Fault Tolerant External Memory Algorithms

Gerth Stølting Brodal[1], Allan Grønlund Jørgensen[1,⋆], and Thomas Mølhave[1,⋆]

BRICS, MADALGO⋆⋆, Department of Computer Science, Aarhus University, Denmark. {gerth,jallan,thomasm}@madalgo.au.dk

Abstract. Algorithms dealing with massive data sets are usually designed for I/O-efficiency, often captured by the I/O model by Aggarwal and Vitter. Another aspect of dealing with massive data is how to deal with memory faults, *e.g.* captured by the adversary based faulty memory RAM by Finocchi and Italiano. However, current fault tolerant algorithms do not scale beyond the internal memory. In this paper we investigate for the first time the connection between I/O-efficiency in the I/O model and fault tolerance in the faulty memory RAM, and we assume that both memory and disk are unreliable. We show a lower bound on the number of I/Os required for any deterministic dictionary that is resilient to memory faults. We design a static and a dynamic deterministic dictionary with optimal query performance as well as an optimal sorting algorithm and an optimal priority queue. Finally, we consider scenarios where only cells in memory or only cells on disk are corruptible and separate randomized and deterministic dictionaries in the latter.

1 Introduction

In this paper we conduct the first study of algorithms and data structures for external memory in the presence of an unreliable internal and external memory.

Contemporary memory devices such as SRAM and DRAM [1, 2] can be unreliable due to a number of factors, such as power failures, radiation, and cosmic rays. The content of a cell in unreliable memory can be silently altered and in standard memory circuits there is no direct way of detecting these types of corruptions.

Corrupted content in memory cells can greatly affect many standard algorithms. For instance, in a typical binary search in a sorted array, a single corruption encountered in the early stages of the search can cause the search path to end $\Omega(N)$ locations away from its correct position. Replication of data can help in dealing with corruptions, but is not always feasible, since the time and space overheads of storing and fetching replicated values can be significant. Memory

⋆ Supported in part by an Ole Roemer Scholarship from the Danish National Science Research Council.

⋆⋆ Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

corruptions have been addressed in various ways, both at the hardware and software level. At the software level, soft memory errors are dealt with using several different low-level techniques [3–6]. However, most of these handle instruction corruptions rather than data corruptions. Corruptions can also often be discovered by existing hardware techniques, but even these techniques can fail and let some corrupted data take part of computations.

Finocchi and Italiano [7] introduced the *faulty-memory random access machine*, based on the traditional RAM model. In this model, memory corruptions can occur at any time and at any place in memory during the execution of an algorithm, and corrupted memory cells cannot be distinguished from uncorrupted cells. In the faulty-memory RAM, it is assumed that there is an adaptive adversary, that chooses how, where, and when corruptions occur. The model is parametrized by an upper bound, δ, on the number of corruptions the adversary can perform during the lifetime of an algorithm, and $\alpha \leq \delta$ denotes the actual number of corruptions that takes place. Motivated by the fact that registers in the processor are considered incorruptible, $O(1)$ safe memory locations are provided. Moreover, it is assumed that reading a word from memory is an atomic operation. In randomized computation, as defined in [7], the adversary does not see the random bits used by an algorithm. An algorithm is *resilient* if it works correctly on the set of uncorrupted cells in the input. For instance, a resilient sorting algorithm outputs all uncorrupted elements in sorted order while corrupted elements can appear at arbitrary positions in the output. A resilient searching algorithm must return yes if there is an uncorrupted element matching the search key.

Memory corruptions are of particular concern for applications dealing with massive amounts of data since such applications typically run for a very long time, and are thus more likely to encounter memory cells containing corrupted data. However, algorithms designed in the RAM model assume that an infinite amount of memory cells are available. This is not true for typical computers where internal memory is limited and elements are transferred between the memory and a much larger, but dramatically slower, hard drive in large consecutive blocks. This means that it is important to design algorithms with a high degree of locality in their memory access pattern, that is, algorithms where data accessed close in time is also stored close in memory. This situation is modeled in the *I/O model* of computation [8]. In this model a disk of unlimited size and a memory of size M are available. Elements are transferred between disk and memory in blocks of size B and computation is performed on elements in memory only. The complexity measure is the number of block transfers (*I/O*s) performed.

