# The Price of Resiliency: A Case Study on Sorting with Memory Faults*

Umberto Ferraro-Petrillo † Irene Finocchi ‡ Giuseppe F. Italiano §

**Abstract**

We address the problem of sorting in the presence of faults that may arbitrarily corrupt memory locations, and investigate the impact of memory faults both on the correctness and on the running times of mergesort-based algorithms. To achieve this goal, we develop a software testbed that simulates different fault injection strategies, and perform a thorough experimental study using a combination of several fault parameters. Our experiments give evidence that simple-minded approaches to this problem are largely impractical, while the design of more sophisticated resilient algorithms seems really worth the effort. Another contribution of our computational study is a carefully engineered implementation of a resilient sorting algorithm, which appears robust to different memory fault patterns.

**Keywords:** sorting, memory faults, memory models, fault injection, computing with unreliable information, experimental algorithmics.

## 1 Introduction

A standard assumption in the design and analysis of algorithms is that the contents of memory locations do not change throughout the algorithm execution unless they are explicitly written by the algorithm itself. This assumption, however, may not necessarily hold for very large and inexpensive memories used in modern computing platforms. The trend observed in the design of today's memory technologies, in fact, is to avoid the use of sophisticated error checking and correction circuitry that would impose non-negligible costs in terms of both performance and money: as a consequence, memories may be quite error-prone and can be responsible of silent data corruptions. Memory failures are usually classified as either hard errors (i.e., permanent physical defects whose repair requires component replacement), or soft errors (i.e., transient faults in semiconductor devices and recoverable errors in disks and other devices) [32, 42]. Soft errors are a particularly big concern in the reliability of storage systems

*This work has been partially supported by the Sixth Framework Programme of the EU under Contract Number 507613 (Network of Excellence "EuroNGI: Designing and Engineering of the Next Generation Internet") and by MIUR, the Italian Ministry of Education, University and Research, under Project MAINSTREAM: ("Algorithms for Massive Information Structures and Data Streams"). A preliminary version of this work was presented at the *14th Annual European Symposium on Algorithms* (ESA'06).

†Dipartimento di Statistica, Probabilità e Statistiche Applicate, Università di Roma "La Sapienza", P.le Aldo Moro 5, 00185 Rome, Italy. Email: umberto.ferraro@uniroma1.it.

‡Dipartimento di Informatica, Università di Roma "La Sapienza", Via Salaria 113, 00198, Roma, Italy. Email: finocchi@di.uniroma1.it.

§Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Via del Politecnico 1, 00133 Roma, Italy. Email: italiano@disp.uniroma2.it.

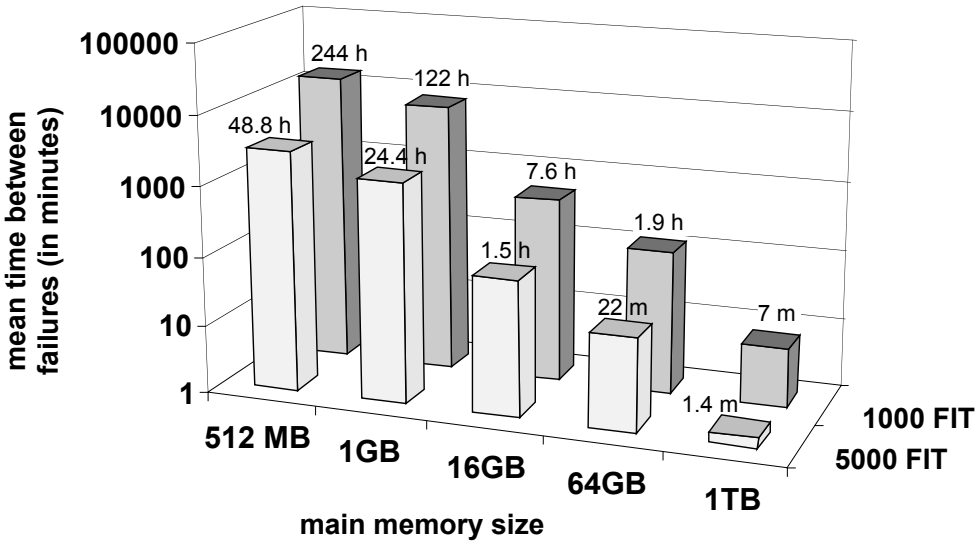



Figure 1: Mean time between failures of different memories as a function of the memory size and of the number of failures in time per Mbit (FIT).

as recently discussed, e.g., in [27, 31, 36, 38, 39, 41]. Such errors have several origins and exhibit complicated patterns. Hardware or power failures, as well as environmental conditions such as cosmic rays and alpha particles, can temporarily affect the behavior of semiconductor devices resulting in unpredictable, random, independent bit flips. According to memory vendors (see, e.g., [41]), the standard soft error rates at sea level for modern memory devices are between 1000 and 5000 failures in time per Mbit (in short, FIT): this means that a memory chip of 1 Mbit is likely to experience between 1000 and 5000 faults every $10^9$ hours of use. We remark that the intensity of cosmic radiation depends strongly on altitude, and thus the failure rate considerably increases with altitude, being even two orders of magnitude larger at 10000 meters [27]. Figure 1 illustrates the mean time between failures for computing platforms with different memory sizes using currently available technologies (at sea level). The mean time between failures has been obtained as $10^9/(\text{FIT} \times \text{memory size in Mbit})$ and gives the expected number of hours between two faults as a function of the memory size and of the FIT rate. As shown by the chart, this number decreases as the memory size becomes larger: for instance, a system with Terabytes of memory, such as a large cluster of computing platforms with a few Gigabytes per node, is likely to experience one bit flip every few minutes. Multiple bit errors can also occur at different levels of the memory hierarchy. For instance, a systematic analysis of data corruption recently conducted in the CERN computer center [33] has shown that disks can experience transient errors due to environmental factors, such as high temperature, and to usage activities, and that a single error can often determine the corruption of large regions of data, up to 64 KB. This can be a serious problem for algorithms that work by copying data from disk to main memory, as happens in all those applications designed to cope with large data sets.

