $B_1 < B_2$, a block of size B_1 is contained in exactly one block of size B_2.

Lemma 2.1 ([23, Section 2.3.4.5]) *For a binary tree with N leaves, the average depth of a leaf is at least $\lg N$.*

Lemma 2.2 *If a search algorithm on a search structure for block sizes B_1 and B_2, where $B_2 = {B_1}^c$ and $1 < c \leq 2$, guarantees that the average number of block reads is at most $\delta \log_{B_1} N$ and $\delta \log_{B_2} N$, respectively, then*

$$\delta \geq \frac{1}{2/c + c - 2 + 3/(c \lg B_1)} .$$

Proof: Let T denote the binary decision tree constituting the search algorithm. Our goal is to transform T into a new binary decision tree T' by transforming each node that accesses a new size B_1 block in T into a binary decision tree of small height, and discarding all other nodes in T. A lower bound on the average depth of leaves in T' then translates into a lower bound on the average number of blocks accesses in T.

To count the number of I/Os of each type (size B_1 blocks and size B_2 blocks) for each path in T, we mark some of the internal nodes by tokens τ_1 and τ_2. A node v is marked iff none of its ancestors accesses the size B_1 block accessed by v, i.e. if v is the first access to the block. The node v may also be the first access to the size B_2 block accessed by v. In this case, v is marked by τ_2, else it is marked by τ_1. Note that the word "first" above corresponds to viewing each path in the tree as a timeline—this view will be implicit in the rest of the proof.

For any root-to-leaf path, let b_i denote the number of distinct size B_i blocks accessed and let a_i denote the number of τ_i tokens on the path, for $i = 1, 2$. By the assumption stated above Lemma 2.1, a first access to a size B_2 block implies a first access to a size B_1 block, so we have $b_2 = a_2$ and $b_1 = a_1 + a_2$.

We transform T into a new binary decision tree T' in a top-down fashion. The basic step in the transformation is to substitute a marked node v with a specific binary decision tree T_v resolving the relation between the search key y and a carefully chosen subset S_v of the elements. More precisely, in each step of the transformation, the subtree rooted at v is first removed, then the tree T_v is inserted at v's former position, and finally a copy of one of the two subtrees rooted at the children of v is inserted at each leaf of T_v. The top-down transformation then continues downwards at the leafs of T_v. When the transformation reaches a leaf, it is left unchanged. The resulting tree can contain several copies of each leaf of T.

We now describe the tree T_v inserted, and first consider the case of a node v marked τ_2. We let the subset S_v consist of the at most B_1 distinct elements in the block of size B_1 accessed by v, plus every $\frac{B_2}{2B_1}$th element in sorted order among the at most B_2 distinct elements in the block of size B_2 accessed by v. The size of S_v is at most $B_1 + B_2/(B_2/(2B_1)) = 3B_1$.

The tree T_v is a binary decision tree of minimal height resolving the relation of the search key y to all keys in S_v. If we have $S_v = \{z_1, z_2, \ldots, z_t\}$, with elements listed in sorted order and $t \leq 3B_1$, this amounts to resolving which of the at most $6B_1 + 1$ intervals

$$(-\infty; z_1)\ ,\ \ [z_1; z_1]\ ,\ \ (z_1; z_2)\ , \ldots ,\ [z_t; z_t]\ ,\ \ (z_t; \infty)$$

that y belongs to (we resolve for equality because we chose to allow both $<$ and $\leq$ comparisons in the definition of comparison trees, and want to handle both types of nodes in the transformation). The tree T_v has height at most $\lceil \lg(6B_1 + 1) \rceil$, since a perfectly balanced binary search tree on S_v, with one added layer to resolve equality questions, will do. As B_1 is a power of two, $\lg(8B_1)$ is an integer and hence an upper bound on the height.

For the case of a node v marked τ_1, note that v in T has exactly one ancestor u marked τ_2 that accesses the same size B_2 block β as v does. When the tree T_u was substituted for u, the inclusion in S_u of the $2B_1$ evenly sampled elements from β ensures that below any leaf of T_u, at most $\frac{B_2}{2B_1} - 1$ of the elements in β can still have an unknown relation to the search key. The tree T_v is a binary decision tree of minimal height resolving these relations. Such a tree has at most $2\frac{B_2}{2B_1} - 1 = \frac{B_2}{B_1} - 1$ leaves and hence height at most $\lg \frac{B_2}{B_1}$, as B_1 and B_2 are powers of two.

Since in both cases T_v resolves the relation between the search key y and all sampled elements, the relation between the search key and the element accessed at v is known at each leaf of T_v, and we can choose either the left or right child of v to continue the transformation with.

When we in the top-down transformation meet an unmarked internal node v (i.e. a node where the size B_1 block accessed at the node has been accessed before), we can similarly discard v together with either the left or right subtree, since we already have resolved the relation between the search key y and the element accessed at v. This follows from the choice of trees inserted at marked nodes: when we access a size B_2 block β_2 for the first time at some node u, we resolve the relation between the search key y and all elements in the size B_1 block β_1 accessed at u (due to the inclusion of all of β_1 in S_u), and when we first time access a key in β_2 outside β_1, we resolve all remaining relations between y and elements in β_2.

The tree T' resulting from this top-down transformation is a binary decision tree. By construction, each search in T' ends in a leaf having the same label as the leaf that the same search in T ends in (this is an invariant during the transformation), so T' is a correct search algorithm if T is.

