Data Structures Resilient to Memory Faults: An Experimental Study of Dictionaries

Umberto Ferraro-Petrillo[1], Fabrizio Grandoni[2], and Giuseppe F. Italiano[2]

[1] Dipartimento di Statistica, Probabilità e Statistiche Applicate, Sapienza Università di Roma, P.le Aldo Moro 5, 00185 Rome, Italy. Email: `umberto.ferraro@uniroma1.it`.

[2] Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, Via del Politecnico 1, 00133 Roma, Italy. Email: {`grandoni,italiano`}`@disp.uniroma2.it`.

Abstract. We address the problem of implementing data structures resilient to memory faults which may arbitrarily corrupt memory locations. In this framework, we focus on the implementation of dictionaries, and perform a thorough experimental study using a testbed that we designed for this purpose. Our main discovery is that the best-known (asymptotically optimal) resilient data structures have very large space overheads. More precisely, most of the space used by these data structures is not due to key storage. This might not be acceptable in practice since resilient data structures are meant for applications where a huge amount of data (often of the order of terabytes) has to be stored. Exploiting techniques developed in the context of resilient (static) sorting and searching, in combination with some new ideas, we designed and engineered an alternative implementation which, while still guaranteeing optimal asymptotic time and space bounds, performs much better in terms of memory without compromising the time efficiency.

1 Introduction

Memories in modern computing platforms are not always fully reliable, and sometimes the content of a memory word may be corrupted. This may depend on manufacturing defects, power failures, or environmental conditions such as cosmic radiation and alpha particles [14,19]. This type of phenomena can seriously affect the computation, especially when the amount of data to be processed is huge and the storage devices are inexpensive. This is for example the case for Web search engines, that store and process terabytes of dynamic data sets, including inverted indices which have to be maintained sorted for fast document access. For such large data structures, even a small failure probability can result in bit flips in the index, which may become responsible of erroneous answers to keyword searches [15,16].

The classical way to deal with memory faults is via error detection and correction mechanisms, such as redundancy, Hamming codes, etc. These traditional approaches imply non-negligible costs in terms of time and money, and thus they are not always adopted in large-scale clusters of PCs. Hence, it makes sense to try to solve the problem at the application level, i.e., to design algorithms and data structures which are resilient to memory faults. Dealing with unreliable information has been addressed in the algorithmic community in a variety of different settings, including the liar model [1,4,8,17,20], fault-tolerant sorting networks [2,18,21], resiliency of pointer-based data structures [3], and parallel models of computation with faulty memories [7].

The Faulty-RAM Model. In this paper we focus on the *faulty-RAM* model introduced in [12,13][3]. In this model, an adaptive adversary can corrupt any memory word, at any time, by overwriting its value. Corrupted values cannot be (directly) distinguished from correct ones. An upper bound δ is given on the total number of memory faults that can occur throughout the execution of an algorithm or during the lifetime of a data structure. However, we can exploit $O(1)$ *safe* memory words, whose content never gets corrupted. The adaptive adversary knows the algorithm and the state of safe and unsafe memory at any time. Furthermore, it can react to the actions of the algorithm. In other terms, corruptions do not need to be scheduled *a priori* (this is relevant for randomized algorithms). However, the adversary cannot access the sequence of random bits used by a randomized algorithm. Furthermore, read operations are considered atomic, i.e., the adversary cannot corrupt a memory word right after the algorithm starts to read it.

A natural approach to the design of algorithms and data structures in the faulty-RAM model is data replication. Informally, a *resilient variable* consists of $(2\delta + 1)$ copies x_1, x_2,..., $x_{2\delta+1}$ of a standard variable. The value of a resilient variable is given by the majority of its copies (which can be computed in linear time and constant space [5]). Observe that the value of x is reliable, since the adversary cannot corrupt the majority of its copies. The approach above induces a $\Theta(\delta)$ multiplicative overhead in terms of both space and running time. For example, a trivially-resilient implementation of a standard dictionary based on AVL trees would require $O(\delta n)$ space and $O(\delta \log n)$ time for each `search`, `insert` and `delete` operation. Thus, it can tolerate only $O(1)$ memory faults while maintaining optimal time and space asymptotic bounds.

This type of overhead seems unavoidable if one wishes to operate correctly in the faulty-RAM model. For example, with less than $2\delta + 1$ copies of a key, we cannot avoid that its correct value gets lost. Since a $\Theta(\delta)$ multiplicative overhead could be unacceptable in several applications even for small values of δ, the next natural thing to do is relaxing the notion of correctness. We say that an algorithm or data structure is *resilient* to memory faults if, despite the corruption of some memory location during its lifetime, it is nevertheless able to operate correctly (at least) on the set of uncorrupted values.