Previous Work: Several problems have been addressed in the faulty-memory RAM, see a very recent survey [9] for more information. In [10, 7], matching upper and lower bounds for resilient sorting and randomized searching were given. Sorting N elements requires $\Theta(N \log N + \alpha\delta)$ time [7]. Searching in a sorted array requires $\Omega(\log N + \delta)$ time, and an optimal deterministic algorithm matching that bound is described in [11]. It has been empirically shown that resilient algorithms are of practical interest [12]. Recently, in [11, 13, 14] resilient data structures

	I/O Complexity	Assumptions	I/O Tolerance (max δ)	Time Tolerance (max δ)
Det. Dict.	$O\left(\frac{1}{\varepsilon}\log_B N + \frac{\delta}{B^{1-\varepsilon}}\right)$	$\frac{1}{\log B} < \varepsilon < 1$	$O(B^{1-\varepsilon}\log_B N)$	$O(\log N)$
Ran. Dict.	$O(\log_B N + \frac{\delta}{B})$	Memory Safe	$O(B\log_B N)$	$O(\log N)$
P. Queue	$O(\frac{1}{1-\varepsilon}\frac{1}{B}\log_{M/B}(N/M))$	$\delta \leq M^{\varepsilon}$, $\varepsilon < 1$	$O(M^{\varepsilon})$	$O(\log N)$
Sorting	$O(\frac{1}{1-\varepsilon}\mathrm{Sort}(N))$	$\delta \leq M^{\varepsilon}$, $\varepsilon < 1$	$O(M^{\varepsilon})$	$O(\sqrt{N\log N})$

Table 1. The first column shows the I/O upper bounds presented in our paper with the assumptions shown in the second column. The third and fourth column shows how many corruptions the algorithms can tolerate while still matching the optimal algorithms in the I/O and comparison model respectively. Note that the restriction imposed by the time bounds are orders of magnitude stronger than the ones imposed by the I/O bounds for realistic values of M, B and N.

were proposed, in particular a resilient dynamic dictionary supporting searches in optimal $\Theta(\log N + \delta)$ time with an amortized update cost of $O(\log N + \delta)$ was presented in [11], and a priority queue supporting operations in $O(\log N + \delta)$ time was presented in [14]

For the I/O model, a comprehensive list of results have been achieved. It is shown in [8] that sorting N elements requires $\mathrm{Sort}(N) = \Theta(N/B\log_{M/B}(N/B))$ I/Os. See recent surveys [15, 16] for an overview of other results. In the I/O model, a comparison based dictionary with optimal queries can be achieved with a B-tree [17], which supports queries and updates in $O(\log_B N)$ I/Os.

Current resilient algorithms do not scale past the internal memory of a computer and thus, it is currently not possible to work with large sets of data I/O-efficiently while maintaining resiliency to memory corruptions. Since both models become increasingly interesting as the amount of data increases, it is natural to consider whether it is possible to achieve resilient algorithms that use the disk optimally. Very recently, this was also proposed as an interesting direction of research by Finocchi *et al.* [9, 10].

Our Contribution: The work in this paper combines the faulty memory RAM and the external memory model in the natural way. The model has three levels of memory: a disk, an internal memory of size M, and $O(1)$ CPU registers. All computation takes place on elements placed in the registers. The content of any cell on disk or in internal memory can be corrupted at any time, but at most δ corruptions can occur. Moving elements between memory and registers takes constant time and transferring a chunk of B consecutive elements between disk and memory costs one *I/O*. Transfers between the different levels are atomic, no data can be corrupted while it is being copied. Correctness of an algorithm is proved with the assumption that an adaptive adversary may perform corruptions during execution. For randomized algorithms we assume that the random bits are hidden from the adversary. In two natural variants of our model it is assumed that corruptions take place only on disk, or only in memory.

In this paper, we present I/O-efficient solutions to all problems that, to the best of our knowledge, have previously been considered in the faulty memory

RAM. It is not clear that resilient algorithms can be optimal both in time and in I/O-complexity. Most techniques for designing I/O-efficient algorithms naturally try to arrange data on disk such that few blocks need to be read in order to extract the information needed, whereas resilient algorithms try to put little emphasis on individual, potentially corrupted, memory cells.

It is known that any resilient comparison based search algorithm must examine $\Omega(\log N + \delta)$ memory cells [10]. Combining this with the well-known $\Omega(\log_B N)$ I/O lower bound on external memory comparison based searching, we get a simple lower bound of $\Omega\left(\log_B N + \frac{\delta}{B}\right)$ I/Os, and $\Omega(\log N + \delta)$ time. In Section 2 we prove a stronger lower bound of $\Omega\left(\frac{1}{\varepsilon}\log_B N + \frac{\delta}{B^{1-\varepsilon}}\right)$ I/Os for a search, for all $\log_B N \leq \delta \leq B \log N$ and ε given by the equation $\delta = \frac{B^{1-\varepsilon}}{\varepsilon}\log_B N$. In the case where $\delta = \Theta(\frac{B}{\log B}\log_{\log B} N)$, setting $\varepsilon = \frac{\log\log B}{\log B}$ gives a lower bound of $\Omega(\log_{\log B} N + \frac{\delta}{B}\log B)$ which is $\omega(\log_B N + \frac{\delta}{B})$. We come to the interesting conclusion that no deterministic resilient dictionary can obtain an I/O bound of $O(\log_B N + \frac{\delta}{B})$ without some assumptions on δ. The lower bound is valid for randomized algorithms as long as the internal memory is unreliable. For deterministic algorithms, the lower bound also holds if the internal memory is reliable and corruptions only occur on disk.

