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

UMBERTO FERRARO-PETRILLO, Università di Roma - LA Sapienza
FABRIZIO GRANDONI, Dalle Molle Institute
GIUSEPPE F. ITALIANO, Università di Roma - Tor Vergata

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.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems–Sorting and searching

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: sorting, memory faults, memory models, fault injection, computing with unreliable information, experimental algorithmics

## 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 [Hamdioui et al. 2003; May and Woods 1979; Tezzaron Semiconductor 2004]. 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 [Henzinger 2004; 2007].

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

Author's addresses: U. Ferraro-Petrillo (umberto.ferraro@uniroma1.it), Dipartimento di Scienze Statistiche, Sapienza Università di Roma, P.le Aldo Moro 5, 00185 Rome, Italy ; F. Grandoni (fabrizio@idsia.ch), Dalle Molle Institute (IDSIA) Galleria 1, 6928, Manno, Switzerland; G. F. Italiano (italiano@disp.uniroma2.it), Dipartimento di Ingegneria Civile e Ingegneria Informatica, Università di Roma "Tor Vergata" via del Politecnico 1, 00133 Roma, Italy

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

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 [Aslam and Dhagat 1991; Borgstrom and Kosaraju 1993; Feige et al. 1994; Rivest et al. 1978; Pelc 2002], fault-tolerant sorting networks [Assaf and Upfal 1991; Leighton and Ma 1999; Yao and Yao 1985], resiliency of pointer-based data structures [Aumann and Bender 1996], and parallel models of computation with faulty memories [Chlebus et al. 1996].

### 1.1. The Faulty-RAM Model

In this paper we focus on the *faulty-RAM* model introduced in [Finocchi et al. 2005; Finocchi and Italiano 2008]. 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 $\delta$ 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.

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 [Boyer and Moore 1982]). 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 $\delta$, 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 [Finocchi et al. 2009; Finocchi and Italiano 2008], the *resilient sorting* problem is considered. Here, we are given a set $K$ of $n$ keys. A key is a finite (possibly negative) real value. We call a key *faithful* if it is never corrupted, and *faulty* otherwise. The problem is to compute a *faithfully 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 [Finocchi and Italiano 2008], 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 [Finocchi et al. 2009]. In the special case of polynomially-bounded integer keys, an improved running time of $O(n + \delta^2)$ can be achieved [Finocchi et al. 2009]. In [Ferraro-Petrillo et al. 2009] 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 [Finocchi et al. 2009; Finocchi and Italiano 2008]. Here we are given a faithfully sorted sequence $K$ of $n$ keys, and a search key $\kappa$. The problem is to return a key (faulty or faithful) of value $\kappa$, if $K$ contains a faithful key of that value. If there is no faithful key equal to $\kappa$, one can either return *no* or return a (faulty) key equal to $\kappa$. Note that, also in this case, this is the best possible: the adversary may indeed introduce a corrupted key equal to $\kappa$ 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 $\kappa$ (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 [Finocchi and Italiano 2008] for deterministic algorithms, and later extended to randomized algorithms in [Finocchi et al. 2009]. A $O(\log n + \delta^2)$ deterministic algorithm is given in [Finocchi and Italiano 2008]. An optimal $O(\log n + \delta)$ randomized algorithm `ResSearch` is provided in [Finocchi et al. 2009]. An optimal $O(\log n + \delta)$ deterministic algorithm is eventually given in [Brodal et al. 2007]. For both resilient sorting and searching, the space usage is $O(n)$.

## 1.2. 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 [Finocchi et al. 2007], Finocchi et al. present a resilient dictionary using $O(\log n+\delta^2)$ amortized time per operation. In [Brodal et al. 2007], 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 $2\delta$, which contains between $\delta/2$ and $2\delta$ keys at any time. Intervals are stored in a classical AVL tree, where standard variables (pointers, interval boundaries, etc.) are replaced by resilient variables. 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 $2\delta$ 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.