In [10,13], the *resilient sorting* problem is considered. Here, we are given a set K of n keys. A key is a (possibly negative) real value. We call a key *faithful* if it is never corrupted, and *faulty* otherwise. The problem is to compute a *faithfully*

[3] Due to space constraints, we refer to [12,13] for a detailed description of the model.

sorted permutation of K, that is a permutation of K such that the subsequence induced by the faithful keys is sorted. This is the best one can hope for, since the adversary can corrupt a key at the very end of the algorithm execution, thus making faulty keys occupy wrong positions. This problem can be trivially solved in $O(\delta\, n \log n)$ time. In [13], an $O(n \log n + \delta^3)$ time algorithm is described together with a $\Omega(n \log n + \delta^2)$ lower bound. A sorting algorithm `ResSort` with optimal $O(n \log n + \delta^2)$ running time is later presented in [10]. In the special case of polynomially-bounded integer keys, an improved running time of $O(n + \delta^2)$ can be achieved [10]. In [9] an experimental study of resilient sorting algorithms is presented. The experiments show that a careful algorithmic design can have a great impact on the performance and reliability achievable in practice.

The *resilient searching* problem is studied in [10,13]. Here we are given a faithfully sorted sequence K of n keys, and a search key κ. The problem is to return a key (faulty or faithful) of value κ, if K contains a faithful key of that value. If there is no faithful key equal to κ, one can either return *no* or return a (faulty) key equal to κ. Note that, also in this case, this is the best possible: the adversary may indeed introduce a corrupted key equal to κ at the very beginning of the algorithm, such that this corrupted key cannot be distinguished from a faithful one. Hence, the algorithm might return that corrupted key both when there is a faithful key of value κ (rather than returning the faithful key), and when such faithful key does not exist (rather than answering *no*). There is a trivial algorithm which solves this problem in $O(\delta \log n)$ time. A lower bound of $\Omega(\log n + \delta)$ is described in [13] for deterministic algorithms, and later extended to randomized algorithms in [10]. A $O(\log n + \delta^2)$ deterministic algorithm is given in [13]. An optimal $O(\log n + \delta)$ randomized algorithm `ResSearch` is provided in [10]. An optimal $O(\log n + \delta)$ deterministic algorithm is eventually given in [6]. For both resilient sorting and searching, the space usage is $O(n)$.

Resilient Dictionaries. More recently, the problem of implementing resilient data structures has been addressed. A *resilient dictionary* is a dictionary where the `insert` and `delete` operations are defined as usual, while the `search` operation must be resilient as described before. In [11], Finocchi et al. present a resilient dictionary using $O(\log n + \delta^2)$ amortized time per operation. In [6], Brodal et al. present a simple randomized algorithm `Rand` achieving optimal $O(\log n + \delta)$ time per operation. Using an alternative, more sophisticated approach, they also obtain a deterministic resilient dictionary `Det` with the same asymptotic performances. For all the mentioned implementations, the space usage is $O(n)$, which is optimal. However, as we will see, the constant hidden in the latter bound is not negligible in practice.

We next give some more details about `Rand`, since it will be at the heart of our improved implementation `RandMem`, which is described in Section 3. The basic idea is maintaining a dynamically evolving set of intervals spanning $(-\infty, +\infty)$, together with the corresponding keys. Intervals are merged and split so that at any time each interval contains $\Theta(\delta)$ keys. More precisely, each interval is implemented as a buffer of size 2δ, which contains between $\delta/2$ and 2δ keys at any time. Intervals are stored in a classical AVL tree, where standard variables (pointers, interval boundaries, etc.) are replaced by resilient variables. The number of intervals is at most $1 + \frac{n}{\delta/2}$. For each interval, we store a buffer of size 2δ plus a constant number of resilient variables, each one of size $2\delta + 1$. Hence, the overall space usage is $O(n + \delta)$. A `search` is performed in the standard way, where, instead of reading the $2\delta + 1$ copies of each relevant variable, the algorithm only reads one of those copies uniformly at random. At the end of the search, the algorithm reads reliably (in $\Theta(\delta)$ time) the boundaries of the final interval, in order to check whether they include the searched key. If not, the process is started from scratch. Otherwise, the desired key is searched for by linearly scanning the buffer associated to the interval considered. Operations `insert` and `delete` are performed analogously. In particular, the insertion of a key already present in the dictionary is forbidden (though duplicated keys might be inserted by the adversary). When, after one `insert`, one interval contains 2δ keys, it is split in two halves. When, after one `delete`, one interval contains $\delta/2$ keys, it is merged with a boundary interval. If the resulting interval contains more than $3\delta/4$ keys, it is split in two halves. The modifications of the interval set above involve a modification of the search tree of cost $O(\delta \log n + \delta^2)$. However, this cost is amortized over sequences of $\Omega(\delta)$ `insert` and `delete` operations. We remark that, for $\delta > n$, it is sufficient to maintain all the keys as an unsorted sequence into a buffer of size $O(n)$. In this case, each operation can be trivially implemented in $O(n) = O(\delta)$ time. Hence, the space usage can be reduced to $O(n)$ without increasing the running time.