In Section 3 we construct a resilient dictionary supporting searches using expected $O\left(\log_B N + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time for any δ if corruptions occur exclusively on disk. Thus, we have an interesting separation between the I/O complexity of resilient randomized and resilient deterministic searching algorithms. This also proves that it is important whether it is the disk or the internal memory that is unreliable.

In Section 4 we present an optimal resilient static dictionary supporting queries in $O\left(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time when $\log N \leq \delta \leq B \log N$ and $\frac{1}{\log B} \leq c \leq 1$. Queries use $O(\log_B N + \frac{\delta}{B})$ I/Os and $O(\log n + \delta)$ time for $\delta \leq \log N$ and $\delta > B \log N$. Additionally, we construct randomized and deterministic dynamic dictionaries with optimal query bounds using our static dictionaries.

Finally, in Section 5 we describe a resilient multi-way merging algorithm. We use this algorithm to design an optimal resilient sorting algorithm using $O(\frac{1}{1-\varepsilon}\text{Sort}(N))$ I/Os and $O(N \log N + \alpha\delta)$ time under the assumption that $\delta \leq M^{\varepsilon}$, for $0 \leq \varepsilon < 1$. The multi-way merging algorithm is also used to design a resilient priority queue for the case $\delta \leq M^{\varepsilon}$, where $0 \leq \varepsilon < 1$. Our priority queue supports *insert* and *delete-min* in optimal $O(\frac{1}{1-\varepsilon}(1/B)\log_{M/B}(N/M))$ I/Os amortized, matching the bounds for non-resilient external memory priority queues. The amortized time bound for both operations is $O(\log N + \delta)$ matching the time bounds of the optimal resilient priority queue of [14].

Table 1 shows an overview of the upper bounds in this paper. The two last columns in the table shows how many corruptions our algorithms can tolerate while still achieving optimal bounds in the I/O model and comparison model respectively. Note that the bounds on δ required to get optimal time are orders of magnitude smaller than the bounds required to get optimal I/O performance for realistic values of N, M and B. We conclude that it is possible, under realistic

assumptions, to get resilient algorithms that are optimal in both the I/O-model and the comparison model without restricting δ more than what was required to obtain optimal time bounds in the faulty memory RAM.

Preliminaries: Throughout the paper we use the notion of a *reliable value*, which is a value stored in unreliable memory that can be retrieved reliably in spite of possible corruptions. This is achieved by replicating the given value in $2\delta + 1$ consecutive cells. Since at most δ of the copies can be corrupted, the majority of the $2\delta + 1$ elements are uncorrupted. The value can be retrieved using $O(\frac{\delta}{B})$ I/Os and $O(\delta)$ time with the majority algorithm in [18], which scans the $2\dot{+}1$ values keeping a single majority candidate and a counter in reliable memory. A sequence is *faithfully ordered* if the uncorrupted elements form a sorted subsequence.

2 Lower Bound for Dictionaries

Any resilient searching algorithm must examine $\Omega(\log N\dot{+})$ memory cells in the comparison model [10]. The $\Omega(\log N)$ term follows from the comparison model lower bound for searching. It is well-known that comparison based searching in the I/O model requires expected $\Omega(\log_B N)$ I/Os. Since any resilient searching algorithm must read at least $\Omega(\delta)$ elements to ensure at least some non-corrupted information is the basis for the output, we get the following trivial lower bound.

Lemma 1. *For any comparison based randomized resilient dictionary the average-case expected search cost is* $\Omega\left(\log_B N + \frac{\delta}{B}\right)$ *I/Os.*

In this section we prove a stronger lower bound on the worst-case number of I/Os required for any deterministic resilient static dictionary in the comparison model. We do not make any assumptions on the data structure used by the dictionary, nor on the space it uses. Additionally, we do not bound the amount of computation time used in a query and we assume that the total order of all elements stored in the dictionary are known by the algorithm initially. During the search for an element e, an algorithm gains information by performing block I/Os, each I/O reading B elements from disk. Before a block of B elements is read into memory the adversary can corrupt the elements in the block. The adversary is allowed to corrupt up to δ elements during the query operation, but does not have to reveal when it chooses to do so. Also, the adversary adaptively decides what the rank of the search element has among the N dictionary elements. Of course, the rank must be consistent with the previous uncorrupted elements read by the algorithm.

