Alcom-FT Technical Report Series
ALCOMFT-TR-03-74

On the Limits of Cache-Obliviousness

Gerth Stølting Brodal*,† Rolf Fagerberg*

Abstract

In this paper, we present lower bounds for permuting and sorting in the cache-oblivious model. We prove that (1) I/O optimal cache-oblivious comparison based sorting is not possible without a tall cache assumption, and (2) there does not exist an I/O optimal cache-oblivious algorithm for permuting, not even in the presence of a tall cache assumption.

Our results for sorting show the existence of an inherent trade-off in the cache-oblivious model between the strength of the tall cache assumption and the overhead for the case $M \gg B$, and show that Funnelsort and recursive binary mergesort are optimal algorithms in the sense that they attain this trade-off.

Categories and Subject Descriptors

F.1.1 [Theory of Computation]: Models of Computation—*Relations between models*

General Terms

Theory, Algorithms

Keywords

Cache-oblivious model, lower bound, tall cache assumption, sorting, permuting

1 Introduction

Modern computers contain a hierarchy of memory levels, with each level acting as a cache for the next. Typical components of the memory hierarchy are: registers, level 1 cache, level 2 cache, level 3 cache, main memory, and disk. The time for accessing a level increases for each new level (most dramatically when going from main memory to disk), making the cost of a memory access depend highly on what is the current lowest memory level containing the element accessed.

As a consequence, the memory access pattern of an algorithm has a major influence on its running time in practice. Since classic asymptotic analysis of algorithms in the RAM model (depicted in Figure 1) is unable to capture this, a number of more elaborate models for analysis have been proposed. The most widely used of these is the I/O model introduced by of Aggarwal and Vitter [2] in 1988, which assumes a memory hierarchy containing two levels, the lower level having size M and the transfer between the two levels taking place in blocks of B consecutive elements. This model is illustrated in Figure 2.

*BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: {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).

The cost of the computation in the I/O model is the number of blocks transferred between the two memory levels. The strength of the model is that it captures part of the memory hierarchy (in particular, it adequately models the situation where the memory transfer between two levels of the memory hierarchy dominates the running time), while being sufficiently simple to make analysis of algorithms feasible. By now, a large number of results for the I/O model exists—see the surveys by Arge [3] and Vitter [24].

Among the fundamental facts are that in the I/O model, comparison based sorting takes $\Theta(\mathrm{Sort}_{M,B}(N))$ I/Os in the worst case, where $\mathrm{Sort}_{M,B}(N) = \frac{N}{B}\log_{M/B}\frac{N}{B}$, and permuting takes $\Theta(\mathrm{Perm}_{M,B}(N))$ I/Os in the worst case, where $\mathrm{Perm}_{M,B}(N) = \min\{N\ ,\ \mathrm{Sort}_{M,B}(N)\}$ [2].

More elaborate models for multi-level memory hierarchies have been proposed ([24, Section 2.3] gives an overview), but fewer analyses of algorithms have been done. For these models, as for the I/O model of Aggarwal and Vitter, algorithms are assumed to know the characteristics of the memory hierarchy.

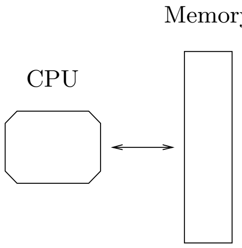

Figure 1: The RAM model

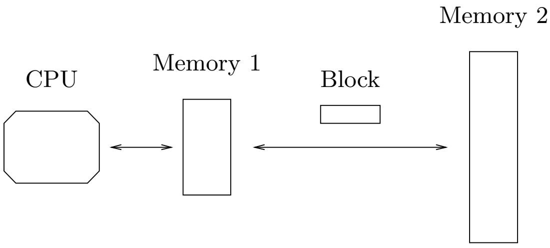

Figure 2: The I/O model

Recently, the concept of *cache-oblivious* algorithms has been introduced by Frigo et al. [18]. In essence, this designates algorithms formulated in the RAM model, but analyzed in the I/O model for arbitrary block size B and memory size M. I/Os are assumed to be performed automatically by an offline optimal cache replacement strategy. This seemingly simple change has significant consequences: since the analysis holds for any block and memory size, it holds for *all* levels of a multi-level memory hierarchy (see [18]). In other words, by optimizing an algorithm to one unknown level of the memory hierarchy, it is optimized to each level automatically. Thus, the cache-oblivious model in a elegant way combines the simplicity of the I/O-model with a coverage of the entire memory hierarchy. An additional benefit is that the characteristics of the memory hierarchy do not need to be known, and do not need to be hardwired into the algorithm for the analysis to hold, which increases the portability of implementations of the algorithm.

Frigo et al. introduced the concept of cache-oblivious algorithms in 1999, and presented optimal cache-oblivious algorithms for matrix transposition, FFT, and sorting [18], and also gave a proposal for static search trees [21] with search cost matching that of standard (cache-aware) B-trees [7]. Since then, quite a number of results for the model have appeared, including the following: Bender et al. [11] gave a proposal for cache-oblivious dynamic search trees with search cost matching B-trees. Simpler cache-oblivious search trees with complexities matching that of [11] were presented in [12, 16, 22], and a variant with worst case bounds for updates appear in [9]. Cache-oblivious algorithms have been given for problems in computational geometry [1, 9, 14], for scanning dynamic sets [8], for layout of static trees [10], and for partial persistence [9]. Cache-oblivious priority queues have been developed in [4, 15], which in turn gives rise to several cache-oblivious graph algorithms [4]. Some of these results, in particular those involving sorting and algorithms to which sorting reduces, such as priority queues, are proved under the assumption $M \geq B^2$, which is also known as the *tall cache assumption.* In particular, this applies to the Funnelsort algorithm of Frigo et al. [18]. A variant termed Lazy Funnelsort [14] works under the weaker tall cache assumption $M \geq B^{1+\varepsilon}$ for any fixed $\varepsilon > 0$, at the cost of a $1/\varepsilon$ factor compared to the optimal sorting bound $\Theta(\mathrm{Sort}_{M,B}(N))$ for the case $M \gg B^{1+\varepsilon}$.