The Experimental Framework. In this paper we focus our attention on resilient dictionaries. We perform an experimental evaluation of the optimal dictionaries `Det` and `Rand`, together with an improved implementation `RandMem` developed by ourselves. In order to evaluate the drawbacks and benefits of resilient implementations, we also consider a standard (non-resilient) implementation of a search tree. In particular, we implemented an AVL binary search tree called `Avl`, in order to make a more direct comparison with `Rand` (which builds upon the same data structure).

In order to test different data structures, we use the same testbed to simulate the faulty-RAM model as in [9][4]. Shortly, we model the data structure and the adversary as two separate parallel threads. The adversary thread is responsible for injecting $\alpha \leq \delta$ faults during the lifetime of the data structure. In order to inject one fault, the adversary selects one unsafe memory word (among the ones used by the data structure) uniformly at random, and overwrites it with a random value. In order to inject α faults, the adversary samples α operations uniformly at random over a given sequence of operations, and injects exactly one random fault during each sampled operation. We performed experiments both on random inputs and on real-world inputs[5]. In random inputs, the instances

[4] The reader is referred to [9] for a more detail description of the testbed.

[5] Our experiments have been carried out on a workstation equipped with two Opteron processors with 2 GHz clock rate and 64 bit address space, 2 GB RAM, 1 MB L2 cache, and 64 KB L1 data/instruction cache. The workstation runs Linux Kernel 2.6.11. All programs have been compiled through the GNU `gcc` com-

consist of a sequence of random operations. A `random insert` simply inserts a random value in a given range $[\ell, r]$ (the actual range is not really relevant). In a `random search` we search for a key κ, where, with probability $1/2$, κ is chosen uniformly at random among the keys currently in the dictionary, and otherwise is set to a random value in $[\ell, r]$. In a `random delete`, we delete a random key κ, where κ is generated as in the case of the `random search`. We also performed experiments with non-random instances, involving the set of words in one English dictionary and a few English books.

2 Evaluation of Existing Dictionaries

In this section we report the results of our experimental study on the asymptotically optimal resilient dictionaries `Det` and `Rand`, plus a standard (non-resilient) implementation `Avl` of an AVL binary search tree. All the results reported here are averaged over 20 (random) input instances. We observed that those results do not change substantially by considering a larger number of instances. For each experiment discussed here, we let $\delta = 2^i$ for $i = 2, 3, \ldots, 10$. This range of values of δ includes both cases where the running time is dominated by the $O(\log n)$ term and cases where the $O(\delta)$ term dominates.

The Importance of Being Resilient. First of all, we wish to test how much the lack of resiliency affects the accuracy of a non-resilient dictionary. To that aim, we measured the fraction of `search` operations which fail to provide a correct answer in `Avl` (which is not resilient), for increasing values of δ. Since `Avl` is affected only by the actual number of faults, in all the experiments we assumed $\alpha = \delta$.

We observed experimentally that even a few memory faults make `Avl` crash very soon, due to corrupted pointers. In order to make a more meaningful test, we implemented a variant of `Avl` which halts the current operation without crashing when that operation tries to follow a corrupted pointer. Even with this (partially resilient) variant of `Avl`, very few faults make a large fraction of the `search` operations fail. This is not surprising, since the corruption of a pointer at top levels in the AVL tree causes the loss of a constant fraction of the keys.

In order to illustrate this phenomenon, let us consider the following input sequence. First of all, we populate the dictionary via a sequence of 10^6 `random insert` operations. Then we corrupt the data structure by injecting $\alpha = \delta$ faults. Finally, we generate a sequence of 10^5 `random search` operations, and count how many operations fail because of a corrupted pointer.