In the design of reliable systems, when specific hardware for fault detection is not available, it makes sense to assume that the algorithms themselves are in charge of dealing with memory faults. Informally, we say that an algorithm is *resilient to memory faults* if, despite the corruption of some memory values before or during its execution, the algorithm is nev-

ertheless able to get a correct output at least on the set of uncorrupted values. We note that classical algorithms are typically non-resilient: if an algorithm is not prepared to cope with memory faults, it may take wrong steps upon reading corrupted values and errors may propagate over throughout its execution. Designing resilient algorithms seems important in a variety of different domains. Many large-scale applications that require the processing of massive data sets demand large memory capacities at low cost: in these cases even very few memory faults may jeopardize the correctness of the underlying algorithms. For instance, the inverted indices used by Web search engines are typically maintained sorted for fast document access: for such large data structures, even a small failure probability can result in bit flips in the index, that may become responsible of erroneous answers to keyword searches. Similar phenomena have been observed in practice [23]. Another application domain is related to avionics systems, which make typically use of embedded software with strong reliability concerns: as observed above, in this case memories may incur rather large numbers of faults, that can seriously compromise safety critical applications. Finally, in fault-based cryptanalysis some optical and electromagnetic perturbation attacks [7, 40] work by manipulating the non-volatile memories of cryptographic devices, so as to induce very timing-precise controlled faults on given individual bits: this forces the devices to output wrong ciphertexts that may allow the attacker to determine the secret keys used during the encryption. Induced memory errors have been effectively used in order to break cryptographic protocols [7, 8, 44], smart cards and other security processors [1, 2, 40], and to take control over a Java Virtual Machine [22]. In this context the errors are introduced by a malicious adversary, which can assume some knowledge of the algorithm's behavior. We remark that, in all the above applications, both the faults rate and the faults patterns (i.e., random or adversarial faults, single or multiple corruptions) can be very different.

**Related work.** Since the pioneering work of von Neumann in the late 50's [43], the problem of computing with unreliable information has been investigated in a variety of different settings, including the liar model [3, 9, 14, 16, 26, 28, 34, 37], fault-tolerant sorting networks [4, 29, 30, 45], resiliency of pointer-based data structures [5], parallel models of computation with faulty memories [11, 12, 24]. We refer the interested reader to [19] and [35] for an overview. In [18], Finocchi and Italiano introduced a faulty-memory random access machine, i.e., a random access machine whose memory locations may suffer from memory faults. In this model, an adaptive adversary may corrupt up to $\delta$ memory words throughout the execution of an algorithm. We remark that $\delta$ is not a constant, but a parameter of the model. The adaptive adversary captures situations like cosmic-rays bursts, memories with non-uniform fault-probability, and hackers' attacks which would be difficult to be modelled otherwise. The algorithm cannot distinguish corrupted values from correct ones and can exploit only $O(1)$ safe memory words, whose content never gets corrupted. The last assumption is not very restrictive, since one can usually afford the cost of storing a small number of running variables in a more expensive and more reliable memory of constant size. In [18, 20] Finocchi *et al.* presented matching upper and lower bounds for resilient sorting and searching in this faulty-memory model.

Let $n$ be the number of keys to be sorted. In [18] Finocchi and Italiano proved that any resilient $O(n \log n)$ comparison-based deterministic algorithm can tolerate the corruption of at most $O(\sqrt{n \log n}\,)$ keys. They also proved that one can sort resiliently in $O(n \log n + \delta^3)$ time: this yields an algorithm (Fast) whose running time is optimal in the comparison model as

long as $\delta = O((n \log n)^{1/3})$. In [20], Finocchi *et al.* closed the gap between the upper and the lower bound, designing a resilient sorting algorithm (Opt) with running time $O(n \log n + \delta^2)$. We note that both Fast and Opt pay only an additive overhead in their running times in order to cope with memory faults: this seems theoretically much better than a simple-minded approach, that would produce a multiplicative overhead of $\Theta(\delta)$ in the running times.

**Our results.** In this paper we perform a thorough experimental evaluation of the resilient sorting algorithms presented in [18, 20], along with a carefully engineered version of Opt (named Opt-Nb). In order to study the impact of memory faults on the correctness and the running time of sorting algorithms, we implemented a software testbed that simulates different fault injection strategies. Our testbed allows us to control the number of faults to be injected, the memory location to be altered, and the fault generation time. Since our study is not tied to a specific application domain, and since fault rate and fault patterns can be very different in different applications, we performed experiments using a variety of combinations of these parameters and different instance families. We tried to determine, for instance, for which values of $\delta$ a naive approach is preferable to more sophisticated algorithms, and, similarly, if there are any values of $\delta$ for which algorithm Fast is preferable to Opt in spite of the theoretical bounds on the running times.

As a first contribution, we show experimentally that even very few memory faults that hit random memory locations can make the sequence produced by a non-resilient sorting algorithm completely disordered: this stresses the need of taking care explicitly of memory faults in the algorithm implementation. We next evaluate the running time overhead of Fast, Opt, and Opt-Nb. Our main findings can be summarized as follows.

- A simple-minded approach to resiliency is largely impractical: it yields an algorithm (Naive) which may be up to hundreds of times slower than its non-resilient counterpart. We remark that Naive turned out to be very slow even for the smallest values of $\delta$ used in our experiments, i.e., $\delta = 1$ or 2.

- The design of more sophisticated resilient algorithms seems worth the effort: Fast, Opt and Opt-Nb are always much faster than Naive (even for small $\delta$) and get close to the running time of non-resilient sorting algorithms. In particular, Opt-Nb is typically at most 3 times slower than its non-resilient counterpart for the parameter settings considered in our experiments.

- Despite the theoretical bounds, Fast can be superior to Opt in case of a small number of faults: this suggests that Opt has larger implementation constants. However, unlike Opt, the performance of Fast degrades quickly as the number of faults becomes larger. In particular, the experiments suggest that for large values of $\delta$ the theoretical analyses of both Fast and Opt predict rather accurately their practical performances.

- The time interval in which faults happen may influence significantly the running times of Fast, while seems to have a negligible effect on the running times of Opt and Opt-Nb.

- Our engineered implementation Opt-Nb typically outperforms its competitors and seems to be the algorithm of choice for resilient sorting for moderate and large values of $\delta$. Fast may be still preferable when $\delta$ is rather small.

| Algorithms | Running times | References |
|---|---|---|
| NAIVE | $O(\delta\, n \log n)$ | |
| FAST | $O(n \log n + \alpha\delta^2)$ | [18] |
| OPT | $O(n \log n + \alpha\delta)$ | [20] |
| OPT-NB | $O(n \log n + \alpha\delta)$ | [17], this paper |

Table 1: Summary of the running times of the resilient algorithms under evaluation: $\delta$ and $\alpha$ denote the maximum and the actual number of memory faults, respectively.