### 1.3. 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 developed a testbed to simulate the faulty-RAM model. We model the data structure and the adversary as two separate parallel threads. The data structure thread simply runs the data structure considered on the input sequence of operations. We keep track of all the unsafe memory words used by the data structure. 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 $\alpha$ faults, the adversary samples $\alpha$ operations uniformly at random over a given sequence of operations, and injects exactly one random fault during each sampled operation. We remark that this type of adversary is much less powerful than the adaptive adversary considered in the faulty-RAM (which seems very hard to implement, and probably too pessimistic for practical applications). However, our implementations guarantee the theoretical asymptotic bounds with respect to the adaptive adversary.

We performed experiments both on random inputs and on real-world inputs. In random inputs, the instances 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 $\kappa$, where, with probability $1/2$, $\kappa$ 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 $\kappa$, where $\kappa$ 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.

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` compiler 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/`.

Random values are produced by the `random()` pseudo-random source of numbers provided by the ANSI C standard library. The running time of each experiment is measured by means of the standard system call `getrusage()`. The resilient dictionary and the adversary run as two different parallel processes on our biprocessor architecture: concurrent accesses to shared memory locations (e.g., the corruption of a key in unreliable memory) are solved at the hardware level by spending only a few CPU cycles. This allows us to get a confident measure of the algorithms' running time, without taking into account also the time spent for injecting faults.

The rest of this paper is organized as follows. In Section 2 we illustrate our experimental evaluation of the time and space performance of `Rand` and `Det`, i.e., the optimal resilient dictionaries known in the literature. Furthermore, we compare it with the performance of a standard (non-resilient) dictionary `Avl` (see Section 2.2). We also describe our experimental findings about the sensitivity of `Avl` to faults (see Section 2.1). The main insight of our experiments is that all the mentioned data structures (including `Avl`) have a very large overhead in terms of space, which might not be acceptable in the applications considered (involving huge data sets). Motivated by that, we developed a variant `RandMem` of `Rand`, which, according to our experiments, is as fast as `Rand` and `Det`, and requires much less space. The implementation and performance of `RandMem` is discussed in Section 3. We conclude the paper with a summary of our experimental findings (see Section 4).

## 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 $\delta$ includes both cases where the running time is dominated by the $O(\log n)$ term and cases where the $O(\delta)$ term dominates.

### 2.1. 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 $\delta$. 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. 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., $\delta$ is roughly $0.1\%$ of the number of keys), roughly $1100$ `search` operations fail (i.e., roughly $1\%$ of the operations). Hence, resilient implementations seem to be really worth the effort.

### 2.2. 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 $\alpha$ (besides to $\delta$), we considered different values of the ratio $\alpha/\delta$: 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

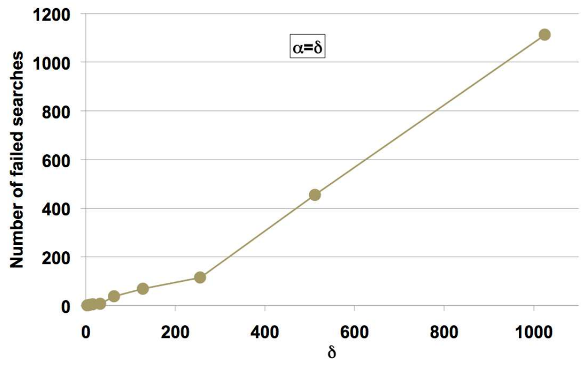


Fig. 1. Number of failed `search` operations of `Avl` when processing a sequence of $10^5$ random searches on a dictionary containing $10^6$ random keys. In this experiment $\alpha = \delta$.