By the height stated above for the inserted T_v trees, it follows that if a search for a key y in T corresponds to a path containing a_1 and a_2 tokens of type τ_1 and τ_2, respectively, then the search in T' corresponds to a path with length bounded by the following expression.

$$\begin{aligned} & a_2 \lg(8B_1) + a_1 \lg \frac{B_2}{B_1} \\ = \; & b_2 \lg(8B_1) + (b_1 - b_2) \lg \frac{B_2}{B_1} \\ = \; & b_2 \left(\lg(8B_1) - \lg \frac{B_2}{B_1} \right) + b_1 \lg \frac{B_2}{B_1} \end{aligned}$$

The coefficients of b_2 and b_1 are positive by the assumption $B_1 < B_2 \leq {B_1}^2$, so upper bounds on b_1 and b_2 imply an upper bound on the expression above. By assumption, the average values over all search paths of b_1 and b_2 are bounded by $\delta \log_{B_1} N$ and $\delta \log_{B_2} N = (\delta \log_{B_1} N)/c$, respectively.

If we prune the tree for paths not taken by any search for the keys $x_1, \ldots, x_N$, the lengths of root-to-leafs paths can only decrease. The resulting tree has N leaves, and Lemma 2.1 gives a $\lg N$ lower bound on the average depth of a leaf. Hence, we get

$$\begin{aligned} \lg N \leq \; & \frac{\delta}{c} \log_{B_1} N \left(\lg(8B_1) - \lg \frac{B_2}{B_1} \right) \\ & + \delta \log_{B_1} N \lg \frac{B_2}{B_1} \\ = \; & \frac{\delta}{c} \log_{B_1} N (3 + \lg B_1 - (c-1) \lg B_1) \\ & + \delta \log_{B_1} N (c-1) \lg B_1 \\ = \; & \delta \lg N (3/(c \lg B_1) + 1/c - (c-1)/c + (c-1)) \\ = \; & \delta \lg N (3/(c \lg B_1) + c + 2/c - 2) \; . \end{aligned}$$

It follows that $\delta \geq 1/(3/(c \lg B_1) + c + 2/c - 2)$. □

Corollary 2.3 *If a search algorithm on a search structure guarantees, for all block sizes B, that the average number of block reads for a search is at most $\delta \log_B N$, then $\delta \geq 1/(2\sqrt{2} - 2) \approx 1.207$.*

Proof: Letting $c = \sqrt{2}$ in Lemma 2.2, we get $\delta \geq 1/(2\sqrt{2} - 2 + 3/(\sqrt{2} \lg B_1))$. The lower bound follows by letting B_1 grow to infinity. □

Lemma 2.4 *If a search algorithm on a search structure for block sizes $B_1, B_2, \ldots, B_k$, where $B_i = {B_1}^{c_i}$ and $1 = c_1 < c_2 < \cdots < c_k \leq 2$, guarantees that the average number of block reads for a search is at most $\delta \log_{B_i} N$ for each block size B_i, then*

$$\delta \geq \frac{1}{\sum_{i=1}^{k-1} \frac{c_{i+1}}{c_i} + \frac{2}{c_k} \left(1 + \frac{\lg(8k)}{2 \lg B_1} \right) - k} \; .$$

Proof: The proof is a generalization of the proof of Lemma 2.2 for two block sizes, and we here assume familiarity with that proof. The transformation is basically the same, except that we have a token τ_i, $i = 1, \ldots, k$, for each of the k block sizes.

Again, a node v is marked if none of its ancestors access the size B_1 block accessed by v, i.e. if v is the first access to the block. The node v may also be the first access to blocks of larger sizes, and we mark v by τ_i, where B_i is the largest block size for which this is true. Note that v must be the first access to the size B_j block accessed by v for all j with $1 \leq j \leq i$.

For any root-to-leaf path, let b_i denote the number of distinct size B_i blocks accessed and let a_i denote the number of τ_i tokens on the path, for $i = 1, \ldots, k$. We have $b_i = \sum_{j=i}^{k} a_j$. Solving for a_i, we get $a_k = b_k$ and $a_i = b_i - b_{i+1}$, for $i = 1, \ldots, k-1$.

As in the proof of Lemma 2.2, the transformation proceeds in a top-down fashion, and substitutes marked nodes v by binary decision trees T_v. We now describe the trees T_v for different types of nodes v.

For a node v marked τ_k, the tree T_v resolves the relation between the query key y and a set S_v of size $(2k-1)B_1$, consisting of the B_1 elements in the block of size B_1 accessed at v, plus for $i = 2, \ldots, k$ every $\frac{B_i}{2B_1}$th element in sorted order among the elements in the block of size B_i accessed at v. This tree can be chosen to have height at most $\lceil \lg(2(2k-1)B_1 + 1) \rceil \leq \lg(8kB_1)$.

For a node v marked τ_i, $i < k$, let β_j be the block of size B_j accessed by v, for $1 \leq j \leq k$. For $i + 1 \leq j \leq k$, β_j has been accessed before, by the definition of τ_i. We now consider two cases. Case I is that β_{i+1} is the only block of size B_{i+1} that has been accessed inside β_k. By the definition of the tree T_u inserted at the ancestor u of v where β_k was first accessed, at most $B_{i+1}/2B_1 - 1$ of the elements in β_{i+1} can have unknown relations with respect to the search key y. The tree T_v inserted at v resolves these relations. It can be chosen to have height at most $\lg \frac{B_{i+1}}{B_1}$. Case II is that β_{i+1} is not the only block of size B_{i+1} that has been accessed inside β_k. Then consider the smallest j for which β_{j+1} *is* the only block of size B_{j+1} that has been accessed inside β_k. When we first time accessed the second block of size B_j inside β_k at some ancestor u of v, this access was necessarily inside β_{j+1}, and a Case I substitution as described above took place. Hence a tree T_u was inserted which resolved all relations between the search key and elements in β_{j+1}, and the empty tree can be used for T_v, i.e. v and one of its subtrees can simply be discarded.