All the algorithms under investigation make explicit use of an upper bound $\delta$ on the number of faults in order to guarantee correctness. Since it is not always possible to know in advance the number of memory faults that will occur during the algorithm execution, we analyzed the sensitivity of the algorithms with respect to variations of $\delta$, showing that rounding up $\delta$ (in absence of a good estimate) does not affect significantly the performances of OPT and OPT-NB even when $\delta$ is rather large. Finally, we considered a more realistic scenario, called *faults per unit time per unit memory model*, where algorithms with larger space consumption and larger execution times are likely to incur a larger number of memory faults. Even in this scenario, we observed the same relative performances of the algorithms: OPT and OPT-NB appear to be more robust than FAST and can tolerate higher fault rates. However, given an error rate $\sigma$ (per unit memory and per unit time), for all the resilient algorithms there exists an upper bound on the largest instance that can be faithfully sorted in the presence of faults occurring with rate $\sigma$.

**Organization of the paper.** The remainder of this paper is organized as follows. Section 2 describes the mergesort-based resilient algorithms under investigation and discusses some implementation issues. Section 3 presents our experimental framework, focusing on fault injection strategies and performance indicators. Our experimental findings are summarized in Section 4, and concluding remarks are listed in Section 5.

## 2 Resilient Sorting Algorithms

In this section we describe the mergesort-based resilient algorithms presented in [18, 20]. The worst-case running times of these algorithms are summarized in Table 1. We recall that $n$ is the number of keys to be sorted, $\delta$ is an upper bound on the total number of memory faults, and $\alpha$ is the actual number of faults that happen during a specific execution of a sorting algorithm. Throughout this paper, we will say that a key is *faithful* if its value is never corrupted by any memory fault, and faulty otherwise. A sequence is *$k$-unordered*, for some $k \geq 0$, if the removal of at most $k$ faithful keys yields a subsequence in which all the faithful keys are sorted (see Figure 2). According to this definition, a sequence in which only the corrupted keys appear in the wrong order is 0-*unordered*. The notion of $k$-unordered sequence has been derived from a classical measure of disorder (see, e.g., [15]) and can be used in order to characterize the correctness of sorting algorithms in the presence of memory faults. In particular, we will say that a sorting or merging algorithm is *resilient* to memory faults if it is able to produce a 0-unordered sequence.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 2 | 4 | 0 | 1 | 5 | 7 | 8 |

Figure 2: A sequence $X$, where white and gray locations indicate faithful and faulty values, respectively. Sequence $X$ is not faithfully ordered, but it is 2-unordered since the subsequence obtained by removing elements $X[4] = 0$ and $X[5] = 1$ is faithfully ordered.

### 2.1 Naive Resilient Sorting

A simple-minded resilient variant of standard merging takes the minimum among $(\delta+1)$ keys per sequence at each merge step, and thus considers at least one faithful key per sequence. By plugging this into mergesort, we obtain a resilient sorting algorithm, called NAIVE, which has a worst-case running time of $O(\delta\, n \log n)$. As noted in [18], whenever $\delta = \Omega(n^{\epsilon})$ for some constant $\epsilon > 0$, it can be shown that NAIVE runs faster, i.e., in $O(\delta\, n)$ time.

### 2.2 Two Basic Tasks: Merging and Purifying

Algorithms FAST and OPT reduce the time spent to cope with memory faults to an *additive* overhead (see Table 1). This is achieved by temporarily relaxing the requirement that the merging must produce a 0-unordered sequence and by allowing its output to be $k$-unordered, for some $k > 0$. We now describe the main subroutines used by algorithms FAST and OPT. In the analysis of each subroutine, we will use $\alpha$ to denote the number of faulty keys that appear out of order in the input sequences plus the number of faults introduced during the execution of the subroutine itself.

***Weakly-resilient merge*** [18] is an $O(n)$-time merging algorithm, which, although unable to produce a 0-unordered sequence, can guarantee that not too many faithful keys are out of place in the output sequence. It resembles classical merging, with the addition of suitable checks and error recovery. Checks are performed when the algorithm keeps on advancing in one of the two sequences for $(2\delta + 1)$ consecutive steps: if a check fails, a faulty key can be identified and removed from further consideration. Each check requires $\Theta(\delta)$ time, which can be amortized against the time spent to output the last $(2\delta + 1)$ keys. In [18] it is proved that each faulty key may prevent $O(\delta)$ faithful keys from being returned at the right time: this implies that the output sequence is $O(\alpha\delta)$-unordered.

***Purify*** [18] is a resilient variant of the Cook-Kim division algorithm [13]. Given a $k$-unordered sequence $X$ of length $n$, it computes a 0-unordered subsequence $S$ in $O(n + \delta \cdot (k + \alpha))$ worst-case time. It is guaranteed that the length of $S$ is at least $n - 2(k + \alpha)$, i.e., only $O(k + \alpha)$ keys are discarded in order to purify $X$.

***Purifying-merge*** [20] is a fast resilient merging algorithm that may nevertheless fail to merge all the input keys: the algorithm produces a 0-unordered sequence $Z$ and a disordered fail sequence $F$ in $O(n + \alpha\delta)$ worst-case time, where $|F| = O(\alpha)$, i.e., only $O(\alpha)$ keys can fail to get inserted into $Z$. This is an improvement over the weakly-resilient merge described above (obtained at a small price on the running time), and is achieved by a clever use of buffering techniques and more sophisticated consistency checks on data. The algorithm uses two auxiliary input buffers, in which keys to be merged are copied, and an auxiliary output buffer, from which merged keys are extracted. The merging process is divided into rounds of length $O(\delta)$. At each round the contents of the input buffers are merged until either an inconsistency in the input keys is found or the output buffer becomes full. In the former case

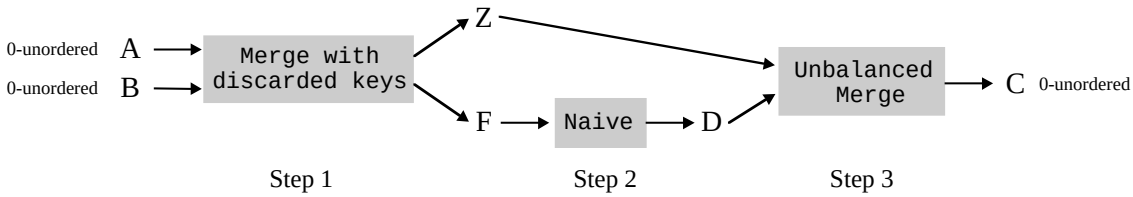



Figure 3: Skeleton of the resilient merging subroutine used by algorithms FAST [18] and OPT [20].

a purifying step is performed, moving two keys (one of which is faulty) to the fail sequence $F$. In the latter case the content of the output buffer is flushed to the output sequence $Z$. In both cases the input buffers are refilled with an appropriate number of keys and a new round is started. Note that, after a purifying step, the remaining keys must be compacted before starting a new round: thanks to the use of buffers, which have size $\Theta(\delta)$, this compaction can be done in $O(\delta)$ time. This guarantees that the total cost of purifying steps is only $O(\alpha\,\delta)$ [20].