Theorem 1. *Given N and δ, any deterministic resilient static dictionary requires worst-case* $\Omega\left(\frac{1}{\varepsilon}\log_B N\right)$ *I/Os for a search, for all ε where* $\frac{1}{\log B} \leq \varepsilon \leq 1$ *and* $\delta \geq \frac{1}{\varepsilon}B^{1-\varepsilon}\log_B N$.

Proof. We design an adversary that uses corruptions to control how much information any correct query algorithm gains from each block transfer.

Let ε be a constant such that $\frac{1}{\log B} \leq \varepsilon \leq 1$. The strategy of the adversary is as follows. For each I/O, the adversary narrows the *candidate interval* where e can be contained in by a factor B^ε. Initially, the candidate interval consists of all N elements. For each I/O, the adversary implicitly divides the sorted set of elements in the candidate interval into B^ε slabs of equal size. Since the search algorithm only reads B elements in an I/O, there must be at least one slab containing at most $B^{1-\varepsilon}$ of these elements. The adversary corrupts these elements, such that they do not reveal any information, and decides that the search element resides in this slab. The remaining elements transferred are not corrupted and are automatically consistent with the interval chosen for e. The game is then played recursively on the elements of the selected slab, until all elements in the final candidate interval have been examined.

For each I/O, the candidate interval decreases by a factor B^ε. The algorithm has no information regarding elements in the slab except for the corrupted elements from the I/Os performed so far. After k I/Os the candidate interval has size $\frac{N}{(B^\varepsilon)^k}$ and the adversary has introduced at most $kB^{1-\varepsilon}$ corruptions. The game continues as long as there is at least one uncorrupted element among the elements remaining in the candidate interval, which the adversary can choose as the search element. All corrupted elements may reside in the current candidate interval, and the game ends when the size of the candidate interval, $\frac{N}{(B^\varepsilon)^k}$, becomes smaller than or equal to the total number of introduced corruptions, $kB^{1-\varepsilon}$. It follows that at least $\Omega\left(\log_{B^\varepsilon} \frac{N}{B^{1-\varepsilon}}\right) = \Omega\left(\frac{1}{\varepsilon}\log_B N\right)$ I/Os are required. The adversary introduces at most $B^{1-\varepsilon}$ corruptions in each step. If ε satisfies $\frac{1}{\varepsilon}B^{1-\varepsilon}\log_B N \leq \delta$, then the adversary can play the game for at least $\frac{1}{\varepsilon}\log_B N$ rounds and the theorem follows. □

For deterministic algorithms it does not matter whether elements can be corrupted on disk or in internal memory. Since the adversary is adaptive it knows which block of elements an algorithm will read into internal memory next, and may choose to corrupt the elements on disk just before they are loaded into memory, or corrupt the elements in internal memory just after they have been written there. In randomized algorithms where the adversary does not know the algorithm's random choices it cannot determine which block of elements will be fetched from disk before the transfer has started. Therefore, the adversary can follow the strategy above only if it can corrupt elements in internal memory.

By setting $\delta = \frac{1}{\varepsilon}B^{1-\varepsilon}\log_B N$ in Theorem 1, we get the following corollary.

Corollary 1. *Any deterministic resilient static dictionary requires worst-case* $\Omega(\frac{1}{\varepsilon}\log_B N) = \Omega(\frac{\delta}{B^{1-\varepsilon}})$ *I/Os for a search, where* $\delta \in [\log_B N, B\log N]$*, and* ε *given by* $\delta = \frac{1}{\varepsilon}B^{1-\varepsilon}\log_B N$*.*

The trivial I/O lower bound for a resilient searching algorithm is $\Omega\left(\log_B N + \frac{\delta}{B}\right)$. Setting $\varepsilon = \frac{\log\log B}{\log B}$ in Theorem 1 shows that this is not optimal.

Corollary 2. *For* $\delta = \frac{B}{\log B}\log_{\log B} N$ *any deterministic resilient static dictionary requires worst-case* $\Omega(\frac{\log B}{\log\log B}(\log_B N + \frac{\delta}{B}))$ *I/Os for a search.*

3 Randomized Static Dictionary

In this section we describe a simple I/O-efficient randomized static dictionary, that is resilient to corruptions on the disk. Corruptions in memory are not allowed, thus the adversarial lower bound in Theorem 1 does not apply. The dictionary supports queries using expected $O\left(\log_B N + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time. The algorithm is similar to the randomized binary search algorithm in [10]. Remember that, if only elements on disk can be corrupted, the lower bound from Theorem 1 also holds for deterministic algorithms. This means that deterministic and randomized algorithms are separated by the result in this section.