For an unmarked node v, there is a token τ_i on the ancestor u of v in T where the size B_1 block β_1 accessed by v was first accessed. This gave rise to a tree T_u in the transformation, and this tree resolved the relations between the search key and all elements in β_1, either directly ($i = k$) or by resolving the relations for all elements in a block containing β_1 ($1 \leq i < k$), so v and one of its subtrees can be discarded.

After transformation and final pruning, the length of a root-to-leaf path in the final tree is bounded by the following equation.

$$
\begin{aligned}
& a_k \lg(8kB_1) + \sum_{i=1}^{k-1} a_i \lg \frac{B_{i+1}}{B_1} \\
= \quad & b_k \lg(8kB_1) + \lg B_1 \sum_{i=1}^{k-1} (b_i - b_{i+1})(c_{i+1} - 1) \\
= \quad & \lg B_1 \left[b_k \left(1 + \frac{\lg(8k)}{\lg B_1}\right) + b_1(c_2 - 1) \right. \\
& \qquad \left. + \sum_{i=2}^{k-1} b_i(c_{i+1} - c_i) - b_k(c_k - 1) \right] \\
= \quad & \lg B_1 \left[\sum_{i=1}^{k-1} b_i(c_{i+1} - c_i) + b_k \left(2 + \frac{\lg(8k)}{\lg B_1} - c_k\right) \right]
\end{aligned}
$$

For all i, the average value of b_i over all search paths is by assumption bounded by $\delta \log_{B_i} N = (\delta \log_{B_1} N)/c_i$, and the coefficient of b_i is positive, so we get the following bound on the average number of comparisons on a search path.

$$
\begin{aligned}
& \delta \log_{B_1} N \lg B_1 \left[\sum_{i=1}^{k-1} \frac{1}{c_i}(c_{i+1} - c_i) \right. \\
& \qquad \left. + \frac{1}{c_k} \left(2 + \frac{\lg(8k)}{\lg B_1} - c_k\right) \right] \\
= \quad & \delta \lg N \left[\sum_{i=1}^{k-1} \frac{c_{i+1}}{c_i} + \frac{1}{c_k} \left(2 + \frac{\lg(8k)}{\lg B_1}\right) - k \right]
\end{aligned}
$$

By Lemma 2.1 we have

$$
\delta \lg N \left[\sum_{i=1}^{k-1} \frac{c_{i+1}}{c_i} + \frac{1}{c_k} \left(2 + \frac{\lg(8k)}{\lg B_1}\right) - k \right] \geq \lg N ,
$$

and the lemma follows. □

Theorem 2.5 *If a search algorithm on a search structure guarantees, for all block sizes B, that the average number of block reads for a search is at most $\delta \log_B N$, then $\delta \geq \lg e \approx 1.443$.*

Proof: Let k be an integer, and for $i = 1, \ldots, k$ define $B_i = 2^{k+i-1}$. In particular, we have $B_i = {B_1}^{c_i}$ with $c_i = (k+i-1)/k$. Consider the following subexpression of Lemma 2.4.

$$
\begin{aligned}
& \frac{2}{c_k} \left(1 + \frac{\lg(8k)}{2 \lg B_1}\right) + \sum_{i=1}^{k-1} \frac{c_{i+1}}{c_i} - k \\
= \quad & \frac{2k}{2k-1} \left(1 + \frac{\lg(8k)}{2k}\right) + \sum_{i=1}^{k-1} \frac{k+i}{k+i-1} - k \\
= \quad & \frac{2k}{2k-1} \left(1 + \frac{\lg(8k)}{2k}\right) - 1 + \sum_{i=1}^{k-1} \frac{1}{k+i-1} \\
\leq \quad & \frac{2k}{2k-1} \left(1 + \frac{\lg(8k)}{2k}\right) - 1 + \int_{k-1}^{2k-2} \frac{1}{x}\, dx \\
= \quad & \frac{2k}{2k-1} \left(1 + \frac{\lg(8k)}{2k}\right) - 1 + \ln 2
\end{aligned}
$$

Letting k grow to infinity Lemma 2.4 implies $\delta \geq 1/\ln 2 = \lg e$. □

3 Upper Bound

In this section we give a tight analysis of the van Emde Boas layout of Prokop [29]. Then we propose and analyze a generalized van Emde Boas layout.

In Prokop's vEB layout, we split the tree evenly by height, except for roundoff. Thus, a tree of height h is split into a top tree of height $\lceil h/2 \rceil$ and bottom tree of height $\lfloor h/2 \rfloor$. In Subsection 3.1 we analyze this method. It is shown in [14, 15, 19] that the number of memory transfers for a search is $4 \log_B N$ in the *worst case*; we give a matching configuration showing that this analysis is tight. We then consider the average-case performance over starting positions of the tree in memory, and we show that the expected search cost is $2(1 + 3/\sqrt{B}) \log_B N + O(1)$ memory transfers, which is tight within a $o(1)$ factor. We assume that the data structure begins at a random position in memory; if there is not enough space, then the data structure "wraps around" to the first locations in memory.