***Unbalanced merge*** [18] requires superlinear time, but is particularly well suited at merging unbalanced sequences. It works by repeatedly extracting a key from the shorter sequence and placing it in the correct position with respect to the longer sequence: we need some care to identify this proper position, due to the appearance of memory faults. The algorithm runs in $O(n_1 + (n_2 + \alpha) \cdot \delta)$ time, where $n_1$ and $n_2$ denote the lengths of the sequences, with $n_2 \leq n_1$.

## 2.3 Fast Resilient Sorting

We now recall from [18, 20] how the subroutines described in Section 2.2 can be used to implement resilient sorting. Consider the merging algorithm represented in Figure 3. A first merging attempt on the input sequences $A$ and $B$ may fail to merge all the input keys, producing a 0-unordered sequence $Z$ and a disordered fail sequence $F$. The sequence $F$ is sorted using algorithm NAIVE: this produces another 0-unordered sequence $D$. The two 0-unordered unbalanced sequences, $Z$ and $D$, can be finally merged using unbalanced merging.

Step 1 (the first merging) can be implemented either by using the weakly-resilient merge and then purifying its output sequence, or by invoking directly the purifying-merge algorithm: in the former case $|F| = O(\alpha\delta)$ and the merge running time is $O(n + \alpha\delta^2)$, while in the latter case $|F| = O(\alpha)$ and the merge running time is $O(n + \alpha\delta)$. For both subroutines $\alpha$ is defined as in Section 2.2. A resilient sorting algorithm can be obtained by plugging these merging subroutines into mergesort: the two implementation choices for Step 1 yield algorithms FAST and OPT, respectively. We refer the interested reader to references [18, 20] for the low-level details and the analysis of the method.

## 2.4 Algorithm Implementation Issues

We implemented all the algorithms in `C++`, within the same algorithmic and implementation framework. Since recursion may not work properly in the presence of memory faults (the recursion stack may indeed get corrupted), we relied on a bottom-up iterative implementation of mergesort, which makes $\lceil \log_2 n \rceil$ passes over the array, where the $i$-th pass merges sorted subarrays of length $2^{i-1}$ into sorted subarrays of length $2^i$. For efficiency issues we applied the standard technique of alternating the merging process from one array to the other in order to

avoid unnecessary data copying. We also took care to use only $O(1)$ reliable memory words to maintain array indices, counters, and memory addresses.

In the implementation of algorithm Opt, again for efficiency reasons, we avoided allocating and deallocating its auxiliary buffers at each call of the merging subroutine. In spite of this, in a first set of experiments the buffer management overhead slowed down the execution of Opt considerably for some choices of the parameters: this depends on the fact that data need to be continuously copied to and from the auxiliary buffers during the merging process. Hence, we implemented and engineered a new version of Opt with the same asymptotic running time but which avoids completely the use of buffers: throughout this paper, we will refer to this implementation as Opt-Nb (i.e., Opt with No Buffering).

The algorithmic ideas behind Opt-Nb are exactly the same introduced for algorithm Opt. The only difference is related to low-level implementation details of `Purifying-merge`: the implementation of this subroutine used by Opt-Nb benefits from the same approach used by algorithm Fast, i.e., it avoids copying data to/from the auxiliary buffers by maintaining a constant number of suitable array indices and by working directly on the input sequences. Given two faithfully sorted input sequences $X$ and $Y$, the aim is to merge them into a unique faithfully sorted sequence $Z$, while possibly discarding $O(\alpha)$ keys. Here $\alpha$ denotes the number of faulty keys that appear out of order in $X$ and $Y$ plus the number of faults introduced during the execution of the merging process. Discarded keys are added to a fail sequence $F$. The algorithm scans $X$ and $Y$ and builds $Z$ and $F$ sequentially: the running indexes on all these sequences are stored in safe memory. Similarly to `Purifying-merge`, the merging process is divided into rounds. However, the implementation of `Purifying-merge` presented in [20] uses two auxiliary input buffers of size $\delta$ in which keys of $X$ and $Y$ to be merged at each round are copied. In order to avoid the use of these buffers, in Opt-Nb we identify the subsequences to be merged at each round by means of four auxiliary indexes that are stored in safe memory. Similarly, the output is not buffered, but written directly to the output sequence $Z$: an additional auxiliary index $z$ maintains the position of the last key added to $Z$ during previous rounds. The subsequences of $X$ and $Y$ identified by the auxiliary indexes are merged in the standard way. At the end of the round the algorithm performs a consistency check of cost $O(\delta)$. If the check succeeds, the auxiliary index $z$ can be updated with the current value of the running index of $Z$. Otherwise, the algorithm is able to identify a pair of keys which are not ordered correctly in either of the two merged subsequences: these two keys are moved to the fail set $F$. Before starting a new round, the remaining keys in either $X$ or $Y$ are compacted (shifting them towards higher positions) and the auxiliary indexes are updated. The analysis of the running time and of the number of discarded keys is exactly the same as for the buffered implementation of `Purifying-merge` (see [20] and Section 2.2 of this paper).

# 3 Experimental Framework

In this section we describe our experimental framework, discussing fault injection strategies, performance indicators, and additional implementation details.

## 3.1 Fault Injection: a Simulation Testbed

In order to study the impact of memory faults on the correctness and running time of sorting algorithms, we implemented a software testbed that simulates different fault injection

strategies. Our simulation testbed is based on two threads: the *sorting thread* runs the sorting algorithm, while the *corrupting thread* is responsible for injecting memory faults. In the following we discuss where, when, and how many faults can be generated by the corrupting thread.

**Fault location.** In order to simulate the appearance of memory faults, we implemented an *ad-hoc* memory manager: faults are injected in memory locations that are dynamically allocated through our manager. The location to be altered by a fault is chosen uniformly at random. As observed in Section 2.4, the algorithms' implementation is such that, at any time during the execution, only a constant number of reliable memory words is in use. To maximize the damage produced by a fault, the new value of the corrupted memory location is chosen (at random) so as to be always larger than the old value.