Our results for this case are shown in Figure 1(a). As expected, the number of incorrect `search` operations grows with $\alpha = \delta$. More interestingly, the expected number of wrong operations is roughly one order of magnitude larger than the number of faults. For example, in the case $\delta = 1024$ (i.e., δ is roughly 0.1% of

piler version 3.3.5 with optimization level `O3`. The full package, including algorithm implementations, and a test program, is publicly available at the URL: `http://www.statistica.uniroma1.it/users/uferraro/experim/faultySearch/`.

the number of keys), roughly 1100 `search` operations fail (i.e., roughly 1% of the operations). This suggests that using resilient implementations can be really worth the effort.

The Cost of Resiliency. In order to evaluate how expensive resiliency is, we experimentally compared the time and space performance of `Det`, `Rand` and `Avl`. In order to test the sensitivity of resilient algorithms to the actual number of faults α (besides to δ), we considered different values of the ratio α/δ: we next provide results only for the extreme cases $\alpha = 0$ and $\alpha = \delta$. For `Avl` we only considered $\alpha = 0$, since in the presence of faults the space and time usage of this non-resilient dictionary is not very meaningful (as shown in previous subsection).

The results obtained for the following input instance summarize the qualitative behaviors that we observed. We bulk-load the dictionary via a random sequence of 10^6 `random insert` operations. Then we generate a sequence of 10^5 operations, where each operation is uniformly chosen to be a `random insert`, `random search`, or `random delete`. The adversary injects α faults during the last 10^5 operations. We consider the average time per operation of the latter operations. The space usage is measured at the end of the process.

The results concerning the running time for the above instance are shown in Figure 1(b)-(c). As expected, `Avl` is much faster than the resilient dictionaries. This suggests that resilient implementations should be used only when the risk of memory faults is concrete.

Interestingly enough, the time performance of `Rand` and `Det` is very sensitive to δ, but it is almost not affected by α. This suggests that, in any practical implementation, δ should be chosen very carefully: underestimating δ compromises the resiliency of the dictionary while overestimating it might increase the running time drammatically.

`Rand` is rather faster than `Det` for large values of δ, but it is much slower than `Det` when δ is small. Not surprisingly, the running time of `Det` grows with δ. More interestingly, the running time of `Rand` initially quickly decreases with δ and then slowly increases. This behavior arises from the fact that, for small values of δ, `Rand` often restructures the AVL search tree in order to keep the number of keys in each interval within the range $[\delta/2, 2\delta]$: this operation dominates the running time. This interpretation is confirmed in next section, where we consider a variant of `Rand` which maintains intervals with a number of keys in $[a\delta/2, 2a\delta]$, for a proper constant $a > 1$. For example, in the case $a = 32$ the running time of this variant of `Rand` is monotonically increasing in δ.

The results concerning the space usage for the mentioned case are shown in Figure 1(d). The space usage in the figure are given as multiples of the total space occupied by keys (which is a lower bound on the space needed). We first of all observe that the space usage of `Rand` and `Det` is almost not affected by δ (this is obvious in the case of `Avl`). In particular, the space usage initially decreases with δ, and then reaches a stationary value rather quickly. This might seem surprising at a first glance. The reason for this behavior is that both dictionaries exploit a data structure containing $\Theta(n/\delta)$ nodes, each one of size $\Theta(\delta)$. Hence the space usage is $\Theta(n)$, irrespectively of δ. The experiments show that even the constants in the asymptotic notation are weakly related to δ.

Not surprisingly, `Rand` uses much more space than `Avl`. What is more surprising is that `Det` uses much less space than `Avl` (despite the fact that `Avl` does not need to take care of faults). This behavior might be explained by the fact that `Det` builds upon data structures developed in the context of algorithms for external memory. In more detail, the combination of buffering techniques and lazy updates in `Det` reduces the use of pointers with respect to `Avl`.

We remark that all dictionaries use much more space than the space occupied by keys only. In particular, the space usage of `Rand`, `Avl`, and `Det` is roughly 16, 10, and 5 times the total size of the keys, respectively. This space overhead is determined by the use of pointers for all those implementations. Furthermore, in the case of `Rand` and `Det` part of the space is wasted due to buffers which are partially empty. Such a large space overhead may not be acceptable in the applications, where keys alone already occupy huge amounts of memory. This is also the main motivation for the refined implementation `RandMem` described in next section.

3 A Refined Resilient Dictionary

Motivated by the large space overhead of `Rand`, in this section we describe a new randomized resilient dictionary `RandMem`, which is a (non-trivial) variant of `Rand`. `RandMem` has optimal asymptotic time and space complexity, but it performs better in practice. In particular, it uses an amount of space closer to the lower bound given by the total space occupied by keys. Furthermore, it is sometimes slightly slower and often even faster than `Rand` and `Det`.