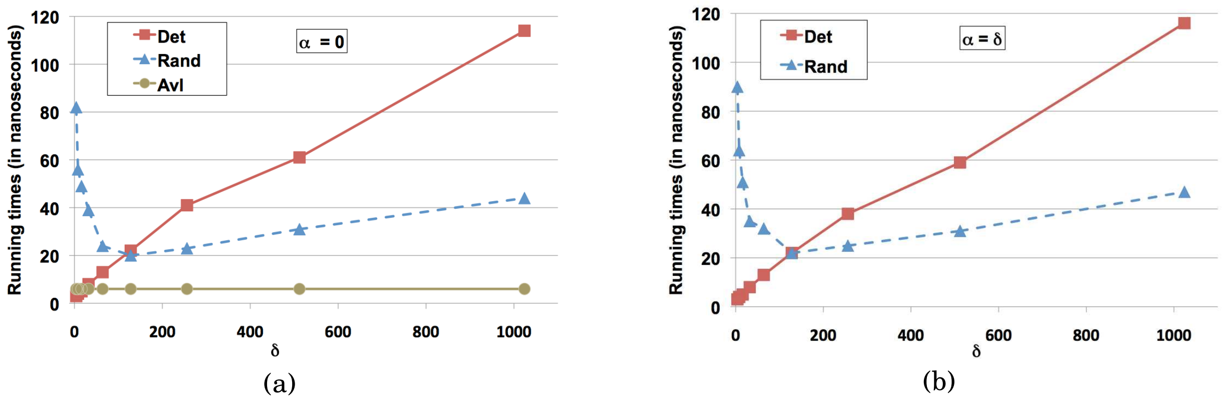


Fig. 2. Average running time per operation of `Rand`, `Det` and `Avl`, when processing a sequence of $10^5$ random operations on a dictionary initially containing $10^6$ random keys. The value of $\alpha$ is $0$ and $\delta$ in (a) and (b), respectively.

$10^6$ `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 $\alpha$ 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.

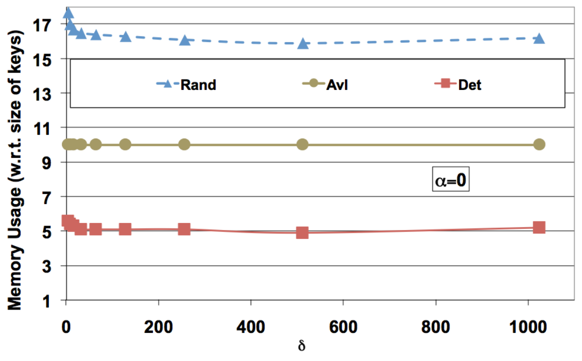
The results concerning the running time for the above instance are shown in Figure 2. As expected, `Avl` is much faster than the resilient dictionaries. Interestingly enough, the time performance of `Rand` and `Det` is very sensitive to $\delta$, but it is almost not affected by $\alpha$. This suggests that, in any practical implementation, $\delta$ should be chosen very carefully: underestimating $\delta$ 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 $\delta$, but it is much slower than `Det` when $\delta$ is small. Not surprisingly, the running time of `Det` grows with $\delta$. More interestingly, the running time of `Rand` initially quickly decreases with $\delta$ and then slowly increases. This behavior arises from the fact that, for small values of $\delta$, `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 $\delta$.

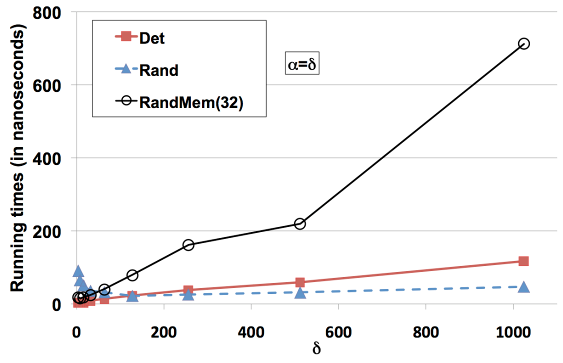
The results concerning the space usage for the mentioned case are shown in Figure 3(a). The space usage in the figure are given as multiples of the total space occupied by keys (which is a lower bound on the space needed). We first of all observe that the space usage of `Rand` and `Det` is almost not affected by $\delta$ (this is obvious in the case of `Avl`). In particular, the space usage initially decreases with $\delta$, 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 $\delta$. The experiments show that even the constants in the asymptotic notation are weakly related to $\delta$.

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.



(a)



(b)

Fig. 3. 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. The value of $\alpha$ is 0 and $\delta$ in (a) and (b), respectively.