**Fault injection models.** The number of faults to be injected can be specified according to two different models. The *upper bound model* requires an upper bound $\delta$ on the total number of memory faults, and the actual number $\alpha$ of faults that should happen during the execution of an algorithm: it must be $\alpha \leq \delta$. The fault injection strategy ensures that exactly $\alpha$ memory faults will occur during the execution of an algorithm, independently from the algorithm's running time. This assumption, however, may not be true in a more realistic scenario, where algorithms with larger space consumption and larger execution times are likely to incur a larger number of faults. The *faults per unit time per unit memory* model overcomes this limitation (not addressed by the theoretical model of [18]) by using faults generated on a periodic time basis.

The faults per unit time per unit memory model requires to specify the error rate $\sigma$, i.e., the number of faults that must be injected per unit memory and per unit time. We note that the algorithms under investigation were not designed for this model, as they make explicit use of $\delta$ in their implementation. Hence, in order to stress an algorithm in this more realistic scenario, we need to start from an error rate $\sigma$ and to generate a suitable value of $\delta$ to be used by the algorithm itself. Such a value $\delta$ should be the smallest value larger than the expected number of faults generated during the execution of the algorithm, assuming fault rate equal to $\sigma$: this would guarantee correctness, while limiting the overhead as much as possible. Since the running time of the algorithm may be an increasing function of $\delta$, if the fault injection rate $\sigma$ is too fast it may be even possible that no value of $\delta$ satisfies the condition above. In this case the algorithm is not guaranteed to behave correctly in the faults per unit time per unit memory model. Additional details will be given in Section 4.4.

**Fault generation time.** In the upper bound model, before running the algorithm the corrupting thread precomputes $\alpha$ breakpoints, at which the sorting thread will be interrupted and one fault will be injected. The breakpoints can be spread over the entire algorithm execution, or concentrated in a given temporal interval (e.g., at the beginning or at the end of the execution). In both cases the corrupting thread needs an accurate estimate of the algorithm's running time in order to guarantee that the correct number of faults will be generated and evenly spread as required: an automatic tuning mechanism takes care of this estimate. In the faults per unit time per unit memory model, faults are simply generated at regular time intervals.

### 3.2 Experimental Setup

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, memory manager, and a test program, is publicly available at the URL: `http://www.dsi.uniroma1.it/~finocchi/experim/faultySort/`.

Unless stated otherwise, in our experiments we average each data point on ten different instances, and, for each instance, on five runs using different fault sequences on randomly chosen memory locations. Random values are produced by the `rand()` 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 sorting and the corrupting threads run as two different parallel processes on our biprocessor architecture and operating system: 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.

## 4 Experimental Results

In this section we summarize our main experimental findings. We performed experiments using a wide variety of parameter settings and instance families. In this paper we only report the results of our experiments with uniformly distributed integer keys: to ensure robustness of our analysis, we also experimented with skewed inputs such as almost sorted data and data with few distinct key values, and obtained similar results. We mainly focus on experiments carried out in the upper bound fault injection model, for which the algorithms have been explicitly designed. The same relative performances of the algorithms have been observed in the more realistic faults per unit time per unit memory model: we address this issue in Section 4.4.

### 4.1 The Price of Non-resiliency: Correctness

Our first aim was to measure the impact of memory faults on the correctness of the classical (non-resilient) mergesort, which we refer to as Vanilla mergesort. In the worst case, when merging two $n$-length sequences, a single memory fault may be responsible for a large disorder in the output sequence: namely, if the memory location affected by the fault is adversarially chosen, it may be necessary to remove as many as $\Theta(n)$ elements in order to obtain a 0-unordered subsequence. A natural question to ask is whether the output can be completely out of order even when few faults hit memory locations chosen at random.

In order to characterize the non-resiliency of Vanilla mergesort in this scenario, we ran the algorithm on several input sequences with a fixed number of elements while injecting an increasing number of faults spread over the entire algorithm's execution time. In our experiment memory locations hit by faults were chosen at random, and the corrupted value was chosen (at random) so as to be always larger than the old value. The correctness of the output has been measured using the $k$-unordered metric: we recall from Section 2 that a sequence is $k$-unordered if $k$ is the minimum number of (faithful) keys that must be removed

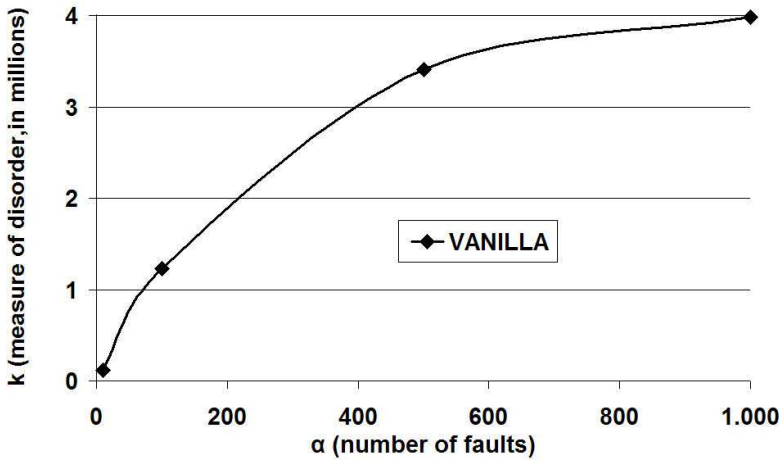



Figure 4: Disorder produced by random memory faults in the sequence output by VANILLA mergesort. In this experiment $n = 5 \cdot 10^6$ and $\alpha = \delta$ increases up to 1000.

in order to obtain a faithfully ordered subsequence. Given a sequence $S$, it is not difficult to compute the measure $k$ of disorder of $S$ by a dynamic programming algorithm: corrupted keys can be easily recognized during this computation (they are marked as corrupted by our memory manager) and not considered.

The outcome of the experiment, exemplified in Figure 4, shows a deep vulnerability of VANILLA mergesort even in the presence of very few random faults. As it can be seen from Figure 4, when sorting 5 million elements, it is enough to have only 10 random faults (i.e., roughly only 0.0002% of the input size) to get a $k$-unordered output sequence for $k \approx 115 \cdot 10^3$: in other words, only 0.0002% faults in the input are able to produce errors in approximately 2.3% of the output. The situation gets dramatically worse as $\delta$ increases: with 1000 random faults (i.e., roughly only 0.02% of the input size) 80% of the output will be disordered! Similar results have been obtained for different instance sizes. The influence of memory faults on the correctness of VANILLA mergesort turns out to be even stronger if we consider that: (1) our implementation uses two arrays of size $n$ to store the input and to assemble the output at each merging pass; and (2) among the memory faults happening during a merging pass, only faults corrupting an input memory location not yet read or an output memory location already written may be potentially dangerous. Since at any time there exist exactly $n$ such locations, a random fault has probability 1/2 to hit any of them. This halves the number of faults that can effectively influence the sorting process.