Our refined data structure is based on a careful combination of the results developed in the context of static sorting and searching in faulty memories [10,13], together with some new, simple ideas. This machinery is exploited to implement more efficiently the part of each operation which involves the keys in a given interval. The rest of the implementation is exactly as in `Rand`. In particular, the structure of the AVL tree and the way it is explored and updated is the same as before. Rather than describing directly `RandMem`, we illustrate the logical steps which led us to its development.

Reducing Space Usage. A simple way to reduce the space overhead of `Rand` is modifying the algorithm so that, for a proper parameter $a > 1$, the number of keys in each interval is in the range $[a\delta/2, 2a\delta]$ rather than $[\delta/2, 2\delta]$ (adapting the update operations consequently). Intuitively, the larger is a, the smaller is the number of intervals and hence the space overhead. This allows one to reach asymptotically a space usage of at most 4 times the total space occupied by keys, the worst case being when all the intervals contain $a\delta/2$ keys each (while the space reserved is roughly $2a\delta$ per interval). In practice, the space usage might even be smaller since we expect to see an intermediate number of keys in each interval (rather than a number close to the lower bound).

We tested this variant of `Rand`, that we called `RandMem`(a), for growing values of a. As expected, the space usage is a decreasing function of a. In Figure 1(d) we report on the space usage of `RandMem`(32) on the same input instances used in Section 2. The memory usage is much smaller than the one of `Rand` and `Det`, and it is roughly twice the space occupied by keys. In our experiments, larger values of a do not provide any substantial reduction of the space usage.

We remark that it is not hard to achieve a space usage arbitrarily close to the total size of the keys (for growing values of a). The idea is requiring that each interval contains between $(1-\epsilon)a\delta$ and $(1+\epsilon)a\delta$ keys, for a small constant $\epsilon > 0$. Of course, this alternative implementation implies a more frequent restructuring of the search tree, which a consequent negative impact on the running time (in particular, the running time is an increasing function of $1/\epsilon$). We do not discuss here the experimental results for this alternative implementation due to space constraints.

In the following A denotes the buffer (of size $2a\delta$) containing the keys associated to the interval I under consideration. We implicitly assume that empty positions of A (to the right of the buffer) are marked with a special value ∞, which stands for a value larger than any feasible key. Note that the adversary is allowed to write ∞ in a memory location. In the next paragraphs we show how to speed up each operation.

Reducing Searching Time. The main drawback of the approach described above is that, in each operation, `RandMem`(a) needs to linearly scan a buffer A of size $\Theta(a\delta)$: for large values of a this operation is very expensive. This is witnessed by the experiment shown in Figure 1(e), where we report on the running time of `RandMem`(32) for the same input instances as in Section 2.

One way to reduce the `search` time is keeping all the keys in A faithfully sorted. Each time we insert or delete a key from A, we faithfully sort its keys with a (static) resilient sorting algorithm: here we used the optimal resilient sorting algorithm `ResSort` described in [10]. In order to search for a key, we exploit a resilient (static) searching algorithm. In particular, we decided to implement the simple randomized searching algorithm `ResSearch` in [10].

Reducing Insertion Time. Of course, keeping A faithfully sorted makes `insert` and `delete` operations more expensive.

In order to reduce the `insert` time, without increasing substantially the time needed for the other two operations, we introduce a secondary buffer B of size $b\delta$, for a proper constant $a > b > 0$. All the keys are initially stored in B (which is maintained as an unsorted sequence of keys). Like for A, empty positions of B are set to ∞. When B is full, buffers A and B are merged into A, so that the resulting buffer A is faithfully sorted. In order to perform this merging, first of all we faithfully sort B by means of the classical `SelectionSort` algorithm (which turns out to be a resilient sorting algorithm [10]). For small enough b, `SelectionSort` is faster than `ResSort`. Then we apply to A and B the procedure `UnbalancedMerge`, which is one of the key procedures used in `ResSort`. This procedure takes as input two faithfully sorted sequences, one of which is much shorter than the other, and outputs a faithfully sorted sequence containing all the keys of the original sequences. We remark that merging two faithfully sorted sequences is faster than sorting everything from scratch (using, e.g., `ResSort`). Of course, now `search` and `delete` operations have to take buffer B into consideration. In particular, those operations involve a linear scan of B.