The idea is to store the N elements in the dictionary in sorted order in an array S and to build 2δ B-trees [17], denoted $T_1, \ldots, T_{2\delta}$, of size $\lfloor \frac{N}{2\delta} \rfloor$. The i'th B-tree T_i stores the $2\delta j + i$'th element in S for $j = 0, \ldots, \lfloor \frac{N}{2\delta} \rfloor - 1$. Each node in each tree is represented by a faithfully ordered array of $\Theta(B)$ search keys. The nodes of the B-tree are laid out in left to right breadth first order, to avoid storing pointers, *i.e.* the c'th child of the node at index k has index $Bk + c - (B-1)$.

The search for an element e proceeds as follows. A random number $r_1 \in \{1, \ldots, 2\delta\}$ is generated, and the root block of T_{r_1} is fetched into the internal memory. In this block, a binary search is performed among the search keys resulting in an index, i, of the child where the search should continue. A new random number $r_2 \in \{1, \ldots, 2\delta\}$ is generated, and the i'th child of the root in tree T_{r_2} is fetched and the algorithm proceeds iteratively as above. The search terminates when a leaf is reached and two keys $S[2\delta j + i]$ and $S[2\delta(j+1) + i]$ have been identified such that $S[2\delta j + i] \leq e < S[2\delta(j+1) + i]$. If the binary search was not mislead by corruptions of elements, then e is located in the subarray $S[2\delta j + i, \ldots, 2\delta(j+1) + i]$. To check whether the search was mislead, the following *verification procedure* is performed. Consider the neighborhoods $L = S[2\delta(j-1) + i - 1, \ldots, 2\delta j + i - 1]$ and $R = S[2\delta(j+1) + i + 1, \ldots, 2\delta(j+2) + i + 1]$, containing the $2\delta+1$ elements in S situated to the left of $S[2\delta j + i]$ and to the right of $S[2\delta(j+1) + i]$ respectively. The number $s_L = |\{z \in L \mid z \leq e\}|$ of elements in L that are smaller than e is computed by scanning L. Similarly, the number s_R of elements in R that are larger than e is computed. If $s_L \geq \delta + 1$ and $s_R \geq \delta + 1$, and the search key is not encountered in L or R, we decide whether it is contained in the dictionary or not by scanning the subarray $S[2\delta j, \ldots, 2\delta(j+1)]$. If s_L or s_R is smaller than $\delta + 1$, at least one corruption has misguided the search. In this case, the search algorithm is restarted.

Theorem 2. *The data structure described is a linear space randomized dictionary supporting searches in expected $O\left(\log_B N + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time assuming that memory cells are incorruptible and block transfers are atomic.*

Details will appear in the full paper. If memory cells were corruptible the atomic transfer assumption would be of little use. The adversary could simply corrupt the elements in the internal memory after the block transfer completes, decreasing the benefit of the randomization.

4 Optimal Deterministic Static Dictionary

In this section we present a linear space deterministic resilient static dictionary. Let c be a constant such that $\frac{1}{\log B} \leq c \leq 1$. The dictionary supports queries in $O\left(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time. In Section 2 we proved a lower bound on the I/O complexity of resilient dictionaries, and by choosing c in the above bound to minimize the expression for $\alpha = \delta$, this bound matches the lower bound. Thus, this dictionary is optimal.

Our data structure is based on the B-tree and the resilient binary search algorithm from [11]. In a standard B-tree search one corrupted element can misguide the algorithm, forcing at least one I/O in the *wrong* part of the tree. To circumvent this problem, each guiding element in each internal node is determined by taking majority of B^{1-c} copies. This gives a trade-off between the number of corruptions required to misguide a search, and the fan-out of the tree, which becomes B^c. Additionally, each node stores $2\delta + 1$ copies of the minimum and maximum element contained in the subtree, such that the search algorithm can reliably check whether it is on the correct path in each step. We ensure that the query algorithm avoids reading the same corrupted element twice by ensuring that any element is read at most once. The exact layout of the tree and the details of the search operation are as follows.

Structure: Let S be the set of elements contained in the dictionary and let N denote the size of S. The dictionary is a B^c-ary search tree T built on $\frac{N}{\delta}$ leaves. The elements of S are distributed to the leaves in faithful order such that each leaf contains δ elements. Each leaf is represented by a *guiding element* which is smaller than the smallest uncorrupted element in the leaf and larger than the largest uncorrupted element in the preceding leaf. The top tree is built using these guiding elements. The tree is stored in a breadth-first left-to-right layout on disk, such that no pointers are required.