## 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 [Finocchi et al. 2009; Finocchi and Italiano 2008], 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 3(a) we report on the space usage of `RandMem`$(32)$ on the same input instances used in Section 2.2. 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, with 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 $\infty$, which stands for a value larger than any feasible key. Note that the adversary is allowed to write $\infty$ 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 3(b), where we report on the running time of `RandMem`$(32)$ for the same input instances as in Section 2.2.

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 [Finocchi et al. 2009]. 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 [Finocchi et al. 2009].

**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 $\infty$. 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 [Finocchi et al. 2009]). 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 $\infty$ (recall that empty positions are set to $\infty$ as well). When $A$ and $B$ are merged after one `insert`, we can easily get rid of this extra values $\infty$. Note that the $\infty$ 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 $\tau$, we *compress* $A$ so that the values different from $\infty$ occupy the first positions in the buffer (while the final $\infty$ 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 $\tau$ (this costs roughly $4\delta$ 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 $\tau$, 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 $\alpha_C$ and $\alpha_A$ 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

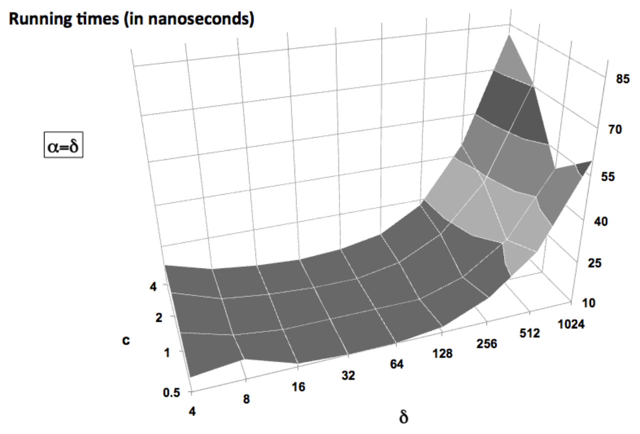


Fig. 4. Average running time per operation of RandMem(32,1,$c$), for different combinations of $c$ and $\alpha = \delta$, when processing a sequence of $10^5$ random operations on a dictionary initially containing $10^6$ random keys.

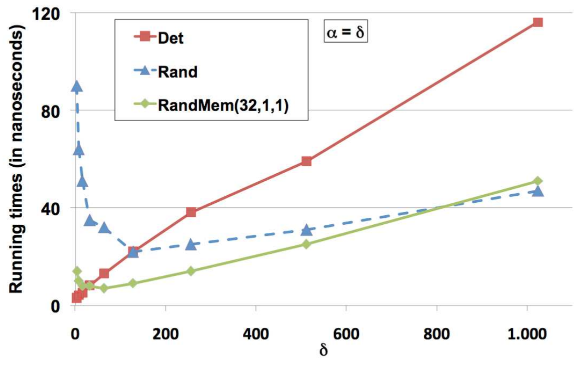
faults are needed to force a compression of cost $O(a\delta)$. Hence, this type of phenomenon does not affect the running time substantially. We remark that unary counters were not used before in the literature on reliable algorithms and data structures.

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.

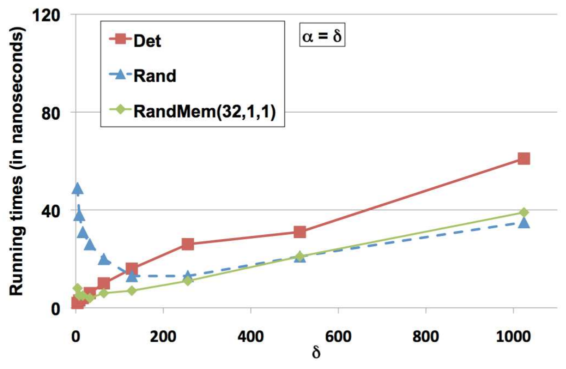
**Experimental Evaluation**. We tested RandMem($a$,$b$,$c$) in different scenarios and for different values of the triple $(a, b, c)$. We next restrict our attention to the input instances as considered in Section 2.2, with $\alpha = \delta$. Furthermore, we assume $a = 32$, which minimizes the space 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 4, where we fixed $a = 32$ and $b = 1$, and considered different combinations of $\delta$ 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 3(a) shows 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 5(a) shows that RandMem(32,1,1) is much faster than Rand for small values of $\delta$, and slightly slower for large values of $\delta$. The improvement of the performance for small values of $\delta$ is due to the use of a larger primary buffer, as mentioned in the previous section. The fact that RandMem(32,1,1) becomes slower than Rand for large values of $\delta$ 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 $\delta$ is very small.