In [18], Frigo et al. raised the question of the complexity theoretic relationship between cache-oblivious algorithms and cache-aware algorithms. Clearly, cache-aware algorithms can only use caches better than cache-oblivious algorithms, since they have more knowledge about the system on which they are running. Frigo et al. asked whether there is a *separation* between the two classes, i.e. a problem for which the asymptotical I/O complexity for all cache-oblivious algorithms is worse than for the best cache-aware.

In this paper, we prove such a separation for the two fundamental problems of comparison based sorting and permuting. Specifically, we prove (1) I/O optimal cache-oblivious comparison based sorting is not possible without a tall cache assumption, and (2) there does not exist an I/O optimal cache-oblivious algorithm for permuting, not even in the presence of a tall cache assumption.

At a more detailed level (see Section 3), our results for sorting show the existence of an inherent trade-off in the cache-oblivious model between the strength of the tall cache assumption and the overhead for the case $M \gg B$, and show that Lazy Funnelsort and recursive binary mergesort are optimal algorithms in the sense that they attain this trade-off.

Only little previous work has been done on the question. Bilardi and Peserico [13] have investigated the portability of algorithms in the HRAM-model, where the access to memory location $A[i]$ takes time $f(i)$ for some non-decreasing function f. Their model of computation is the CDAG, which is a directed acyclic graph describing the dependencies of the individual operations of a straight-line program. They give a specific CDAG and two different HRAM machines for which they prove that any fixed scheduling of the operations of the CDAG will be sub-optimal by a factor polynomial on N on at least one of the machines.

A basic element of our approach is the transformation of comparison trees, which was inspired by the work of Arge et al. [5] for the I/O model. Key new features of our proofs are our definition of a working set, bounds on online searches, and a formal model for comparison based cache-oblivious sorting.

This paper is organized as follows: In Section 2, we make precise our model of cache-obliviousness. In Section 3, we state our main theorems and prove implications of them. In

Section 4, we give the proofs of our main theorems.

2 Definitions

We define a cache-oblivous algorithm to be simply an algorithm formulated in the classic RAM model, consisting of an CPU with some specified set of basic operations, and a single level of memory viewed as an array A of cells, as shown in Figure 1.

We will study comparison based sorting in this model. Informally, this corresponds to restricting the set of operations of the CPU w.r.t. elements to copying elements between memory locations, and comparing elements in two memory locations. Formally, we define the concept of *RAM-decision-trees*, and let that be our model of comparison based cache-oblivious algorithms.

Definition 1 *A* RAM-decision-tree *is a rooted tree, with a unary root called the* start node*, and with the other nodes being of the following types:*

- Comparison nodes*, which are nodes of degree two, labeled with a pair (i, j) of indices. Such a node models the comparison $A[i] \leq A[j]$, with the left subtree corresponding to a positive outcome.*
- Assignment nodes*, which are nodes of degree one, labeled with a pair (i, j) of indices. Such a node models the assignment $A[i] := A[j]$.*
- Result nodes*, which are nodes of degree zero, labeled with the answer of the computation.*

In the case of sorting, the label of a result node is the permutation of the input elements. We assume that the input elements $x_0, x_1, \ldots, x_{N-1}$ initially are located in the first locations of the array (i.e. $A[i] = x_i$ for $i < N$), and that the remaining locations are undetermined (i.e. $A[i] = \perp$ for $i \geq N$, where $\perp$ is some value which is not allowed to take part in comparisons). A RAM-decision-tree is said to be a correct sorting algorithm for input size N if for any input of size N, the root-to-leaf path determined by the input ends in a leaf labeled with the initial permutation of the input elements. Clearly, a RAM-decision-tree can be transformed into a standard decision-tree (see e.g. [6]) by keeping track of elements positions, converting indices for memory cells into indices for elements, and finally removing the assignment nodes. In short, RAM-decision-trees are a version of decision trees with explicit references to memory locations (which is necessary to capture I/O issues). By pruning subtrees which cannot be reached by any input, we may assume that a RAM-decision-tree which is a correct sorting algorithm has exactly $N!$ leaves.

In the cache-oblivious model, paging is taking place (even though the algorithm is ignorant hereof), and the goal is to bound the number of I/Os (page swaps) for any values of B and M. We now add paging to our model. We assume that the memory A is divided into contiguous blocks of B memory cells, such that the kth block comprises the memory cells $A[kB], A[kB + 1], \ldots, A[(k + 1)B - 1]$, for $0 \leq k$. The cache has room for $M/B = m$ blocks.

For simplicity, and without loss of generality, we in the model assume that there always are exactly m blocks in the cache, and that its initial contents is the first m blocks of A. An I/O is specified by an ordered pair (k, l) of block indices, where block k is currently not in cache, and block l is currently in cache. The effect of this I/O is to move block k into the cache and block l out of the cache.

Definition 2 *Given a RAM-decision-tree T, a* paging *is an annotation of each edge of T with a sequence (possible of length zero) of I/Os, such that for any decision node or assignment node v with label (i, j), the two memory locations $A[i]$ and $A[j]$ are in blocks currently residing in memory.*

In the above, the word "currently" has the natural meaning given by interpreting each root-to-leaf path as a timeline.

Note that our definition of paging captures online paging strategies. This is in contrast to the original definition of cache-obliviousness [18], where optimal offline paging is assumed (corresponding to edges having one annotation for *each* leaf below it in the RAM-decision-tree). Allowing only online paging strategies is arguably a more realistic definition of the cache-oblivious model, and does not affect the existing upper bounds, as the proofs of these still work if the online LRU paging policy is assumed instead of optimal offline paging. However, if the original definition is preferred, we can convert our results to it at the cost of a factor two in M and in the I/O bound, by appealing to Sleator and Tarjans classic competitiveness result [23] for LRU-paging. Also note that like [18], we assume the cache is fully associative.