## 4.2 The Price of Resiliency: Running Time Overhead

In order to operate correctly, resilient algorithms must cope with memory faults and be prepared to pay some overhead in their running times. In the following experiments we will try to evaluate this overhead by comparing the sorting algorithms of Table 1 with respect to the non-resilient VANILLA mergesort.

For correctness, in Figure 5 we first compare the running time of VANILLA mergesort with the running times of different (non-resilient) implementations of QUICKSORT and SHELLSORT. The algorithm named QUICKSORT1 is a carefully tuned recursive implementation of quicksort

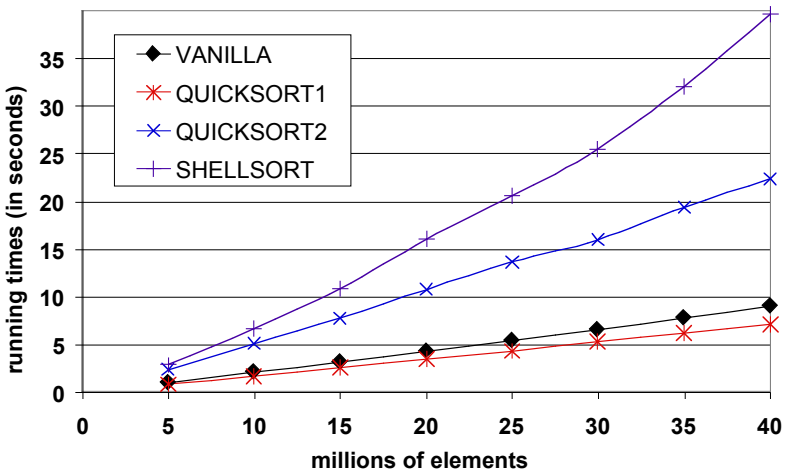
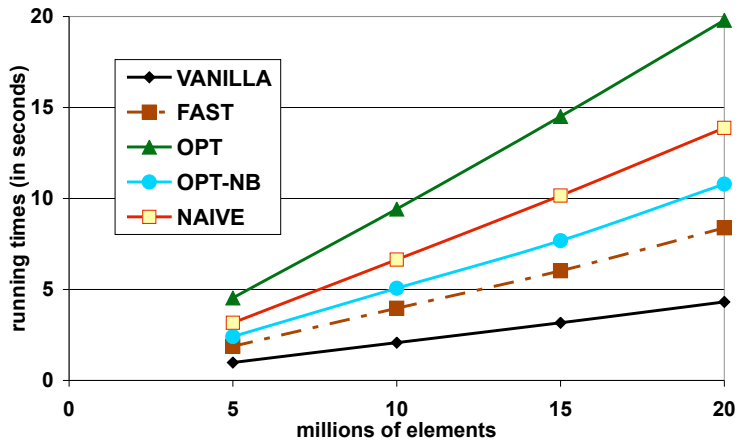



Figure 5: Comparison of the running times of non-resilient algorithms: VANILLA mergesort against tuned implementations of QUICKSORT and SHELLSORT.

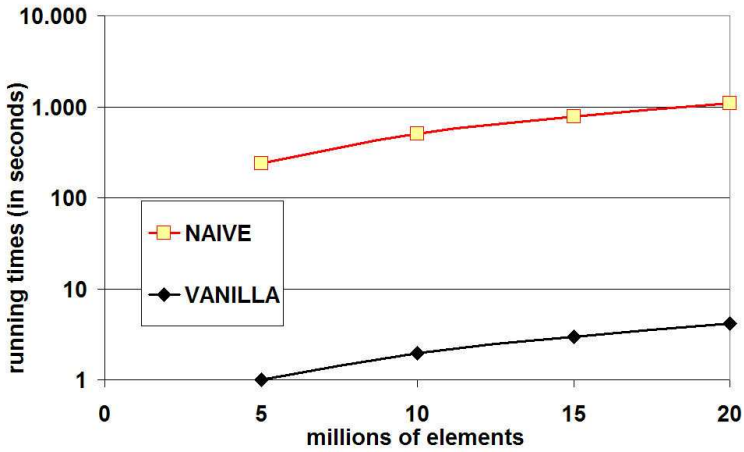
due to Bentley and Sedgewick [6]. The algorithms named QUICKSORT2 and SHELLSORT are available in the standard C library in Linux: these are general-purpose algorithms and use a comparator function that is passed as an argument. Thus, we expect them to be less efficient than QUICKSORT1. We note that our implementation of VANILLA mergesort compares rather well: its running time is only slightly larger than the running time of the fastest implementation of Quicksort, which is among the fastest algorithms in practice for internal memory sorting.

**Overhead of NAIVE.** Once the need for resilient algorithms is clear, even in the presence of very few non-pathological faults, another natural question to ask is whether we really need sophisticated algorithms for this problem. Put in other words, one might wonder whether a simple-minded approach to resiliency (such as the one used by algorithm NAIVE) would be enough to yield a reasonable performance in practice. To answer this question, we measured the overhead of NAIVE on random input sequences using different values of $\delta$. Figure 6(a) and Figure 6(b) illustrate two such experiments, where we measured the running times of VANILLA and NAIVE on random input sequences of increasing length by keeping fixed the number of faults injected during the sorting process ($\alpha = \delta = 2$ and $\alpha = \delta = 500$, respectively). The running times of NAIVE are comparable with those of VANILLA and of the other resilient algorithms when $\delta$ is very small: note, however, that even in this case FAST and OPT-NB are still preferable. As suggested by the theoretical analysis, when $\delta = 500$ the $\Theta(\delta)$ multiplicative factor in the running times of NAIVE makes this algorithm even hundreds of time slower than its non-resilient counterpart, and thus largely impractical. For this reason, we will not consider NAIVE any further in the rest of the paper and from now on we will focus our attention only on the more sophisticated algorithms FAST, OPT, and OPT-NB.