(a)



(b)

Fig. 5. Average running time per operation of `Rand`, `Det` and `RandMem(32,1,1)`, when (a) processing a sequence of $10^5$ random operations on a dictionary initially containing $10^6$ random keys, and (b) searching for all the $81965$ words in "The Picture of Dorian Gray" on a dictionary initially containing $234936$ distinct English words. In these tests, $\alpha = \delta$.

**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 5(b) 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 5(a) 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 5(a)).

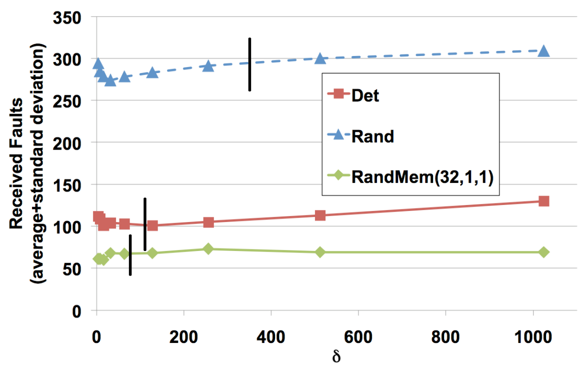
**Experimenting with realistic faults**. A frequent use-case for dictionaries is as resident data structures running on a server machine, and used to store and to provide a fast access to large sets of keys. In such a scenario, it is quite common that the dictionaries stay resident in memory for a long period of time and, therefore, are more subject to memory faults. Indeed, it is unlikely that a fault will occur, in the real life, during the execution of an operation on a dictionary, as this usually takes few microseconds on the average. Instead, it is much more likely that a dictionary used on a server will experience several faults during its entire lifespan. For this reason, we introduce a new experimentation model where we do not take into account just the time needed to perform a batch of operations but, instead, we consider the whole lifecycle of the dictionary.

From the experimental point of view, we simulated this scenario by assuming that, after being initialized, the dictionary stays resident in memory for a large number of (simulated) days. During each day, the dictionary is used to perform a certain number of operations (i.e., insertions, deletions and searches) and may experience faults, with a probability that is proportional to the amount of memory it uses. The memory locations being hit by a fault remain corrupted, until they are explicitly overwritten by the algorithm.

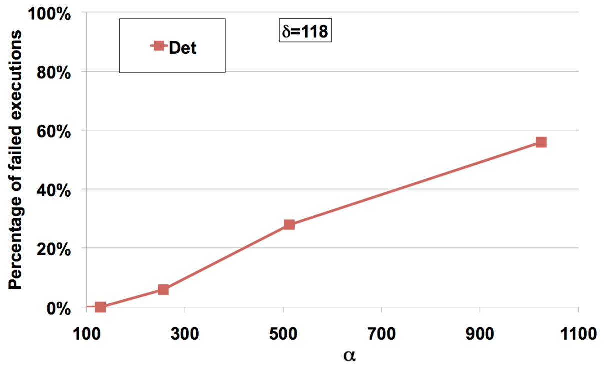
This model allows us to perform a better estimation of the values of $\delta$ which provide an optimal trade-off about resiliency and performance. In our case, we carried out this estimation by examining the behavior of different dictionaries during a simulated year of operation. In details, we populated the dictionaries via a sequence of $10^6$ `random` `insert` operations, then we executed, for each day of the year, $3.000$ random operations. Moreover, we configured our framework to inject faults with a FIT rate (i.e., average number of failures per Megabit and per billion device hours) of $50.000$: this is a good approximation of the average failure rate for standard DIMM memories, as described in [Schroeder et al. 2009].