3 Results

We first consider the problem of comparison based sorting. Below, in Theorem 1 and it corollaries, we show that cache-oblivious comparison based sorting is not possible without a tall cache assumption.

We first note that if the Lazy Funnelsort algorithm of [14] is tuned to the tall cache assumption $M \geq B^{1+\varepsilon}$, i.e. its parameters are chosen such that the algorithm is I/O-optimal when $M = B^{1+\varepsilon}$, then it for the case $M \gg B$ (defined for instance as $M \geq B^2$) is a factor of $\Theta(1/\varepsilon)$ worse than the optimal I/O bound.

Corollary 2 below states that this is best possible. Hence it gives a trade-off for comparison based sorting which is inherent in the cache-oblivious model—a trade-off between the strength of the tall cache assumption and the overhead for the case $M \gg B$. In particular, Corollary 2 proves that no cache-oblivious algorithm for comparison based sorting can be asymptotically I/O optimal for all values of M and B.

Corollary3 is a version of Corollary 2 focusing on the two extreme points 1 and $M/2$ of the possible range of B. It even more directly shows that I/O-optimal cache-oblivious comparison based sorting without a tall cache assumption does not exist. It also has the natural interpretation that if we want a cache-oblivious algorithm which is I/O-optimal for the case $B = M/2$, then binary mergesort (the recursive version, in order to get M in the denominator in the $\log N/M$ part of its I/O bound) is best possible—any other algorithm will be the same factor of $\Theta(\log M)$ worse than the optimal I/O bound for the case $M \gg B$.

Theorem 1 (Comparison Based Sorting) *Let T be a RAM-decision-tree which is a correct sorting algorithm for input size N. Let $\mathcal{P}_1$ be a paging with block size B_1 and memory size M, and let $\mathcal{P}_2$ be a paging with block size B_2 and memory size M, where $B_2 > B_1$ and B_1 divides B_2. If all root-to-leaf paths contain at most t_1 I/Os in $\mathcal{P}_1$ and at most t_2 I/Os in $\mathcal{P}_2$, then the following holds:*

$$8t_1B_1 + 3t_1B_1 \log \frac{8Mt_2}{B_1t_1} \geq N \log \frac{N}{M} - 1.45N \ .$$

Corollary 1 *For any cache-oblivious comparison based sorting algorithm, let t_i be an upper bound on the number of I/Os performed for block size B_i. If for some real number $c > 0$ we have*

$$t_1 = c \cdot \mathrm{Sort}_{M,B_1}(N) ,$$

then

$$t_2 \geq \mathrm{Sort}_{M,B_2}(N) \cdot \frac{\log(M/B_2)}{\log(M/B_1)} \cdot \frac{cB_2}{8M} \cdot (M/B_1)^{1/8c} ,$$

under the conditions $\frac{8Mt_2}{t_1 B_1} \geq 4$ and $N \geq 2^{12}M$.

Proof. By the conditions assumed, we from Theorem 1 get

$$8t_1 B_1 \log \frac{8Mt_2}{B_1 t_1} \geq N \log \frac{N}{M} .$$

Inserting $t_1 = c\frac{N}{B_1} \log_{M/B_1} \frac{N}{M}$ and manipulating gives

$$8c \log \frac{8Mt_2}{cN \log_{M/B_1} \frac{N}{M}} \geq \log \frac{M}{B_1}$$

$$t_2^{8c} (\frac{8M}{cN \log_{M/B_1} \frac{N}{M}})^{8c} \geq \frac{M}{B_1}$$

$$t_2 \geq (\frac{M}{B_1})^{1/8c} \cdot \frac{cN \log_{M/B_1} \frac{N}{M}}{8M}$$

By the values of $\mathrm{Sort}_{M,B_1}(N)$ and $\mathrm{Sort}_{M,B_2}(N)$, the statement follows. □

Corollary 2 *Let $B_1 = 1$ and $B_2 = M^{1/(1+\varepsilon)}$ for some ε with $0 < \varepsilon < 1/2$. For any cache-oblivious comparison based sorting algorithm, let t_i be an upper bound on the number of I/Os performed for block size B_i. If for real numbers $c \geq 0$ and $d \geq 0$ we have $t_1 = c \cdot \mathrm{Sort}_{M,B_1}(N)$ and $t_2 = d \cdot \mathrm{Sort}_{M,B_2}(N)$, then we must have $c > 1/8\varepsilon$.*

Proof. Assume that $c \leq 1/8\varepsilon$. As t_1 and t_2 just need to be upper bounds, we may without loss of generality also assume $c \geq 1$ and $d \geq 1$. Using $M/B_2 = M^{\varepsilon/(1+\varepsilon)}$, it can be checked that the condition $\frac{8Mt_2}{t_1 B_1} \geq 4$ holds. The inequality from Corollary 1 gives

$$d \ \geq \ \frac{\log M/B_2}{\log M/B_1} \cdot \frac{cB_2}{8M} \cdot (M/B_1)^{1/8c} .$$

Inserting the values of B_1 and B_2 leads to

$$\frac{8d}{c} \cdot \frac{\varepsilon}{1+\varepsilon} \geq M^{1/8c - \varepsilon/(1+\varepsilon)} .$$

The assumption $c \leq 1/8\varepsilon$ implies $1/8c - \varepsilon/(1+\varepsilon) > 0$, which contradicts the inequality above for $M \to \infty$ (and $N \geq 2^{12}M$, as required by Corollary 1), since the left-hand side is a constant. □

Corollary 3 *Let* $B_1 = 1$ *and* $B_2 = M/2$. *For any cache-oblivious comparison based sorting algorithm, let* t_i *be an upper bound on the number of I/Os performed for block size* B_i. *If for a real number* $d \geq 0$ *we have* $t_2 = d \cdot \mathrm{Sort}_{M,B_2}(N)$, *then we must have* $t_1 > 1/8 \cdot N \log_2 N/M$.