Each internal node u in T stores three types of elements; guiding elements, minimum elements, and maximum elements, stored consecutively on disk. The guiding elements are stored in $\lceil (2\delta + 1)/B^{1-c} \rceil$ identical blocks. Each block contains B^{1-c} copies of each of the B^c guiding elements in sorted order such that the first B^{1-c} elements are copies of the smallest guiding element. This means that each guiding element is stored $2\delta + 1$ times and can be retrieved reliably. The minimum elements are $2\delta + 1$ copies of the guiding element for the leftmost leaf in the subtree defined by u, stored consecutively in $\lceil \frac{2\delta+1}{B} \rceil$ blocks. Similarly the maximum elements are $2\delta + 1$ copies of the guiding element for the leaf following the rightmost leaf in the subtree defined by u, stored consecutively in $\lceil \frac{2\delta+1}{B} \rceil$ blocks. Additionally, minimum and maximum elements are stored with each leaf. The minimum are 4δ copies of the guiding element representing the leaf, stored consecutively in $\frac{4\delta}{B}$ blocks, and the maximum elements are 4δ copies of the guiding element representing the subsequent leaf, stored consecutively in $\frac{4\delta}{B}$ blocks. These are used to verify that the algorithm found the only leaf that may store an uncorrupted element matching the search element.

Query: A query operation for an element q, uses an index k that indicates how many chunks of B^{1-c} elements the algorithm has discarded during the search,

initially $k = 0$. Intuitively, a chunk is discarded if the algorithm detects that $\Omega(B^{1-c})$ of its elements are corrupted. The query operation traverses the tree top-down, storing in safe memory the index k, and $O(1)$ extra variables required to traverse the tree using the knowledge of its layout on disk. In an internal node u, the algorithm starts by checking whether u is on the correct path in the tree using the copies of the minimum and maximum elements stored in u. This is done by scanning B^{1-c} of the $2\delta + 1$ copies of the minimum element starting with the kB^{1-c}'th copy, counting how many of these that are larger than q. If $B^{1-c}/2$ or more copies of the minimum element are larger than q the block is discarded by incrementing k and the search is restarted (backtracked) at node v, where $v = u$ if u is root of the tree and the parent of u otherwise. The maximum elements are checked similarly. If the algorithm backtracks, k is increased ensuring that the same element is never read more than once.

If the checks succeed the k'th block storing copies of the B^c guiding elements of u is scanned from left to right. The majority value of each of the B^{1-c} copies of each guiding element is extracted in sorted order using the majority algorithm [18] and compared to q, until a retrieved guiding element larger than q is found or the entire block is read. The traversal then continues to the corresponding child. If any invocation of the majority algorithm fails to select a value, or two fetched guiding elements are out of order, the block is discarded as above by increasing k and backtracking the search to the parent node.

Upon reaching a leaf, the algorithm verifies whether the search found the correct leaf. This is achieved by running a variant of the verification procedure designed for the same purpose in [11]. Counters c_l and c_r, which are initially 1, are stored in safe memory. Then the copies of the minimum and maximum element are scanned in chunks of B^{1-c} elements, starting from the $2kB^{1-c}$'th element. If the majority of elements in a chunk of B^{1-c} copies of the minimum element are smaller than the search element, c_l is increased by 1. Otherwise, c_l is decreased and k increased by one. The copies of maximum elements are treated similarly. Note that every decrement of c_l or c_r signals that at least $\frac{B^{1-c}}{2}$ corruptions have been found. Thus, c_l represents the number of chunks scanned that has not yet been contradicted, where the majority of copies indicates that the search element is in the current leaf or in leafs to the right. Similar for c_r. If $\min\{c_l, c_r\}$ reaches 0, we backtrack to the parent of the leaf as above. If $\min\{c_l, c_r\}\frac{B^{1-c}}{2}$ gets larger than $\delta - k(\frac{B^{1-c}}{2}) + 1$ the verification succeeds. The algorithm finishes by scanning the δ elements stored in the leaf, returning whether it finds q or not.

Lemma 2. *The data structure is a linear space resilient dictionary supporting queries in* $O\left(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B}\right)$ *I/Os, for any* $1/\log B \leq c \leq 1$.

The correctness portion of the proof is similar to the proof for the optimal binary search algorithm in [11]. The complexity analysis uses the observation that if a search is guided in the wrong direction, the majority of the B^{1-c} copies of a guiding element in the relevant block are corrupted and each additional step requires $\frac{B^{1-c}}{2}$ additional corruptions in order to pass the check against the minimum and maximum elements. Details will appear in the full paper.

To obtain optimal time bounds for the dictionary, we use the resilient binary search algorithm of [11] on each block, instead of scanning it. If more than $\frac{B^{1-c}}{2}$ corruptions are discovered during the search of a block, it is discarded as above. Otherwise, $\frac{B^{1-c}}{2}$ supporting elements are found on both sides of an element, and the algorithm continues to the corresponding child as before. This reduces the time used per node to $O(\log B + B^{1-c})$. Verification takes $O(\delta)$ time in total.