In Prokop's vEB layout, the top recursive subtree and the bottom recursive subtrees have the same height (except for roundoff). At first glance this even division would seem to yield the best memory-transfer cost. Surprisingly, we can improve the van Emde Boas layout substantially by selecting different sizes for the top and bottom subtrees instead of the even split of the Prokop's vEB layout. The result is that we generate a constant approximation where the constant is significantly less than 2.

The *generalized vEB layout* is as follows: Suppose the complete binary tree contains $N-1 = 2^h - 1$ nodes and has height $h = \lg N$. Let a and b be constants such that $0 < a < 1$ and $b = 1 - a$. Conceptually we split the tree at the edges below the nodes of depth $\lceil ah \rceil$. This splits the tree into a *top recursive subtree* of height $\lceil ah \rceil$, and $k = 2^{\lceil ah \rceil}$ *bottom recursive subtrees* of height $\lfloor bh \rfloor$. Thus, there are roughly N^a bottom recursive subtrees and each bottom recursive subtree contains roughly N^b nodes. We map the nodes of the tree into positions in the array by recursively laying out the subtrees contiguously in memory. The base case is reached when the trees have one node as in the standard vEB layout.

In Subsection 3.2 we find the values of a and b, which yield a layout whose memory-transfer cost is arbitrarily close to $[\lg e + O(\lg\lg B/\lg B)] \log_B N + O(1)$ for $a = 1/2 - \epsilon$. In the full version we show that a numerical analysis within a limited range of values of N can bound this constant over an infinite range of values of N.

Memory transfers can be classified in two types. We focus our analysis on the first level of detail where recursive subtrees have size at most the block size B. There are $\mathcal{V}$ path-length memory transfers, which are caused by accessing different recursive subtrees in the level of detail, and there are $\mathcal{C}$ page-boundary memory transfers, which are caused when a single recursive subtree in this level of detail straddles two consecutive blocks. The total number of memory transfers is $\mathcal{V}+\mathcal{C}$ by linearity of expectation.

The idea of the analysis is to derive a recursive equation for the number of memory transfers $\mathcal{V}+\mathcal{C}$ from the recursive definition of the layout. It turns out that each of these components has the same general recursive expression and differs only in the base cases.

The recursive form obtained contains rounded-off terms ($\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$) that are cumbersome to analyze. We establish that if we ignore the roundoff operators, the error term is small. We obtain a solution expressed in terms of power series of the roots of the characteristic polynomial of the recurrence. We show for both $\mathcal{V}$ and $\mathcal{C}$ that the largest root is unique and hence dominates all other roots, resulting in asymptotic expressions in terms of the dominant root.

Using this asymptotic expressions, we obtain the main result, namely a layout whose total cost is arbitrarily close to $[\lg e + O(\lg\lg B/\lg B)] \log_B N + O(1)$ as the split factor $a = 1/2 - \epsilon$ approaches $1/2$. This matches the lower bound from the previous section up to low-order terms. However, in our experiments we observed that the $O(1)$ factor becomes too large when a is too near to $1/2$. Based on preliminary simulations, a reasonable tradeoff is $a = 3/7$.

3.1 Exact Analysis of van Emde Boas Layout

A simple analysis of this layout shows that the number of memory transfers of this layout, in the worst case, is no greater than four times that of the optimal *cache-size-aware* layout. More formally,

Theorem 3.1 *Consider an $(N-1)$-node complete binary search tree that is stored using the Prokop vEB layout. A search in this tree has memory-transfer cost of $\left(4 - \frac{4}{2+\lg B}\right) \log_B N$ in the worst case.*

Proof: The upper bound has been established before in the literature [14, 15, 19]. For the lower bound we show that this value is achieved asymptotically. Let the block size be $B = \left(2^{2k} - 1\right)/3$ for any odd number k and consider a tree T of size $N-1$, where $N = 2^{2k2^m}$ for some constant m. Number the positions within a block from 0 to $B-1$. As we recurse, we eventually obtain subtrees of size $3B = 2^{2k} - 1$ and one level down of size $2^k - 1$. We align the subtree of size $3B$ containing the root of T so that its first subtree of size $2^k - 1$ (which also contains the root of T) starts in position $B-1$ of a block. In other words, any root-to-leaf search path in this subtree crosses the block boundary because the root is in the last position of a block. Consider the

$\left(\frac{2^k+1}{3}+1\right)$-th subtree of size 2^k-1. The root of this tree starts at position $B-1+(2^k-1)(2^k+1)/3 = 2B-1$, which is also the last position of a block. Thus, any root-to-leaf search path in this subtree crosses the block boundary. Observe that because trees are laid out consecutively, and $3B$ is a multiple of the block size, all other subtrees of size $3B$ start at position $B-1$ inside a block and share the above property (that we can find a root-to-leaf path that has cost 4 inside this size-$3B$ subtree). Notice that a root-to-leaf path accesses 2^m many size-$3B$ subtrees, and if we choose the path according to the above position we know that the cost inside each size $3B$ subtree is 4. More precisely, each size 2^k-1 subtree on this path starts at position $B-1$ in a block. Thus, the total cost is $4\cdot 2^m = 4\log_{3B+1} N = 4(\log_{3B+1} B)\log_B N \le 4\left(1-\frac{1}{2+\lg B}\right)\log_B N$. □

However, few paths in the tree have this property, which suggests that in practice, the Prokop vEB layout results in a much lower memory-transfer cost assuming random placement in memory. We formalize this notion as follows:

Claim 3.2 *Let B be a power of 2, t and t' be positive numbers satisfying $t/2 \le t' \le 2t$, $\sqrt{B}/2 \le t \le \sqrt{B}$, and $t\cdot t' \ge B$. Then*

$$2+\frac{t+t'}{B} \le 2\left(1+\frac{3}{\sqrt{B}}\right)\frac{\lg t+\lg t'}{\lg B}.$$

Theorem 3.3 *Consider a path in an $(N-1)$-node binary complete search tree of height h that is stored in vEB layout, with the initial page starting at a uniformly random position in a block B. Then the expected memory-transfer cost of the search is at most $2(1+3/\sqrt{B})\log_B N$.*

Proof: Although the recursion proceeds to the base case where trees have height 1, conceptually we stop the recursion at the level of detail where each recursive subtree has at most B nodes. Define t so that the number of nodes in T is $t-1$; thus the height of T is $\lg t$. Therefore, any recursive subtree T has $(t-1)$-nodes, where $\sqrt{B}/2 \le t \le B$. Note that because of roundoff, we cannot guarantee that $\sqrt{B} \le t$. In particular, if a tree has $B+1$ nodes and the height h is odd, then the bottom trees have height $\lfloor h/2\rfloor$, and therefore contain roughly $\sqrt{B}/2$ nodes. Then there are $t-2$ initial positions for the upper tree that results in T being laid out across a block boundary. Similarly there are $B-t+2$ positions in which the block does not cross a block boundary. Hence, the local expected cost of accessing T is

$$\frac{2(t-2)}{B}+\frac{B-t+2}{B} = 1+\frac{t-2}{B}.$$

If $\sqrt{B}/2 \le t < \sqrt{B}$ for the recursive subtree T, we consider the next larger level of detail. There exists another recursive subtree T' immediately above T on the search path in this level of detail. Notice that $tt' \ge B$. Because otherwise conceptually there is no conceptual recursion splitting into T and T'. Also because we always cut in the middle, we know that $2t' \ge t \ge \frac{1}{2}t'$. From Lemma 3.2 the expected cost of accessing T and T' is

$$1+\frac{t-2}{B}+1+\frac{t'-2}{B} \le 2\left(1+\frac{3}{\sqrt{B}}\right)\frac{\lg(tt')}{\lg B}.$$

If $\sqrt{B} \le t < B$ for the recursive subtree T, define

$$f(x) = 2\frac{\lg x}{\lg B}\left(1+\frac{1}{\sqrt{B}}\right)-1-\frac{t-2}{B}.$$

By calculating $f''(x)$ we learn that $f(x) \le 0$ for the entire range $\sqrt{B} \le x \le B$. Thus, the expected cost of accessing T is at most $2(1+1/\sqrt{B})\lg t/\lg B$.

Combining the above arguments, we conclude that although the recursive subtrees on a search path may have different sizes, their expected memory-transfer cost is at most

$$\sum_T 2\left(1+\frac{3}{\sqrt{B}}\right)\frac{\lg t}{\lg B} = 2\left(1+\frac{3}{\sqrt{B}}\right)\log_B N.$$

This is a factor of $2(1+3/\sqrt{B})$ times the (optimal) performance of a B-tree. □

3.2 Analysis of Generalized vEB Layout

We now analyze the generalized vEB layout. In Theorems 3.1 and 3.3 we focus on the first level of detail where recursive subtrees have sizes less than B. If a recursive subtree crosses a block boundary, then we assume its block cost is 2. Thus, the expected block cost of accessing a recursive subtree T, where $|T| = t-1$, is at most $1+(t-2)/B$. If the height of a recursive subtree is $x = \lg t$, where $1 \le x \le \lg B$, then the block cost $\mathcal{B}(x)$ for this subtree is at most $\mathcal{B}(x) = 1+\frac{2^x-2}{B}$. Note that by linearity of expectation the expected memory-transfer cost $\mathcal{B}(x)$ satisfies $\mathcal{B}(x) = \mathcal{B}(\lceil ax\rceil)+\mathcal{B}(\lfloor bx\rfloor)$ for $x > \lg B$.

3.2.1 Where Memory Transfers Come From: Path-Length and Block-Boundary-Crossing Functions

We decompose the cost of $\mathcal{B}(x)$. Let $\mathcal{V}(x)$ be the number of recursive subtrees visited along a root-to-leaf path ($\mathcal{V}$ stands for "vertical"), i.e.,

$$\mathcal{V}(x) = \mathcal{V}(\lceil ax \rceil) + \mathcal{V}(\lfloor bx \rfloor)$$

with $\mathcal{V}(x) = 1$ for $1 \le x \le \lg B$. Let $\mathcal{C}(x)$ be the expected number of subtrees straddling block boundaries along the root-to-leaf path ($\mathcal{C}$ stands for "crossing"), with

$$\mathcal{C}(x) = \mathcal{C}(\lceil ax \rceil) + \mathcal{C}(\lfloor bx \rfloor).$$

Hence, by linearity of expectation $\mathcal{B}(x) = \mathcal{V}(x) + \mathcal{C}(x)$ for all $x \ge 1$. It is easy to see that the three recursive functions above are monotonically increasing.