In Figure 6(a) we report the results of an experiment where we measured the average number of faults hitting the memory used by the different dictionaries, when tested with increasing values of $\delta$. As it can be easily seen, `RandMem(32,1,1)` experiences a limited number of faults: this is a direct consequence of the small memory footprint of this algorithm. Instead, we observe that the high memory demand of `Rand` implies an higher number of faults. This difference could have a strong impact on the performance of the two dictionaries, as `Rand` will require a value of $\delta$ much higher than the one needed by `RandMem` in order to preserve its resiliency. We emphasize this difference in the figure by marking with a black line the points where, for each tested dictionary, the number of faults is equal to $\delta$.

Another outcome of this experiment is that `Det` used to crash very often when the number of faults was consistently higher than $\delta$. This is shown in Figure 6(b), where we describe the percentage of crashes experienced by `Det` in the previous experiment. These crashes are due to the strategy used by `Det` to deal with replicated variables. At the beginning of the execution, `Det` uses only the first copy of each replicated variable. If a fault is detected, then the algorithm moves to the second copy, and so on. If the number of detected faults exceeds $\delta$, the algorithm goes in an inconsistent state and terminates. This problem does not affect `Rand` and `RandMem` because these two algorithms always choose at random which copy of a replicated variable to read, during each access and independently of the faults detected so far. The only case where they could crash is when at least $\delta + 1$ copies of a replicated variable are corrupted, a case that is unlikely to happen in a model where faults are generated uniformly at random.



(a)



(b)

Fig. 6. (a) Average number of faults suffered by `Rand`, `Det` and `RandMem(32,1,1)`, when processing a sequence of $1.095.000$ random operations on a dictionary initially containing $10^6$ random keys on a lifespan of a (simulated) year, with $FIT = 50.000$. The black tags mark the points where the number of faults is equal to $\delta$. (b) Percentage of crashes experienced by `Det` when processing a sequence of $1.095.000$ random operations on a dictionary initially containing $10^6$ random keys on a lifespan of a (simulated) year, with $FIT = 50.000$.

## 4. CONCLUSIONS

The main findings of our experimental study are the following:

- `Avl`, a standard (non-resilient) implementation of an AVL binary search tree, is rather sensitive to memory faults. Due to pointer corruptions, a large fraction of the keys may be lost even with very few faults. This suggests that using resilient dictionaries rather than standard dictionaries can be really worth the effort.
- In turn, `Avl` is much faster than the optimal resilient dictionaries `Det` and `Rand`. This suggests that resilient implementations should be used only when the risk of memory faults is concrete.
- The running time of `Rand` and `Det` is rather sensitive to the upper bound $\delta$ on the number of memory faults, but not much to the actual number of faults $\alpha$. This is an indication that $\delta$ has to be fixed very carefully: underestimating $\delta$ compromises the resiliency while overestimating it might increase the running time.
- The space usage of `Det` and `Rand` is not substantially affected by $\delta$. For both data structures, the space usage is much larger than the lower bound given by the total size of the keys. This might not be acceptable in practical applications, where keys occupy terabytes of space. We also observed that `Avl`, which is even non-resilient, has a similar drawback.
- Our improved dictionary `RandMem` uses an amount of memory closer to the lower bound given by key storage, and has a running time close to (sometimes better than) `Rand` and `Det`. This makes `RandMem` probably the data structure of choice for practical applications.

## References

ASLAM, J. A. AND DHAGAT, A. 1991. Searching in the presence of linearly bounded errors. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. STOC '91. ACM, New York, NY, USA, 486–493.

ASSAF, S. AND UPFAL, E. 1991. Fault tolerant sorting networks. *SIAM J. Discret. Math. 4,* 4, 472–480.

AUMANN, Y. AND BENDER, M. A. 1996. Fault tolerant data structures. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*. FOCS '96. IEEE Computer Society, Washington, DC, USA, 580–.