Proof. Assume $t_1 \leq 1/8 \cdot N \log_2 N/M = 1/8 \cdot \log M \cdot \mathrm{Sort}_{M,B_1}(N)$. We note that the proof of Corollary 1 goes through, even if c is not at constant but a function of B, M, and N. The assumption above is equivalent to the assumption $c \leq 1/8 \cdot \log M$ (for all B, M, and N). As t_1 and t_2 just need to be upper bounds, we may without loss of generality assume $c = 1/8 \cdot \log M$ and $d \geq 1$. It can be checked by insertion that the condition $\frac{8Mt_2}{t_1 B_1} \geq 4$ holds. The inequality from Corollary 1 gives

$$d \;\geq\; \frac{\log M/B_2}{\log M/B_1} \cdot \frac{cB_2}{8M} \cdot (M/B_1)^{1/8c} \,.$$

Inserting the values of B_1 and B_2 leads to

$$16d \geq \log M \cdot c \cdot M^{1/8c} \,.$$

As $M^{1/\log M} = 2$, the assumption $c = 1/8 \cdot \log M$ contradicts the inequality above for $M \to \infty$ (and $N \geq 2^{12}M$, as required by Corollary 1), since the left-hand side is a constant. □

We now turn to the problem of permuting. The following theorem states that for all possible tall cache assumptions $B \leq M^\delta$, no cache-oblivious permuting algorithm exists with an I/O bound (even only in the average case sense) matching the worst case bound in the I/O model

Theorem 2 (Permuting) *For all* $\delta > 0$, *there exists no cache-oblivious algorithm for permuting that for all* $M \geq 2B$ *and* $1 \leq B \leq M^\delta$ *achieves* $O(\mathrm{Perm}_{M,B}(N))$ *I/Os averaged over all possible permutations of size* N.

4 Proofs

4.1 Sorting

In this section, we prove Theorem 1. Let t_1 and t_2 be upper bounds on the number of I/Os in $\mathcal{P}_1$, respectively $\mathcal{P}_2$, on any root-to-leaf path in T. We will put T and its two annotations $\mathcal{P}_1$ and $\mathcal{P}_2$ through four transformations.

The *first* transformation is to *normalize* $\mathcal{P}_1$ and $\mathcal{P}_2$. This transformation is done edge by edge in a top down fashion in T as follows, where i is either 1 or 2: For a given edge e, the net effect of the sequence σ_e of I/Os associated with e in $\mathcal{P}_i$ is to change k blocks and leave $M/B_i - k$ blocks unchanged, for some integer k between 0 and M/B_i. We first substitute σ_e by the obvious k I/Os giving the same net effect. At most two of these k I/Os move into cache a block containing an element accessed in the node v (of type comparison or assignment) at the lower end of e. These we keep at e, and the rest of the k I/Os, we push down, i.e. we append them to the sequences of I/Os in $\mathcal{P}_i$ at each of the (at most two) edges leading from v to its children (dublicating the I/Os if v is binary). When the normalization process reaches an edge e above a leaf (i.e. a result node), the I/Os at e in $\mathcal{P}_i$ are discarded. Clearly, the normalization process cannot increase the number of I/Os in $\mathcal{P}_i$ on any root-to-leaf path. In the remainder of this proof, $\mathcal{P}_1$ and $\mathcal{P}_2$ will refer to the normalized versions.

The *second* transformation is to annotate each node v in T with a *working set* $W(v)$. The working set is a set of indices of A (i.e. a set of memory locations), and is defined inductively in a top down fashion as follows: The working set of the root is $\{0, 1, 2, \ldots, M-1\}$, i.e. the initial contents of the cache. The working set for a node v with parent u is given by

$$W(v) = (W(u) \cap \text{Cache}_2) \cup \text{Cache}_1 \ ,$$

where Cache_i means the memory locations contained in the blocks residing in cache at v, according to paging $\mathcal{P}_i$. Effectively, the working set tracks the contents of Cache_1, except that indices will not leave the working set if they are still part of Cache_2.

The *third* transformation is done to ensure the following invariants, where the total order of a set of indices means the total order of the elements currently residing at these memory locations:

1. The total order of the working set is known.
2. When a block b is read into cache in $\mathcal{P}_2$, the total order is known for the parts of b which are not in the working set.

The transformation involves the substitution of nodes of T with specific RAM-decision-trees, and the copying of entire subtrees of T to each of the leaves of these added RAM-decision-trees. The annotation (I/O and working set) of T will be copied along, but the I/O annotation is not necessarily a valid paging anymore—we allow comparison of elements at memory locations not known to be in Cache_i, and only keep the I/O annotation for counting purposes.

The transformation proceeds in a top-down fashion. We first establish the invariants at the root by adding a RAM-decision-tree which will sort the contents of cache and the $(N-M)/B_2$ blocks of size B_2 not in cache, i.e. sort the memory segments $A[0 \ldots (M-1)]$, $A[M \ldots (M+B_2-1)]$, $A[(M+B_2) \ldots (M+2B_2-1)]$,..., $A[(N-B_2) \ldots (N-1)]$ individually. This can be done by a RAM-decision-tree of height $M \log M + B_2 \log B_2 (N-M)/B_2 \leq N \log M$, using e.g. Mergesort as the sorting algorithm. At each leaf of this tree, we attach a copy of T. The resulting tree could now be heavily pruned for subtrees which are not reachable by any input, but for simplicity, we postpone all such pruning until the entire transformation has been done.

We now continue the top-down transformation of the edges which are copies of edges in T. For each such edge $e = (u, v)$, with u being the parent of v, we build a RAM-decision-tree T_e, insert it at e's position, and insert a copy of the subtree rooted at v at each leaf of T_e.