The recurrences describing the functions $\mathcal{B}(x)$, $\mathcal{V}(x)$, and $\mathcal{C}(x)$ are of the form $\mathcal{F}(x) = \mathcal{F}(\lceil ax \rceil) + \mathcal{F}(\lfloor bx \rfloor)$, for $0 < a \le b < 1$ and $a + b = 1$. The floor and ceiling in the recurrence make the analysis more complicated. As we will see, it is easier to analyze the recurrence $\mathcal{G}(x) = \mathcal{G}(ax) + \mathcal{G}(bx)$ with the roundoff removed. The base cases for the recursively defined functions $\mathcal{F}(x)$ and $\mathcal{G}(x)$ are the range $1 \le x \le \lg B$, in which $\mathcal{F}(x) = \mathcal{G}(x)$.

3.2.2 Roundoff Error Is Small

We analyze the difference between these two functions $\mathcal{F}(x)$ and $\mathcal{G}(x)$ defined above, and we show that the difference is small.

Definition Let $a < \min\{1/2, 1 - 2/\lg B\}$. Define the recursive function $\beta(x)$ and $\delta(x)$ as follows:

$$\beta(x) = \begin{cases} 0, & x \le \lg B; \\ \beta(ax+1)+1, & x > \lg B. \end{cases}$$

$$\delta(x) = \begin{cases} 1, & x \le \lg B; \\ \delta(ax+1)(1 + 2\frac{a^{\beta(x)-2}}{\lg B}), & x > \lg B. \end{cases}$$

Lemma 3.4 *For all $x > \lg B$, the function $\beta(x)$ satisfies*

$$\frac{2}{a^2 x} \ge \frac{a^{\beta(x)-2}}{\lg B} \ge \frac{1}{2ax}.$$

Lemma 3.5 *The function $\delta(x)$ has the following properties:*
(1) If $\beta(x) = \beta(y)$, then $\delta(x) = \delta(y)$.
(2) For all $x > \lg B$,

$$\delta(ax+1)(ax+1) \le ax\delta(x).$$

(3) For all $x \ge 1$,

$$\delta(x) \le e^{\frac{2}{a(1-a)\lg B}},$$

which is $1 + O\left(\frac{2}{a(1-a)\lg B}\right) = 1 + O(1/\lg B)$.

Proof: (1) This claim follows from induction.
(2) This claim follows from the recursive definition of $\delta(x)$ and

$$\frac{a^{\beta(x)-2}}{\lg B} \ge \frac{1}{2ax}.$$

(3) For all $x > \lg B$,

$$\frac{\delta(x)}{\delta(ax+1)} = 1 + 2 \cdot \frac{a^{\beta(x)-2}}{\lg B} \le \exp\left(2\frac{a^{\beta(x)-2}}{\lg B}\right).$$

Thus, $\delta(x) \le \exp\left(2 \sum_{i=1}^{\beta(x)} \frac{a^{i-2}}{\lg B}\right) \le \exp\left(\frac{2}{a(1-a)\lg B}\right)$. □

Theorem 3.6 (Roundoff Error) *Let $\mathcal{F}(x) = \mathcal{F}(\lceil ax \rceil) + \mathcal{F}(\lfloor bx \rfloor)$, and $\mathcal{G}(x) = \mathcal{G}(ax) + \mathcal{G}(bx)$, for $0 < a \le b < 1$, and $a + b = 1$. Then for all $x \ge 1$, $\mathcal{F}(x) \le \mathcal{G}(x\,\delta(x))$.*

Proof: We prove the bound inductively. First recall that $\mathcal{F}(x)$ and $\mathcal{G}(x)$ are monotonically increasing. The base case is $\mathcal{F}(x) = \mathcal{G}(x)$ and $\delta(x) = 1$ when $x \le \lg B$. If $\mathcal{F}(x) \le \mathcal{G}(x\delta(x))$ when $x \le t$, then for all $\lg B < x \le t/a - 1$, we have

$$\begin{aligned} \mathcal{F}(x) &= \mathcal{F}(\lceil ax \rceil) + \mathcal{F}(\lfloor bx \rfloor) \le \mathcal{F}(ax+1) + \mathcal{F}(bx) \\ &\le \mathcal{G}((ax+1)\delta(ax+1)) + \mathcal{G}(bx\delta(bx)) \\ &\le \mathcal{G}(ax\delta(x)) + \mathcal{G}(bx\delta(x)) \\ &= \mathcal{G}(x\delta(x)). \end{aligned}$$

Thus, for all $x \ge 1$, we have $\mathcal{F}(x) \le \mathcal{G}(x\delta(x))$. Furthermore, if $\mathcal{G}(x) \le c\,x + O(1)$, then by of Lemma 3.5 Condition 3, $\mathcal{F}(x) \le \mathcal{G}(x\delta(x)) \le c\,x\,\delta(x) + O(1) \le c[1 + O(1/\lg B)]x + O(1)$. □

3.2.3 Bounding the Path-Length and the Page-Boundary Crossing Function

We now develop methods so that given values for a and b, we can determine values of the approximation ratio. We restrict ourselves to splits a and b of the form $a = \frac{1}{q^k}$ and $b = \frac{1}{q^m}$, for integers m and k that are relatively prime and $q > 1$. Notice that $k > m$, and thus $q^k = q^n + 1$, where $n = k - m$. The rationale behind this choice is that this additional structure helps us in the analysis while still being dense; that is, for any given a' and b', we can find a and b defined as above that are as close as we want to a' and b'. We call such an (a, b) pair a *twin power pair*.