BORGSTROM, R. S. AND KOSARAJU, S. R. 1993. Comparison-based search in the presence of errors. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. STOC '93. ACM, New York, NY, USA, 130–136.

BOYER, R. S. AND MOORE, J. S. 1982. MJRTY - a fast majority vote algorithm.

BRODAL, G. S., FAGERBERG, R., FINOCCHI, I., GRANDONI, F., ITALIANO, G. F., JØRGENSEN, A. G., MORUZ, G., AND MØLHAVE, T. 2007. Optimal resilient dynamic dictionaries. In *Proceedings of the 15th annual European conference on Algorithms*. ESA'07. Springer-Verlag, Berlin, Heidelberg, 347–358.

CHLEBUS, B. S., GAMBIN, A., AND INDYK, P. 1996. Shared-memory simulations on a faulty-memory dmm. In *Proceedings of the 23rd International Colloquium on Automata, Languages and Programming*. ICALP '96. Springer-Verlag, London, UK, UK, 586–597.

FEIGE, U., RAGHAVAN, P., PELEG, D., AND UPFAL, E. 1994. Computing with noisy information. *SIAM J. Comput. 23,* 5, 1001–1018.

FERRARO-PETRILLO, U., FINOCCHI, I., AND ITALIANO, G. F. 2009. The price of resiliency: a case study on sorting with memory faults. *Algorithmica 53,* 4, 597–620.

FINOCCHI, I., GRANDONI, F., AND ITALIANO, G. F. 2005. Designing reliable algorithms in unreliable memories. In *Proceedings of the 13th annual European conference on Algorithms*. ESA'05. Springer-Verlag, Berlin, Heidelberg, 1–8.

FINOCCHI, I., GRANDONI, F., AND ITALIANO, G. F. 2007. Resilient search trees. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. SODA '07. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 547–553.

FINOCCHI, I., GRANDONI, F., AND ITALIANO, G. F. 2009. Optimal resilient sorting and searching in the presence of memory faults. *Theor. Comput. Sci. 410,* 44, 4457–4470.

FINOCCHI, I. AND ITALIANO, G. F. 2008. Sorting and searching in faulty memories. *Algorithmica 52,* 3, 309–332.

HAMDIOUI, S., AL-ARS, Z., VAN DE GOOR, A. J., AND RODGERS, M. 2003. Dynamic faults in random-access-memories: Concept, fault models and tests. *J. Electron. Test. 19,* 2, 195–205.

HENZINGER, M. 2004. The past, present and future of web search engines (invited talk). In *Automata, Languages and Programming: 31st International Colloquium, ICALP 2004, Turku, Finland, July 12-16, 2004. Proceedings*. Lecture Notes in Computer Science Series, vol. 3142. Springer.

HENZINGER, M. 2007. Combinatorial algorithms for web search engines: three success stories. 1022–1026.

LEIGHTON, T. AND MA, Y. 1999. Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. *SIAM J. Comput. 29,* 1, 258–273.

MAY, T. AND WOODS, M. 1979. Alpha-particle-induced soft errors in dynamic memories. *Electron Devices, IEEE Transactions on 26,* 1, 2 – 9.

PELC, A. 2002. Searching games with errors—fifty years of coping with liars. *Theor. Comput. Sci. 270,* 1-2, 71–109.

RIVEST, R. L., MEYER, A. R., AND KLEITMAN, D. J. 1978. Coping with errors in binary search procedures (preliminary report). In *Proceedings of the tenth annual ACM symposium on Theory of computing*. STOC '78. ACM, New York, NY, USA, 227–232.

SCHROEDER, B., PINHEIRO, E., AND WEBER, W.-D. 2009. Dram errors in the wild: a large-scale field study. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. SIGMETRICS '09. ACM, New York, NY, USA, 193–204.

TEZZARON SEMICONDUCTOR. 2004. Soft Errors in Electronic Memory - A White Paper. Tech. rep. Jan.

YAO, A. AND YAO, F. 1985. On fault-tolerant networks for sorting. *SIAM Journal on Computing 14,* 1, 120–128.