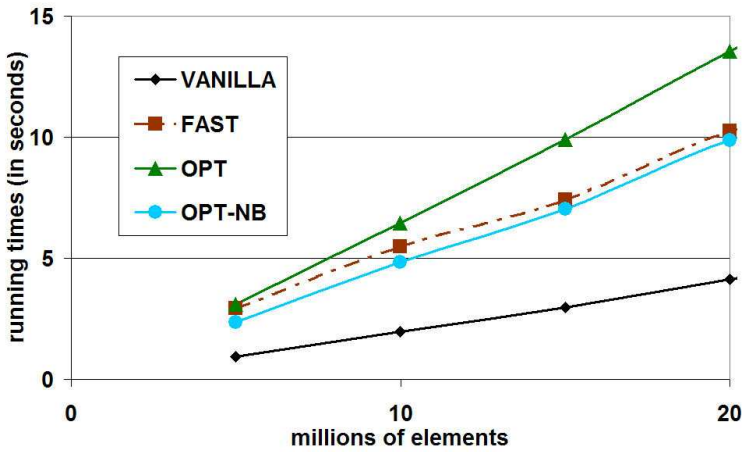
**Overhead of FAST, OPT, and OPT-NB.** According to the theoretical analysis, algorithms FAST, OPT, and OPT-NB are expected to be much faster than NAIVE. Overall, our experiments confirmed this prediction. Figure 6 illustrates the running times of the algorithms on random input sequences of increasing length using both small and large values of $\delta$. The chart




(a)




(b)




(c)

Figure 6: Running times on random input sequences of increasing length. (a) Vanilla and all the resilient algorithms with $\alpha = \delta = 2$; (b) Naive and Vanilla with $\alpha = \delta = 500$; (c) Fast, Opt, Opt-Nb, and Vanilla with $\alpha = \delta = 500$.

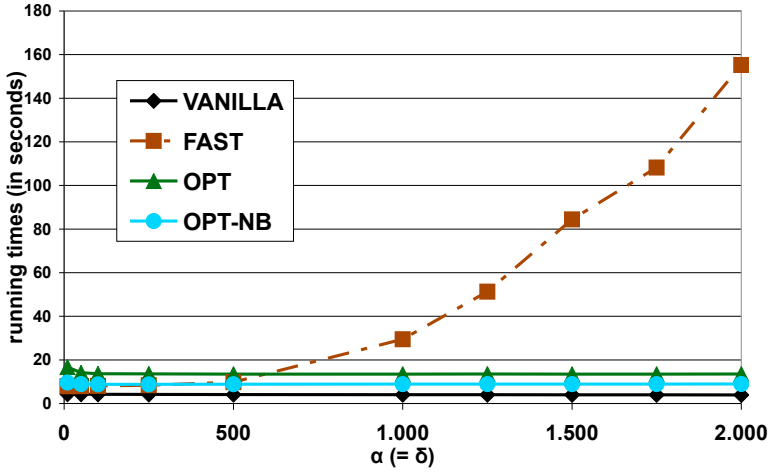
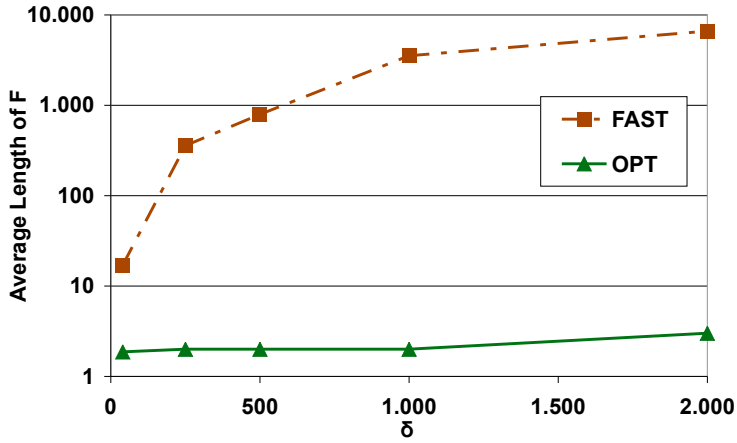



Figure 7: Running times of FAST, OPT, OPT-NB, and VANILLA on random input sequences of length $n = 20 \cdot 10^6$ and increasing number of injected faults.

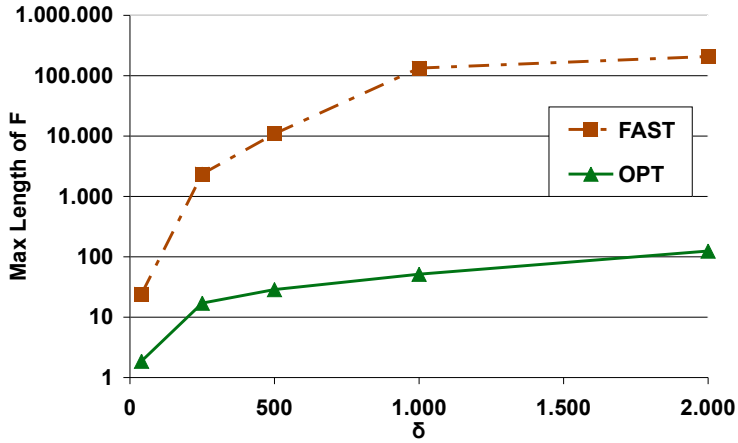
in Figure 6(a), for instance, is related to the case $\alpha = \delta = 2$ and shows that even for such small number of faults all the algorithms, except for OPT, are better than NAIVE. The advantage becomes more evident as $\delta$ gets larger: the charts in Figure 6(b) and Figure 6(c), for instance, have been obtained using $\alpha = \delta = 500$. They show that FAST, OPT, and OPT-NB perform very well for this choice of the parameters: indeed, they exhibit a running time which approximately ranges from 2.5 times to 3 times the running time of VANILLA mergesort, while NAIVE appears to be more than two orders of magnitude slower.

Note that, despite the theoretical bounds, FAST seems to have a better performance than OPT on this data set. This suggests that OPT may have larger implementation constants (probably due to the buffer management overhead) and that there are situations where FAST is able to perform better than OPT, at least in the presence of few faults. Our efforts in engineering OPT so as to avoid the use of buffers seem to pay off and confirm this intuition: indeed, in this experiment OPT-NB performs always better than OPT and better than FAST if $\delta$ is not too small.

**Impact of faults on the running time.** According to the asymptotic analysis, we would also expect that the performance of FAST, OPT, and OPT-NB degrade substantially as the number of faults becomes larger. In order to check this, we designed experiments in which the length of the input sequence is fixed (e.g., $n = 20 \cdot 10^6$) but the number $\alpha = \delta$ of injected faults increases. One of those experiments is illustrated in Figure 7, and shows that only the running time of FAST seems heavily influenced by the number of faults for the parameter settings considered in the experiment. OPT and OPT-NB, instead, appear to be quite robust as their running times tend to remain almost constant even for the largest values of $\delta$. This is not very surprising if we analyze the range of parameters of the experiment: ignoring the constant factors hidden by the asymptotic notation, the overhead $O(\alpha\delta)$ of algorithm OPT is indeed much smaller than the $O(n \log n)$ contribution to the running time when




(a)




(b)

Figure 8: Average and maximum length of the fail sequence $F$ to be sorted by NAIVE.