To build T_e, we first look at the at most two blocks in Cache_2 which was put into Cache_2 by an I/O on the edge. For such a block b, the set of indices $b \cap W$, i.e. the part of b belonging to the working set, cannot have gained new members (but can have lost some) since b last time was in Cache_2. This is because W can only gain members due to I/Os from $\mathcal{P}_1$, and by the normalization step, such an I/O is immediately followed by an access to a member of the block b' put into Cache_1. Since B_1 divides B_2, b' is contained in b, and hence b would have been put into Cache_2 at the same time. Let K be the set of indices in b which has been removed from W since the last time b was in Cache_2, and let L be the set of indices in b which was not in W last time b was in Cache_2. By Invariants 1 and 2, the total order of K and of L is known, so reestablishing Invariant 2 is equivalent to merging two sorted lists of length $|K|$ and $|L|$. The top of T_e will be an optimal RAM-decision-tree T_b reestablishing

Invariant 2. As $|K| + |L| \leq B_2$, this tree has height at most $4|K| + |K| \log \frac{B_2}{|K|}$ [20]. If there is a second such block, we repeat the procedure, inserting another such tree under each leaf of the current T_e.

At the edge e, we now look at the at most two blocks in Cache_1 which was put into Cache_1 by an I/O on e. Each of these induces the addition of at most B_1 new indices in the working set. To maintain Invariant 1, we will for each new index i resolve its order among the elements in the working set. This is done by substituting each leaf of the current T_e by a RAM-decision-tree τ_i resolving this, and repeating the substitution for each of the new indices. The RAM-decision-tree τ_i is chosen to have optimal height among all RAM-decision-tree resolving this *given* the partial order induced by the comparisons on the path from the root of T down to e and further on to the bottom of the current T_e (including previously added τ_i in the current T_e).

The *fourth* and final transformation is just to convert the tree into a standard decision-tree by discarding all annotation and all unary nodes, by pruning subtrees which are not reachable by any input, and by converting references to memory locations into references to input elements. Clearly, the resulting decision-tree is a sorting algorithm. Among the binary nodes pruned are the copies of original binary nodes in T. This is because all comparisons are between two elements of the working set (since $\text{Cache}_1 \subseteq W$ at all times), and by Invariant 1, we had resolved the comparison before we reached the binary node.

Thus, the height of the tree is bounded by the sum of the heights of the trees we insert during transformation three. The first tree inserted had height at most $N \log M$. Any root-to-leaf path in T contained at most t_2 I/Os from $\mathcal{P}_2$. Along any such path, at most $t_1 B_1$ elements are added to W. At the root we have $|W| = M$, and at a leaf we have $|W| \geq M$ (since Cache_1 contains M/B_1 blocks at all times), so at most $t_1 B_1$ can leave W along such a path. To bound the second types of trees inserted, let k_i be the value $|K|$ for the ith I/O from $\mathcal{P}_2$ along such a path. The sum of these trees along any root-to-leaf path is then bounded by $\sum_{i=1}^{t_2} (4k_i + k_i \log \frac{B_2}{k_i})$. By the convexity of the logarithm and $\sum_{i=1}^{t_2} k_i \leq t_1 B_1$, this sum is maximized for $k_i = t_1 B_1 / t_2$, and is therefore bounded by $4t_1 B_1 + t_1 B_1 \log \frac{B_2 t_2}{B_1 t_1}$.

To bound the third type of trees inserted, we along each root-to-leaf path introduce *epochs*, defined by starting a new epoch at the first node in T along the path for which $m_2 = M/B_2$ I/Os from $\mathcal{P}_2$ have taken place since the start of the current epoch. Due to normalization and the assumption that B_1 divides B_2, we know that when new members are added to W, the blocks of size B_2 containing the new member must be in Cache_2. Hence, if $|\text{Cache}_2 \cap W| = s$ at the beginning of an epoch, then at most $(M - s) + B_2 m_2 \leq 2M$ elements can be added to W during an epoch. At all times we have $W \subseteq \text{Cache}_1 \cup \text{Cache}_2$, so taking into account the contents of Cache_1 at the beginning of the epoch, we see that the union of the working sets at the nodes inside one epoch on a root-to-leaf path is bounded in size by $3M$.

We will now bound the heights of the τ_i trees (for the insertion of i into W) along a root-to-leaf path. As said, the size B_2 block b_i containing i must be present in Cache_2 when i is inserted into W. We will attribute this insertion to the I/O which inserted b_i into Cache_2. For a given block b whose lifetime in Cache_2 span r epochs, let k_j^b be the number of i's attributed to b in the j'th of these epochs. We have at most t_2 k_1^b values along a root-to-leaf. For $j \geq 2$, we associate each k_j^b with the epoch it relates to, and note that each epoch can have at most m_2 such values associated. As there are at most t_2/m_2 epochs along a root-to-leaf, the number of i's along a root-to-leaf can be expressed as the sum of at most $t_2 + m_2(t_2/m_2) = 2t_2$ numbers. This sum, which is the number of elements inserted into W

along the path, is bounded by t_1B_1. Each of the i's counted by a k_j^b value comes from a sorted set (by Invariant 2), but the choice from this set is done in an *online* fashion, and hence the introduction of these k_j^b elements into the working set cannot be viewed just as merging of two sorted lists. In Section 4.2, we give an algorithm based on exponential searches which can be used to bound the heights of the τ_i trees. We use the bound stated in Lemma 2, with the set S being the union of the working sets at the nodes inside the epoch on the root-to-leaf path (although we do not have the total order of this set, we have the total order of the actual working set inserted into, and the proof of Lemma 2 still gives a bound on the comparisons done during the exponential searches in our setting). Using $|S| + k_j^b \leq 4M$, we by Lemma 2 get the bound $\sum_{j=1}^{2t_2}(2k_j \log \frac{4M}{k_j} + 4k_j)$ on the combined height of the τ_i trees along a root-to-leaf, for values $k_1, k_2, \ldots, k_{2t_2}$ fulfilling $\sum_{j=1}^{2t_2} k_j \leq t_1B_1$. By the convexity of the logarithm, this sum is bounded by $4t_1B_1 + 2t_1B_1 \log \frac{8Mt_2}{B_1t_1}$.

Adding the bound for the three types of trees inserted during the third transformation, we get that no root-to-leaf path in the final tree is longer than

$$N \log M + 8t_1B_1 + 3t_1B_1 \log \frac{8Mt_2}{B_1t_1} .$$

Since the final tree is a decision-tree for a sorting algorithm, at least one leaf must have depth $N \log N - 1.45N$, by the standard comparison lower bound. This concludes the proof of Theorem 1.