The main advantage of B is that it allows to spend $O(b\delta)$ time per insertion, and only after $\Omega(b\delta)$ insertions one needs to merge A and B, which costs $O((b\delta)^2 + a\delta + \delta^2)$ time. However, this also introduces a $\Theta(b\delta)$ overhead in each `search` and `delete` operation. Henceforth, b has to be tuned carefully.

Reducing Deletion Time. It remains to reduce the `delete` time, without increasing too much the time needed for the other operations. Deleting an element in B is a cheap operation, therefore we focus on the deletion of an element in A.

A natural approach to delete an element is replacing it with the special value ∞ (recall that empty positions are set to ∞ as well). When A and B are merged after one `insert`, we can easily get rid of this extra values ∞. Note that the ∞ entries introduced by deletions are not correctly sorted despite the fact that they are not faults introduced by the adversary. As a consequence, `ResSearch` is not guaranteed to work correctly. However, we can solve the problem by letting `ResSearch` run with respect to a number $\delta' = x + \delta$ of faults, where x is the current number of deleted elements. In other terms, we can consider the x deleted entries as faults introduced by the adversary. When x is large, the `search` (and hence `delete`) operation becomes slower. Hence, we fix a threshold $\tau = c\delta$ for a proper constant $c > 0$. When x reaches τ, we *compress* A so that the values different from ∞ occupy the first positions in the buffer (while the final ∞ entries correspond to empty positions). The $\Theta(a\delta)$ cost of this compression is amortized over $\Omega(c\delta)$ deletions.

It remains to explain how we keep track of x. One natural way to do that is using a resilient variable (i.e., $2\delta + 1$ copies of one variable) as a counter. Each time a new deletion occurs, we increment the counter and we reset it when it reaches τ (this costs roughly 4δ per `delete`). Here we adopt an alternative, novel approach, which is better for small values of c both in terms of time and of space. We define an array C of size τ, which is used as a *unary counter*. In particular, each time a new deletion occurs, we linearly scan C, searching for a 0 entry, and we replace it with a 1. If no 0 entry exists, C is reset to 0 and A is compressed. The number of 1's in C, i.e. the number of deletions in A, is denoted by $|C|$. Note that, differently from the case of the resilient counter, the adversary might reset some entries of C: in that case $|C|$ underestimates the actual number of deletions in A. In principle, this might compromise the correctness of `ResSearch`. However, running `ResSearch` with $\delta' = |C| + \delta$ is still correct. In fact, let α_C and α_A be the total number of faults occurring in C and A, respectively. The number of deletions in A is at most $|C| + \alpha_C$ (each deletion in A not counted by C corresponds to a fault in C). Hence, the total number of unsorted elements in A (faults plus deletions) is at most $|C| + \alpha_C + \alpha_A \leq |C| + \delta$. Of course, the adversary might as well set some 0 entries of C to 1, hence anticipating the compression of A. However, $\Omega(c\delta)$ such faults are needed to force a compression of cost $O(a\delta)$. Hence, this type of phenomenon does not affect the running time substantially.

We remark that unary counters were not used before in the literature on reliable algorithms and data structures.

We next call RandMem(a,b,c) the variant of RandMem(a) which exploits the secondary buffer B and the unary counter C. It is worth to remark that the secondary buffer and the deletions might compromise the asymptotic optimality of the dictionary. In particular, it might happen that the set of intervals (and hence the AVL tree) is modified more often than every $\Omega(\delta)$ operations. By a simple computation, it turns out that $b \leq a/2$ and $c \leq a/8$ are sufficient conditions to avoid this situation, hence maintaining optimal $O(n)$ space complexity and $O(\log n + \delta)$ time per operation.

Experimental Evaluation. We tested RandMem(a,b,c) in different scenarios and for different values of the triple (a, b, c). We next restrict our attention to the input instances as considered in Section 2, with $\alpha = \delta$. Furthermore, we assume $a = 32$, which minimizes the space usage.

We experimentally observed that, for a given value of a, the running time of the data structure may vary greatly according to the choice of c while it is only partially influenced by the choice of b. For fixed values of a and b, the running time first decreases and then increases with c. This is the result of two opposite phenomena. On one hand, small values of c imply more frequent compressions of the primary buffer A, with a negative impact on the running time. On the other hand, small values of c reduce the maximum and average value of $\delta' = |C| + \delta$, hence making the searches on A faster. This behavior is visible in Figure 1(f), where we fixed $a = 32$ and $b = 1$, and considered different combinations of δ and c. In most cases, the best choice for c is 1.