Lemma 3. *For any $\frac{1}{\log B} \leq c \leq 1$, queries use $O((B^{1-c} + \log B)(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}}) + \delta)$ time.*

Corollary 3. *If $\delta > B \log N$, queries use $O(\frac{\delta}{B})$ I/Os and $O(\delta)$ time.*

Corollary 4. *If $\delta < \log N$, queries use $O(\log_B N)$ I/Os and $O(\log N)$ time.*

Corollary 5. *If $\log N \leq \delta \leq B \log N$ for any $\frac{1}{\log B} \leq c \leq 1$, queries use $O(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B})$ I/Os and $O(\log n + \delta)$ time.*

The corollaries follow from Lemma 2 and 3 by setting $c = \frac{1}{\log B}$, $c = 1 - \frac{\log\log B}{\log B}$, and $c \in [\frac{1}{\log B}, 1 - \frac{\log\log B}{\log B}]$ such that $\frac{1}{c}\log_B N = \frac{\delta}{B^{1-c}}$ respectively.

By adapting the techniques of [11, 19] and the static dictionary presented above we obtain a dynamic dictionary. Details will appear in the full paper.

Theorem 3. *There is a deterministic dynamic resilient dictionary supporting searches and updates in $O(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B})$ I/Os and $O(\log N + \delta)$ time, worst-case and amortized respectively with c in the range $\frac{1}{\log B} \leq c \leq 1$.*

5 Priority Queue and Sorting

In this section we present a resilient multi-way merging algorithm and use it to design a resilient sorting algorithm and priority queue. First we show how to merge γ faithfully ordered lists of total size x when $\gamma \leq \min\{\frac{M}{B}, \frac{M}{\delta}\}$.

Multi-way Merging: Initially, the algorithm constructs a perfectly balanced binary tree, T, in memory on top of the γ buffers being merged. Each edge of the binary tree is equipped with a buffer of size $5\delta + 1$. Each internal node $u \in T$ stores the state of a running instance of the *PurifyingMerge*, a resilient binary merging algorithm from [10] that works in rounds. In each round $O(\delta)$ elements from both input buffers are read and the next δ elements in the faithful order are output. If corrupted elements are found, these are moved to a fail buffer and the round is restarted. The algorithm merges elements from the buffers on u's left child edge and right child edge into the buffer of u's parent edge. The states and sizes of all buffers are stored as reliable variables. The entire tree including all buffers and state variables are stored in internal memory, along with one block from each of the γ input streams and one block for the output stream of the root. Instead of storing a fail buffer for each instance of *PurifyingMerge*, a global shared fail buffer F is stored containing all detected corrupted elements.

Let b_l and b_r be the buffers on the edges to the left and right child respectively and let b denote the buffer on the edge from u to its parent. If u is the root, b is the output buffer. The elements are merged using the *fill* operation, which operates on u, as follows. First, it checks whether b_l and b_r contain at least $4\delta + 1$ elements, and if not they are filled recursively. Then, the stored instance of the *PurifyingMerge* algorithm is resumed by running a round of the algorithm outputting the next δ elements to its output stream. The multi-way merging algorithm is initiated by invoking *fill* on the root of T which runs until all elements have been output. Then, the elements moved to F during the *fill* are merged into the output using *NaiveSort* and *UnbalancedMerge* as in [10].

Lemma 4. *Merging* $\gamma = \min\{\frac{M}{B}, \frac{M}{\delta}\}$ *buffers of total size* $x \geq M$ *using* $O(x/B)$ *I/Os and* $O(x \log \gamma + \alpha\delta)$ *time.*

Proof. The correctness follows from Lemma 1 in [10]. The size of T is $O(\gamma(\delta + B)) = O(\min\{\frac{M}{B}, \frac{M}{\delta}\}(\delta + B)) = O(M)$. We use γ I/Os to load the first block in each leaf of T and $O(x/B)$ I/Os for reading the entire input and writing the output. The final merge with F takes $O(x/B)$ I/Os. Since T fits completely in memory we perform no other I/Os.

Merging two buffers of total size n using *PurifyingMerge* takes $O(n + \alpha\delta)$ time where α is the number of detected corruptions in the input buffers. Since detected corruptions are moved to the global fail buffer each corruption is only charged once. Each element passes through $\log \gamma$ nodes of T and the final merge using *NaiveSort* and *UnbalancedMerge* takes $O(x + \alpha\delta)$ time. □

Sorting: Assuming $\delta \leq M^\varepsilon$ for $0 \leq \varepsilon < 1$, we can use the multi-way merging algorithm to implement the standard external memory $M^{1-\varepsilon}$-way mergesort from [8] matching the optimal external memory sorting bound for constant ε.

Theorem 4. *Our resilient sorting algorithm uses* $O(\frac{1}{1-\varepsilon}Sort(N))$ *I/Os and* $O(N \log N + \alpha\delta)$ *time assuming* $\delta \leq M^\varepsilon$.