4.2 Exponential searches

In this section we consider the problem of inserting k elements into a sorted set S containing n elements $x_1 < \cdots < x_n$, where $k \leq n$. For simplicity, we assume all keys are distinct. Hwang and Lin [20] described how to optimally merge two sorted lists of length k and n with $O(k \log \frac{k+n}{k})$ comparisons and similar time bounds for merging two AVL-trees and two level-linked (2,4)-trees were achieved in [17, 19].

Here, we consider a variant of the problem where the k elements $y_1, \ldots, y_k$ are given *online* (not necessarily in sorted order). When inserting the ith element y_i, the elements $y_1, \ldots, y_{i-1}$ must already have been inserted. The algorithm gets the order of $y_1, \ldots, y_i$ for free; in particular the algorithm can assume to know the currently closest inserted predecessor $y_i' = \max\{y_j \mid 1 \leq j < i \ \wedge\ y_j < y_i\} \cup \{-\infty\}$ and successor $y_i'' = \min\{y_j \mid 1 \leq j < i \ \wedge\ y_j > y_i\} \cup \{\infty\}$. In the following, we consider how exponential searches achieves similar comparison bounds for the online problem as for the (offline) version considered by Hwang and Lin. The algorithm is to repeatedly apply exponential searches as described in Lemma 1 such that the insertion of y_i is restricted to $[y_i', y_i''] \cap S$.

We first consider a single insertion of y into S. If y partitions S into two sets S_1 and S_2, where all elements in S_1 are smaller than y and all elements in S_2 are larger than y, then an exponential search uses the following number of comparisons.

Lemma 1 *An exponential search splitting a set S into two non-empty sets S_1 and S_2 can be done with*

$$2 \log \min\{|S_1|, |S_2|\} + 2$$

comparisons, provided $\min\{|S_1|, |S_2|\} \geq 1$. *Otherwise at most 2 comparisons are performed.*

Proof. First compare y with $x_{\lceil n/2 \rceil}$ to decide if $|S_1| \leq n/2$. If $|S_1| \leq n/2$, then start an exponential search at x_1; otherwise, start a symmetric exponential search at x_n. Assume without loss of generality $|S_1| \leq n/2$. Compare y with $x_1, x_2, \ldots, x_{2^j}, \ldots$ for increasing j until $y < x_{2^j}$. If $|S_1| = 0$ then in total two comparisons have been performed and we are done. Otherwise $2^i \leq |S_1| < 2^{i+1}$ and $i+1$ comparisons have been performed to find i. Finally, a binary search is performed among the $2^i - 1$ elements $x_{2^i}, \ldots, x_{2^{i+1}-1}$. Since the binary search requires i comparisons, the exponential search in total requires $1 + (i+1) + i = 2(i+1) = 2\lfloor \log |S_1| \rfloor + 2$ comparisons. □

Lemma 2 *The online insertion of a permutation of k elements into a sorted set S, where $k \leq |S|$, can be done with $2k \log \frac{|S|+k}{k} + 4k$ comparisons.*

Proof. We repeatedly apply exponential searches as described in Lemma 1 such that the insertion of y_i is restricted to $[y_i', y_i''] \cap S$.

We view the sequence of insertions as creating a partition of S, where the insertion of y_i partitions $S \cap [y_i', y_i'']$ into $S \cap [y_i', y_i]$ and $S \cap]y_i, y_i'']$. The created partitions form a *partition tree*, which is a binary tree where each node v is labeled with a subset S_v of S as follows: The root represents the whole set S, the leaves the final partition, and each internal node the insertion of an y_i such that the node represents the set $S \cap [y_i', y_i'']$, and the two children the sets $S \cap [y_i', y_i]$ and $S \cap]y_i, y_i'']$.

We now analyze the number of comparisons for the k insertions bottom-up on the partition tree. The cost of a leaf is zero. For an internal node v with children with sets S_1 and S_2, where $S_v = S_1 \cup S_2$, we by Lemma 1 has an associated cost of $2 \log \min\{|S_1|, |S_2|\} + 2$ comparisons. Assuming that each insertion generates two non-empty subsets, we now prove by induction that the total number of comparisons in the subtree rooted at a node v is at most

$$2 \sum_{w \in L_v} \log |S_w| - 2 \log |S_v| - 4 + 4|L_v| , \tag{1}$$

where L_v denotes the leaves below v (if v is leaf then $L_v = \{v\}$). If v is a leaf, the sum (1) is zero since v is the only leaf. For an internal v with children v_1 and v_2, we from the induction hypothesis get that the total number of comparisons is at most

$$\begin{aligned} & 2 \sum_{w \in L_{v_1}} \log |S_w| - 2 \log |S_{v_1}| - 4 + 4|L_{v_1}| \\ & + 2 \sum_{w \in L_{v_2}} \log |S_w| - 2 \log |S_{v_2}| - 4 + 4|L_{v_2}| \\ & \quad + 2 \log \min\{|S_{v_1}|, |S_{v_2}|\} + 2 \\ \leq\ & 2 \sum_{w \in L_v} \log |S_w| - 2 \log |S_v| - 4 + 4|L_v| , \end{aligned}$$

since we have that $\log |S_v| \leq \log(2 \max\{|S_{v_1}|, |S_{v_2}|\}) = 1 + \log \max\{|S_{v_1}|, |S_{v_2}|\}$ and $|L_v| = |L_{v_1}| + |L_{v_2}|$. By letting $\log |S_v|$ cancel the contribution of one leaf, and exploiting the convexity of the log function we get that (1) is bounded by $2(|L_v| - 1) \log \frac{|S_v|}{|L_v|-1} - 4 + 4|L_v|$. Since the tree has $k+1$ leaves, the total number of comparisons is at most $2k \log \frac{|S|}{k} + 4k$, assuming all insertions induce non-empty partitions. If k_1 insertions induce empty partitions and $k_2 = k - k_1$ insertions induce non-empty partitions, then the total number of comparisons is $2k_1 + 4k_2 + 2k_2 \log \frac{|S|}{k_2} \leq 4k + 2k \log \frac{|S|+k}{k}$. □

4.3 Permutation

In this section we consider lower bound trade-offs for cache-oblivious algorithms for permuting. The computational problem is the following: Given an array of N elements and a permutation, rearrange the elements in the array according to the input permutation. We assume that the algorithm has complete knowledge about the permutation, implying that the only computational task is to move the input elements to the given destination cells.