We next focus on the case $(a, b, c) = (32, 1, 1)$. Figure 1(d) shows that that the space usage of RandMem(32,1,1) is essentially the same as RandMem(32) (hence, much better than both Rand and Det). Figure 1(g) shows that RandMem(32,1,1) is much faster than Rand for small values of δ, and slightly slower for large values of δ. The improvement of the performance for small values of δ is due to the use of a larger primary buffer, as mentioned in previous section. The fact that RandMem(32,1,1) becomes slower than Rand for large values of δ is not surprising, since the first data structure is more complicated (in order to save space). RandMem(32,1,1) is much faster than Det unless δ is very small.

Non-Random Data Sets. We tested resilient dictionaries also on non-random data sets, observing the same qualitative phenomena as with random instances. In particular, we considered the following experiment. First, we bulk-load the resilient dictionary considered with all the words of one English dictionary, and then we search for all the words in one English book. Figure 1(h) shows the average time per `search` of Rand, Det and `RandMem(32,1,1)`, when searching for the words in "The Picture of Dorian Gray". The high-level behavior of the running time is the same as with random instances. The smaller running time with respect to Figure 1(g) is due to the fact that in this experiment we considered `search` operations only: these operations turn out to be faster than `insert` and `delete` operations (which contribute to the average running time in Figure 1(g)).

The above results suggest that `RandMem` is probably the data structure of choice for practical applications.

References

1. J. A. Aslam, A. Dhagat. Searching in the presence of linearly bounded errors. *STOC*, 486–493, 1991.
2. S. Assaf, E. Upfal. Fault-tolerant sorting networks. *SIAM J. Discrete Math.*, 4(4), 472–480, 1991.
3. Y. Aumann, M. A. Bender. Fault-tolerant data structures. *FOCS*, 580–589, 1996.
4. R. S. Borgstrom, S. Rao Kosaraju. Comparison based search in the presence of errors. *STOC*, 130–136, 1993.
5. R. Boyer, S. Moore. MJRTY - A fast majority vote algorithm. University of Texas Tech. Report, 1982.
6. G. S. Brodal, R. Fagerberg, I. Finocchi, F. Grandoni, G. F. Italiano, A. G. Jorgensen, G. Moruz, T. Molhave. Optimal Resilient Dynamic Dictionaries. *ESA*, 347–358, 2007.
7. B. S. Chlebus, A. Gambin, P. Indyk. Shared-memory simulations on a faulty-memory DMM. *ICALP*, 586–597, 1996.
8. U. Feige, P. Raghavan, D. Peleg, E. Upfal. Computing with noisy information. *SIAM J. Comput.*, 23, 1001–1018, 1994.
9. U. Ferraro-Petrillo, I. Finocchi, G. F. Italiano. The Price of Resiliency: a Case Study on Sorting with Memory Faults. *Algorithmica*, 53(4):597–620, 2009.
10. I. Finocchi, F. Grandoni, G. F. Italiano. Optimal sorting and searching in the presence of memory faults. *Theor. Comput. Sci.* 410(44):4457-4470, 2009.
11. I. Finocchi, F. Grandoni, G. F. Italiano. Resilient search trees. *SODA*, 547–553, 2007.
12. I. Finocchi, F. Grandoni, G. F. Italiano. Designing reliable algorithms in unreliable memories. *Computer Science Review*, 1(2), 77–87, 2007.
13. I. Finocchi, G. F. Italiano. Sorting and searching in faulty memories. *Algorithmica*, 52(3):309-332, 2008.
14. S. Hamdioui, Z. Al-Ars, J. V. de Goor, M. Rodgers. Dynamic faults in random-access-memories: Concept, faults models and tests. *J. of Electronic Testing: Theory and Applications*, 19:195–205, 2003.
15. M. R. Henzinger. The past, present and future of Web Search Engines. Invited talk. *ICALP*, 2004.
16. M. R. Henzinger. Combinatorial algorithms for web search engines - three success stories. Invited talk. *SODA*, 2007.
17. D. J. Kleitman, A. R. Meyer, R. L. Rivest, J. Spencer, K. Winklmann. Coping with errors in binary search procedures. *J. Comp. Syst. Sci.*, 20:396–404, 1980.
18. T. Leighton, Y. Ma. Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. *SIAM J. Comput.*, 29(1):258–273, 1999.
19. T. C. May, M. H. Woods. Alpha-Particle-Induced Soft Errors In Dynamic Memories. *IEEE Trans. Elect. Dev.*, 26(2), 1979.
20. A. Pelc. Searching games with errors: Fifty years of coping with liars. *Theoret. Comp. Sci.*, 270, 71–109, 2002.
21. A. C. Yao, F. F. Yao. On fault-tolerant networks for sorting. *SIAM J. on Comput.*, 14, 120–128, 1985.