We ignore the roundoff based on Corollary 3.6. Furthermore, we normalize the range for which $\mathcal{V}(x) = 1$ by introducing a function $H(x) = H(ax) + H(bx)$ with $H(x) = 1$ for $0 < x \leq 1$ as desired. Note that $\mathcal{V}(x \lg B) \leq H(x\delta(x \lg B))$ by Theorem 3.6.

First we state a lemma, which we prove later in this subsection.

Lemma 3.7 *Let $(1/q^k, 1/q^m)$ be a twin power pair, and let $n = k - m$. Then, we have $H(x) \leq (c_1 + \epsilon)q^k x + O(1)$, where the value of c_1 is*

$$\left(\sum_{i=1}^{n} q^{-i} + \sum_{i=n+1}^{k} q^{k-i}\right) / \left(kq^{k-1} - nq^{n-1}\right).$$

Corollary 3.8 *The number of recursive subtrees $\mathcal{V}(x)$ on a root-to-leaf path is bounded by $(c_1 + \epsilon)q^k \log_B N + O(1)$.*

Theorem 3.9 (Path-Length Cost) *The number of recursive subtrees on a root-to-leaf path is* $(\lg e + \epsilon)\log_B N + O(1) \approx 1.443 \log_B N + O(1)$.

Proof: Let $a = 1/q^k$ and $b = 1/q^{k-1}$, where $1/q^k + 1/q^{k-1} = 1$. From this we have $q \approx 1 + \ln 2/k$. Applying Lemma 3.7 and for large k we obtain $c_1 q^k \xrightarrow{k\to\infty} 1/\ln 2 = \lg e$ and hence $\mathcal{V}(x) \to \lg e \log_B N + O(1) \approx 1.443 \log_B N + O(1)$ as claimed. □

To complete the proof of Lemma 3.7, we establish some properties of $H(x)$. Since $H(x)$ is monotonically increasing we can bound the value $H(x)/x$ for $q^i \leq x \leq q^{i+1}$ as follows:

$$\frac{1}{q}\min\left\{\frac{H(q^i)}{q^i}, \frac{H(q^{i+1})}{q^{i+1}}\right\} \leq \frac{H(q^i)}{q^{i+1}} \leq \frac{H(x)}{x} \leq \frac{H(q^{i+1})}{q^i} \leq q \max\left\{\frac{H(q^i)}{q^i}, \frac{H(q^{i+1})}{q^{i+1}}\right\}.$$

Hence, if d is a lower bound and c is an upper bound on $H(q^i)/q^i$ when i is larger than a given integer s, then d/q is a lower bound on $H(x)/x$ and cq is an upper bound on $H(x)/x$ when $x > q^s$.

Define $\alpha_i = H(q^{i-k+1})$. From the recursive formula for $H(x)$ we know that for $i \geq 0$,

$$\begin{aligned}\alpha_{i+k} &= H(q^{i+1}) = H(aq^{i+1}) + H(bq^{i+1}) \\ &= H(q^{i-k+1}) + H(q^{i+n-k+1}) = \alpha_{i+n} + \alpha_i.\end{aligned}$$

Let $r_1, r_2, \ldots, r_k$ be the (possibly complex) roots of the characteristic polynomial function $w(x) = x^k - x^n - 1$. We will claim they are all unique.

The following four lemmas have technical proofs, which appear in the full version.

Lemma 3.10 *The k roots of $w(x) = x^k - x^n - 1$ are unique, when k and n are relatively prime integers such that $1 \leq n < k$.*

Because $h'(x) = kx^{k-1} - nx^{n-1} > 0$ when $x > 1$ and by construction $q > 1$, there is one unique root $q > 1$ of $w(x)$. Without loss of generality let $r_1 = q$.

We now show that if the k roots of the characteristic polynomial function of a series are unique, then the series in question is a linear combination of power series $\{r_j^i\}$ of the roots.

Lemma 3.11 *Consider a series $\{\alpha_i\}$ satisfying $\alpha_{k+s} = \sum_{i=0}^{k-1} d_i \alpha_{i+s}$ for complex numbers d_i, and let $r_1, r_2, \ldots, r_k$ be the k unique roots of the characteristic function $g(x) = x^k - \sum_{i=0}^{k-1} d_i x^i$ for the series $\{\alpha_i\}$. Then there exists complex numbers $c_1, c_2, \ldots, c_k$ such that for all i, $\alpha_i = \sum_{j=1}^{k} c_j r_j^i$.*

Hence we can solve the recurrence $\{\alpha_i\}$ by finding c_i that satisfy $\alpha_i = \sum_{j=1}^{k} c_j r_j^i$ for $i = 0, \ldots, k-1$. The base cases of $\{\alpha_i\}_{i=0}^{k-1}$ are determined by the original definition of $\alpha_i = H(q^{i-k+1})$. For $i = 0, \ldots, k-1$ we have $0 < q^{i-k+1} < 1$ and hence $H(q^{i-k+1}) = 1 = \alpha_i$.

Lemma 3.12 *The dominant root (i.e., the root with largest absolute value) for $w(x) = x^k - x^n - 1$ is $r_1 = q$. All other roots $r_2, \ldots r_k$ have absolute value less than q.*

Lemma 3.13 *The coefficient c_1 in Lemma 3.11 is*

$$\left(\sum_{i=1}^{n} q^{-i} + \sum_{i=n+1}^{k} q^{k-i}\right) / (kq^{k-1} - nq^{n-1}).$$

After establishing the properties of $H(x)$, we give the proof of Lemma 3.7:

Proof of Lemma 3.7: To complete the proof we only need to show that $H(x) \leq (c_1 + \epsilon)q^k x$, where $\epsilon = o(1)$.