In the RAM model, the problem is trivially solved using $O(N)$ element moves. In the I/O model, Aggarwal and Vitter [2] showed that the I/O complexity of the problem is $\Theta(\min\{N, \mathrm{Sort}(N)\})$, i.e. an optimal external memory permuting algorithm is to move one element per I/O or applying an optimal sorting algorithm, depending on what algorithm performs the fewest I/Os. In this section, we prove that a similar I/O bound cannot be achieved in the cache-oblivious model—not even in the presence of a tall cache assumption.

Our proof uses ideas from the permutation lower bound proof of Aggarwal and Vitter [2]: The lower bound is achieved by counting how many permutations can at most be achieved by the I/Os performed so far. In contrast to [2], our analysis is performed with respect to two different block sizes simultaneously.

Let A be a cache-oblivious algorithm for permuting. Let t_1 and t_2 denote an upper bound on the number of I/Os done by two optimal offline I/O strategies for respectively block sizes B_1 and B_2, memory size M, and on any input permutation of N elements. We assume that $B_1 \leq B_2$ and that B_1 divides B_2. The following lemma states our main lower bound trade-off between t_1 and t_2.

Lemma 3 *If $k = t_1 B_1 / t_2$, then*

$$(B_2!)^{N/B_2}\left((N/B_2 + t_2)\binom{B_2}{k}\binom{2M - B_2 + k}{k} + (M/B_2)\binom{B_2}{k}\right)^{2t_2} \geq N!\,.$$

Proof. In the following, we for each input permutation transform A into a new permuting algorithm A' together with an explicit sequence of I/Os for block size B_2 and memory size $2M$. We will use the properties of these A' algorithms to derive a lower bound trade-off for t_1 and t_2.

We first make two simplifying assumptions about A and the optimal I/O strategies performed for block sizes B_1 and B_2. Similar to Aggarwal and Vitter [2], we can assume that there exists at most one copy of each element at any time: Whenever an element is copied to another cell, the old cell is assigned the nil value. To see this assume that the algorithm maintains several copies of the same element. Since only one copy of an element is part of the output, we can simply cancel all copying of elements which are not part of the output.

Secondly, for any block size there exists an optimal offline I/O strategy where each block read is immediately followed by an access to the block read. This follows by delaying the I/Os of any optimal offline I/O strategy, such that a block is read by an I/O just prior to the first access to any element of the block. When analyzing the I/O sequences for respectively block size B_1 and B_2 simultaneously, we can therefore assume that when a block b_1 of size B_1 is read into cache in, then the block b_2 of size B_2 containing b_1 will already have been read into cache when using block size B_2.

For the algorithm A', the cache will consist of two areas of size M: The first M cells contain the "large" blocks of size B_2 currently swapped in by A if using block size B_2, and the second M cells contain the content of the memory of A when using the "small" block size B_1. The second area will be simulated in M/B_2 otherwise not used blocks by A'. Below we give the details and describe how to ensure that A' only maintains one copy of each element.

Whenever a "large" block is read or written by A (when using the offline I/O strategy for block size B_2), we let A' read or write the same block. Whenever A reads a "small" block s, A' moves the contents of s from the "large" block area to the "small" block area, exploiting that we can assume that A will have the "large" block in internal memory as discussed above. Finally, we consider the case where A is evicting a "small" block s. If the "large" block ℓ containing s is in the "large" area, A' moves the content of s from the "small" area to the "large" area. The final case where ℓ is not in the "large" area can be avoided by moving the eviction of s to just prior to the eviction of ℓ. This is possible since by definition, A can only access cells that are read into cache; A cannot do any access to ℓ, and therefore also not to s, in the period between the eviction of ℓ and the eviction of s.

We will guarantee that for a "large" block read by A', there will at most be moved k elements between the "large" area and the "small" area before the "large" block is evicted from cache. This can be guaranteed by evicting a "large" block from cache and immediately reloading it again whenever k cells have been moved from or to the "large" block. This at most increases the number of I/Os for A' from t_2 to $t_2 + t_1 B_1/k = 2t_2$ I/Os. Finally, we without loss of generality can assume that A' only accesses the first $N/B_2 + t_2$ memory blocks, since A' at most loads t_2 blocks from memory.

We will now argue that the number of permutations A' can generate with t I/Os is bounded by

$$(B_2!)^{N/B_2}\left((N/B_2+t_2)\binom{B_2}{k}\binom{2M-B_2+k}{k} + (M/B_2)\binom{B_2}{k}\right)^t . \tag{2}$$

For a given state of A', we define the *working set* to be the cells of the "small" area together with the subblocks of size B_1 of blocks in the "large" area that the algorithm accesses before the "large" block is evicted again. Since A' knows the at most k elements of a "large" block it will access during the block is in cache, we can assume that when A' loads a block, A' provides the at most k cells to be included in the working set.

For each $t \geq 0$, we will consider a superset S_t of all possible pairs of $\langle$permutation,working set$\rangle$ that can be generated with t I/Os (the same permutation can appear several times with distinct working sets). We require that $S_0, \ldots, S_{2t_2}$ satisfy that if a permutation is included together with a given working set, then all permutations which can be reached by permuting the content internally in the working set and permuting the content internally in a block of size B_2, excluded the working set, is also in the superset. The consequence of this assumption is that rearranging elements in the working set, either inside the "small" area or by moving blocks of size B_1 between the "small" and the "large" area, cannot introduce new permutations.