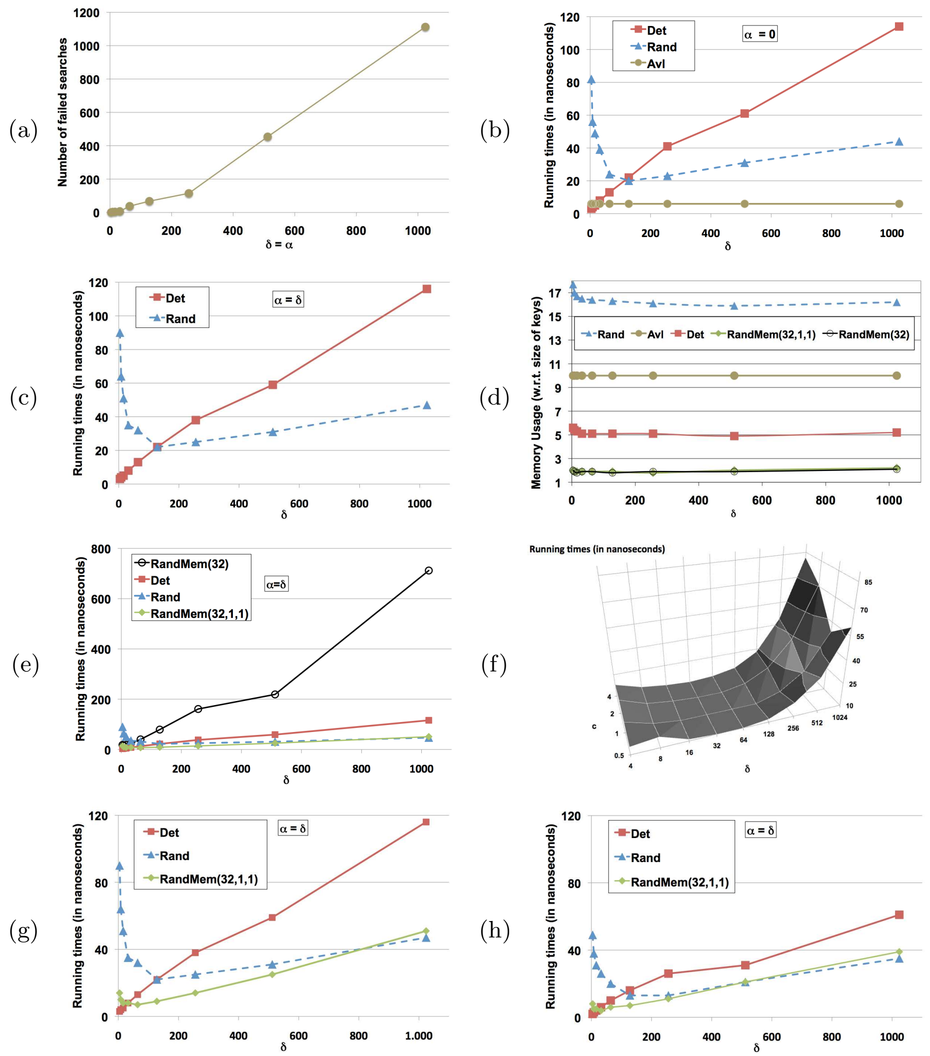

Fig. 1. We use $\alpha = 0$ in (b) and (d), and $\alpha = \delta$ in the other cases. **(a)** Number of failed `search` operations of `Avl` when processing a sequence of 10^5 random searches on a dictionary containing 10^6 random keys. **(b)+(c)** Average running time per operation of `Rand`, `Det` and `Avl`, when processing a sequence of 10^5 random operations on a dictionary initially containing 10^6 random keys. **(d)+(e)** Average memory usage, as multiples of the total size of the keys, and running time per operation of `Rand`, `Det` and `Avl`, `RandMem(32)` and `RandMem(32,1,1)`, when processing a sequence of 10^5 random operations on a dictionary initially containing 10^6 random keys. **(f)** Average running time per operation of `RandMem`(32,1,c), for different combinations of c, when processing a sequence of 10^5 random operations on a dictionary initially containing 10^6 random keys. **(g)+(h)** Average running time per operation of `Rand`, `Det` and `RandMem(32,1,1)`, when processing a sequence of 10^5 random operations on a dictionary initially containing 10^6 random keys, and when searching for all the 81965 words in "The Picture of Dorian Gray" on a dictionary initially containing 234936 distinct English words.