Observe that the function $H(x)$ is monotonically increasing and for each $x > 1$, we have $qx \geq q^{\lceil \ln_q x \rceil} \geq x$. So $H(x) \leq \alpha_{\lceil \ln_q x \rceil + k - 1} \leq (c_1 + \epsilon) q^{\lceil \ln_q x \rceil + k - 1} \leq (c_1 + \epsilon)q^k x$, as claimed. The first inequality is from the definition of $H(x)$; the second inequality is from the equation $\alpha_i = \sum_{j=1}^{k} c_j r_j^i$ and $r_1 = q$ is the dominant root; the third inequality is from the monotonic property of $H(x)$. □

We study the memory-transfer cost from block-boundary crossings, and show that it is dominated by the the memory-transfer cost from the path length. We consider the case when $a \geq 1/4$, which includes the best layouts. Using similar reasoning for computing the path-length cost, we obtain the following theorem:

Theorem 3.14 (Block-Boundary Crossing Cost) *The expected number of block-boundary-induced memory transfers $\mathcal{C}(x)$ on a search is at most $O(\lg\lg B/\lg B)\log_B x$ when $1/4 \leq a < 1/2$.*

Combining Theorems 3.9 and 3.14, we obtain the main theorem.

Theorem 3.15 (Generalized vEB Layout) *The expected cost of a search in the generalized vEB layout is at most $(\lg e + o(1))\log_B N + O(\lg\lg B/\lg B)\log_B N + O(1)$.*

References

[1] P. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley. On cache-oblivious multidimensional range searching. In *Proc. 19th ACM Symp. on Comp. Geom. (SOCG)*, pages 237–245, 2003.

[2] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. of the 19th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 305–314, 1987.

[3] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. of the 28th Annual IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 204–216, 1987.

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

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

[6] S. Alstrup, M. A. Bender, E. D. Demaine, M. Farach-Colton, J. I. Munro, T. Rauhe, and M. Thorup. Efficient tree layout in a multilevel memory hierarchy, 2002. `http://www.arXiv.org/abs/cs.DS/0211010`.

[7] M. Andrews, M. A. Bender, and L. Zhang. New algorithms for the disk scheduling problem. In *Proc. of the 37th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 580–589, 1996.

[8] M. Andrews, M. A. Bender, and L. Zhang. New algorithms for the disk scheduling problem. *Algorithmica*, 32(2):277–301, 2002.

[9] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 268–276, 2002.

[10] R. D. Barve and J. S. Vitter. A theoretical framework for memory-adaptive algorithms. In *Proc. of the 40th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 273–284, 1999.

[11] M. Bender, R. Cole, E. Demaine, and M. Farach-Colton. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Proc. 10th Ann. European Symp. on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 139–151, 2002.

[12] M. Bender, R. Cole, and R. Raman. Exponential structures for cache-oblivious algorithms. In *Proc. 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 2380 of *LNCS*, pages 195–207, 2002.

[13] M. Bender, E. Demaine, and M. Farach-Colton. Efficient tree layout in a multilevel memory hierarchy. In *Proc. 10th Annual European Symp. on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 165–173, 2002.

[14] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. 41st Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.

[15] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 29–39, 2002.

[16] 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. of the 8th Ann. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, 1996.

[17] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 2380 of *LNCS*, pages 426–438, 2002.

[18] G. S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proc. 13th Ann. International Symp. on Algorithms and Computation (ISAAC)*, volume 2518 of *LNCS*, pages 219–228, 2002.

[19] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 39–48, 2002.

[20] E. D. Demaine. Cache-oblivious algorithms and data structures. Unpublished manuscript, June 2002.

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

[22] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. of the 13th Ann. ACM Symp. on Theory of Computation (STOC)*, pages 326–333, 1981.

[23] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. 3rd edition, 1997.

[24] P. Kumar. Cache oblivious algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies, LNCS 2625*, pages 193–212, 2003.

[25] P. Kumar and E. Ramos. I/O efficient construction of Voronoi diagrams. Unpublished manuscript, July 2002.

[26] R. Ladner, R. Fortna, and B.-H. Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *Algorithm Design to Robust and Efficient Software*, volume 2547 of *LNCS*, pages 78–92, 2002.

[27] R. E. Ladner, J. D. Fix, and A. LaMarca. Cache performance analysis of traversals and random accesses. In *Proc. of the Tenth Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 613–622, 1999.

[28] A. LaMarca and R. E. Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1):66–104, 1999. An earlier version appear in SODA 97.

[29] H. Prokop. Cache oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[30] N. Rahman, R. Cole, and R. Raman. Optimised predecessor data structures for internal memory. In *Proc. 5th Int. Workshop on Algorithm Engineering (WAE)*, volume 2141, pages 67–78, 2001.

[31] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, 1994.

[32] J. E. Savage. Extending the Hong-Kung model to memory hierachies. In *Proc. of the 1st Ann. International Conference on Computing and Combinatorics*, volume 959 of *Lecture Notes in Computer Science*, pages 270–281, 1995.

[33] S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *Proc. of the 11th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 829–838, 2000.

[34] R. C. Singleton. An algorithm for computing the mixed radix fast fourier transform. *IEEE Transactions on Audio and Electroacoustics*, AU-17(2):93–103, 1969.

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

[36] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proc. of the 16th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 75–84, 1975.

[37] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[38] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2), 2001.

[39] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

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