In the initial state, we assume that the cache of A' is empty, but that all possible internal permutations of elements in the N/B_2 blocks in memory are contained in S_0, i.e. $|S_0| =$

$(B_2!)^{N/B_2}$.

If the tth I/O is a block read, then there are at most $N/B_2 + t_2$ distinct blocks to read. There are $\binom{B_2}{k}$ distinct ways to select k cells from the block read to add to the working set. For a $\langle$permutation,working set$\rangle$ in S_{t-1}, all possible permutations of the existing working set and the k elements are already contained in S_{t-1}. The only new permutations to add to S_t are the possible ways to interleave the k new elements with the existing working set of size at most $2M - B_2$, which is $\binom{2M-B_2+k}{k}$. It follows that a block read can at most increase the number of pairs $\langle$permutation,working set$\rangle$ by a factor

$$(N/B_2 + t_2)\binom{B_2}{k}\binom{2M - B_2 + k}{k} .$$

Evicting one of the M/B_2 blocks to memory removes at most k cells from the working set which are contained in the block. Since all permutations of the k elements and of the remaining $B_2 - k$ elements in the block are already contained in S_{t-1}, the only new permutations to add to S_t are all the possible ways to interleave the k and $B_2 - k$ elements with the existing working set, which is at most $\binom{B_2}{k}$. It follows that a block write can at most increase the number of pairs $\langle$permutation,working set$\rangle$ by a factor

$$(M/B_2)\binom{B_2}{k} .$$

It follows that t I/Os can at most increase the number of permutations by a factor

$$(N/B_2 + t_2)\binom{B_2}{k}\binom{2M - B_2 + k}{k} + (M/B_2)\binom{B_2}{k} .$$

In total $|S_t|$, and therefore the number of possible permutations generated by A' using t I/Os, is given by (2). Since A' can generate all $N!$ possible permutations using $2t_2$ I/Os, the lemma follows. $\square$

We note that the trade-off in Lemma 3 also holds in the average case where each permutation is equally likely. Assume t_1 and t_2 is the average number of I/Os for block size B_1 and B_2 over a uniform distribution of the input permutations. Then for at least $\frac{3}{4}N!$ permutations there are less than $4t_1$ I/Os for block size B_1, and similarly for $\frac{3}{4}N!$ permutations there are less than $4t_2$ I/Os for block size B_2. We have that for at least $\frac{1}{2}N!$ permutations there are at most $4t_1$ and $4t_2$ I/Os, respectively for block size B_1 and B_2. For the average case we get the following.

Lemma 4 *If $k = t_1B_1/t_2$, then for the average case*

$$(B_2!)^{N/B_2}\left((N/B_2 + 4t_2)\binom{B_2}{k}\binom{2M - B_2 + k}{k}\right.$$

$$\left. + (M/B_2)\binom{B_2}{k}\right)^{8t_2} \geq \frac{1}{2}N! .$$

We now give the proof of Theorem 2. *Proof.*[of Theorem 2] For the sake of contradiction, assume that there exists a $\delta > 0$ and a cache-oblivious algorithm that performs $O(\text{Perm}_{M,B}(N))$ for all $M \geq 2B$ and $1 \leq B \leq M^{\delta}$.

Taking the logarithm on both sides of the equation of Lemma 4, and simplifying using $\frac{M}{B_2} \leq \frac{N}{B_2} \leq t_2$, we get

$$c_1(t_2 \log t_2 + N \log M + t_1 B_1 \log M) \geq N \log N \ ,$$

for some constant c_1. Letting $B_1 = O(1)$ implies $t_1 = O(N)$, and the above inequality can be reduced to

$$c_2(t_2 \log t_2 + N \log M) \geq N \log N \ ,$$

for some constant c_2. Letting $N = M^{2c_2}$ we get

$$t_2 \log t_2 \geq \frac{1}{2c_2} N \log N \ ,$$

implying that $t_2 \geq \frac{1}{2c_2} M^{2c_2}$.

The contradiction follows by letting $B_2 = M^{\delta}$, since $\text{Sort}_{M,M^{\delta}}(M^{2c_2}) = \frac{2c_2-1}{1-\delta} M^{2c_2-\delta}$ is asymptotic smaller than $\frac{1}{2c_2} M^{2c_2} \leq t_2$ for increasing M. □

References

[1] P. Agarwal, L. Arge, A. Danner, and B. Holland-Minkley. On cache-oblivious multidimensional range searching. In *Proc. 19th ACM Symposium on Computational Geometry*, 2003. To appear.

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

[3] L. Arge. External memory data structures. In *Proc. 9th Annual European Symposium on Algorithms (ESA)*, volume 2161 of *LNCS*, pages 1–29. Springer, 2001.

[4] 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*, pages 268–276. ACM Press, 2002.

[5] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In F. K. H. A. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides, editors, *Algorithms and Data Structures, Third Workshop*, volume 709 of *LNCS*, pages 83–94, Montréal, Canada, 11–13 Aug. 1993. Springer.

[6] S. Baase and A. V. Gelder. *Computer Algorithms, Introduction to Design and Analysis.* Addison-Wesley, 3rd edition, 1999.

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

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

[9] 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. Springer, 2002.

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

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

[12] 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*, pages 29–39, 2002.

[13] G. Bilardi and E. Peserico. A characterization of temporal locality and its portability across memory hierarchies. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, volume 2076, pages 128–139. Springer, 2001.

[14] 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. Springer, 2002.

[15] G. S. Brodal and R. Fagerberg. Funnel heap - a cache oblivious priority queue. In *Proc. 13th Annual International Symposium on Algorithms and Computation*, volume 2518 of *LNCS*, pages 219–228. Springer, 2002.

[16] 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*, pages 39–48, 2002.

[17] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979.

[18] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297. IEEE Computer Society Press, 1999.

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

[20] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal of Computing*, 1(1):31–39, 1972.

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

[22] 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. Springer, 2001.

[23] D. D. Sleator and R. E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28:202–208, 1985.

[24] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.