Priority Queue: Our comparison based resilient priority queue is optimal with respect to both time and I/O performance assuming that $\delta \leq M^\varepsilon$ for $0 \leq \varepsilon < 1$. An optimal I/O-efficient priority queue uses $\Theta(1/B \log_{M/B}(N/M))$ I/Os amortized per operation [8]. An $\Omega(\log N + \delta)$ time lower bound for comparison based resilient priority queues was proved in [14]. A resilient priority queue as defined in [14] maintains a set of elements under the operations *insert* and *delete-min*, where *insert* adds an element and a *delete-min* deletes and returns the minimum uncorrupted element or a corrupted one.

Our priority queue is based on an amortized version of the worst-case optimal external memory priority queue of [20] using our new resilient multi-way merging algorithm to move elements between disk and internal memory. Details will appear in the full paper.

Theorem 5. *There is a linear space resilient priority queue supporting* insert *and* delete-min *in amortized* $O(\frac{1}{1-\varepsilon}(1/B) \log_{M/B}(N/M))$ *I/Os and* $O(\log N + \delta)$ *time assuming* $\delta \leq M^\varepsilon$ *where* $0 \leq \varepsilon < 1$.

References

1. Tezzaron Semiconductor: Soft errors in electronic memory - a white paper. http://www.tezzaron.com/about/papers/papers.html (2004)
2. van de Goor, A.J.: Testing Semiconductor Memories: Theory and Practice. ComTex Publishing, Gouda, The Netherlands (1998) ISBN 90-804276-1-6.
3. Huang, K.H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers **33** (1984) 518–528
4. Rela, M.Z., Madeira, H., Silva, J.G.: Experimental evaluation of the fail-silent behaviour in programs with consistency checks. In: Proc. 26th Annual International Symposium on Fault-Tolerant Computing. (1996) 394–403
5. Yau, S.S., Chen, F.C.: An approach to concurrent control flow checking. IEEE Transactions on Software Engineering **SE-6**(2) (1980) 126–137
6. Pradhan, D.K.: Fault-tolerant computer system design. Prentice-Hall, Inc. (1996)
7. Finocchi, I., Italiano, G.F.: Sorting and searching in the presence of memory faults (without redundancy). In: Proc. 36th Annual ACM Symposium on Theory of Computing, New York, NY, USA, ACM Press (2004) 101–110
8. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Commun. ACM **31** (1988) 1116–1127
9. Finocchi, I., Grandoni, F., Italiano, G.F.: Designing reliable algorithms in unreliable memories. Computer Science Review **1**(2) (2007) 77–87
10. Finocchi, I., Grandoni, F., Italiano, G.F.: Optimal resilient sorting and searching in the presence of memory faults. Theoretical Computer Science, ICALP'06 issue (2009)
11. Brodal, G.S., Fagerberg, R., Finocchi, I., Grandoni, F., Italiano, G., Jørgensen, A.G., Moruz, G., Mølhave, T.: Optimal resilient dynamic dictionaries. In: Proc. 14th Annual European Symposium on Algorithms. Volume 4708., Springer Verlag, Berlin (2007) 347–358
12. Petrillo, U.F., Finocchi, I., Italiano, G.F.: The price of resiliency: a case study on sorting with memory faults. In: Proc. 14th Annual European Symposium on Algorithms. (2006) 768–779
13. Finocchi, I., Grandoni, F., Italiano, G.F.: Resilient search trees. In: Proc. 18th ACM-SIAM Symposium on Discrete Algorithms. (2007) 547–554
14. Jørgensen, A.G., Moruz, G., Mølhave, T.: Priority queues resilient to memory faults. In: Proc. 10th International Workshop on Algorithms and Data Structures. (2007) 127–138
15. Arge, L.: External memory data structures. In Abello, J., Pardalos, P.M., Resende, M.G.C., eds.: Handbook of Massive Data Sets. Kluwer Academic Publishers (2002) 313–358
16. Vitter, J.S.: Algorithms and data structures for external memory. Foundations and Trends in Theoretical Computer Science **2**(4) (2008) 305–474
17. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indexes. Acta Informatica **1** (1972) 173–189
18. Boyer, R.S., Moore, J.S.: MJRTY: A fast majority vote algorithm. In: Automated Reasoning: Essays in Honor of Woody Bledsoe. (1991) 105–118
19. Brodal, G.S., Fagerberg, R., Jacob, R.: Cache-oblivious search trees via binary trees of small height. In: Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms. (2002) 39–48
20. Brodal, G.S., Katajainen, J.: Worst-case efficient external-memory priority queues. In: Proc. 6th Scandinavian Workshop on Algorithm Theory. Volume 1432 of Lecture Notes in Computer Science. Springer Verlag, Berlin (1998) 107–118