$\alpha = \delta = 2000$ and $n = 20 \cdot 10^6$. According to additional experiments not reported in this paper, a non constant trend in the behavior of OPT can be observed only for values of $\delta$ that are extremely large if compared to $n$: for instance, when $n = 5 \cdot 10^6$, the running time slowly starts increasing only for $\delta$ larger than 4000. In the case of FAST the situation is different, since even in the experiment illustrated by Figure 7 the overhead $O(\alpha\delta^2)$ dominates the running time. In summary, for large values of $\delta$ the theoretical analyses of both FAST and OPT appear to predict rather accurately their practical performances.

To get a deeper understanding of the large difference between FAST and OPT, we profiled all the algorithms: in particular, for FAST we observed that when $\delta$ is large, most of the time is spent in Step 2, where the disordered fail sequence $F$ returned by `Purify` is sorted by means of algorithm NAIVE. This suggests that either the number of calls to NAIVE or the number of elements to be sorted in each call tends to be bigger in FAST than in OPT. Computing the

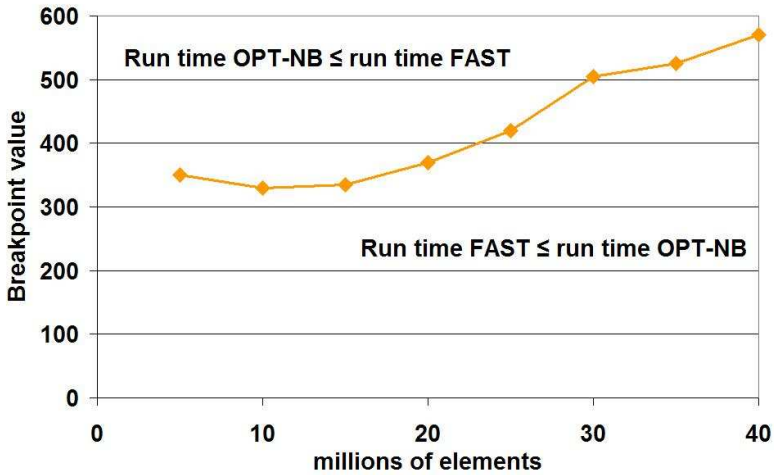


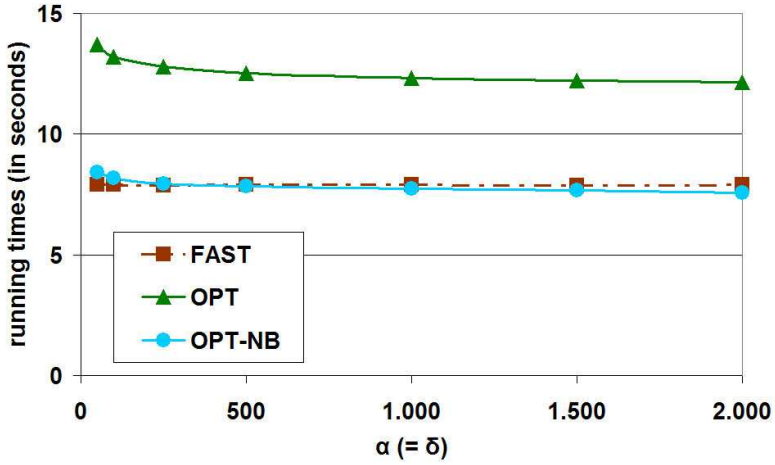

Figure 9: Breakpoint analysis: the breakpoint value is the smallest value of $\delta$ for which Opt-Nb becomes preferable to Fast.

average and the maximum length of the fail sequences $F$ throughout the algorithm execution confirmed the second hypothesis[1]. As shown in Figure 8, the fail sequences in Fast can be even $10,000$ times larger than in Opt. Since sorting the fail sequences appears to be a bottleneck in the resilient algorithms, the capability of obtaining much shorter fail sequences confirms also in practice the theoretical advantage of the purifying-merge approach over the weakly-resilient merge, at least for large values of $\delta$.

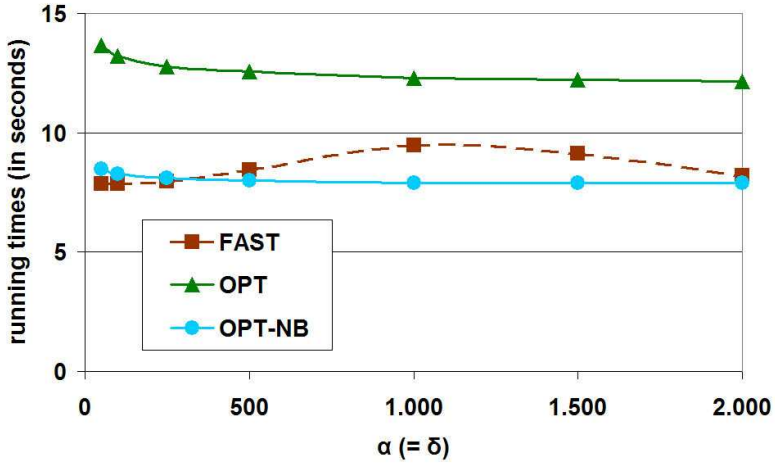
**Breakpoint analysis.** Figure 7 shows that Opt-Nb is always faster than Opt. However, in spite of the carefully engineered implementation of Opt-Nb and of the theoretical bounds, algorithm Fast remains faster for small values of $\delta$. In order to understand when Fast is preferable to Opt-Nb as the instance size changes, we performed a series of experiments whose outcome is summarized by Figure 9. The figure plots, for instance sizes ranging from 5 to 40 millions of elements, the smallest values of $\delta$ for which Opt-Nb becomes preferable to Fast. This breakpoint value roughly increases with the instance size, suggesting that algorithm Opt-Nb, although more efficient in practice than Opt, has still larger constant factors hidden by the asymptotic notation than algorithm Fast.

**Early versus late faults.** In all the experiments presented so far, we have considered random faults uniformly spread in time over the entire execution of the algorithm. The time interval in which faults happen, however, may significantly influence the running time of the algorithm. Indeed all of the resilient algorithms considered tend to process many sequences of smaller size in the initial stage of their execution, and few sequences of larger size at the end of their execution. As a consequence, faults occurring in the initial phase of the execution will likely produce short fail sequences, while faults occurring during the ending phase may produce longer fail sequences. Since sorting fail sequences appears to be a bottleneck in the

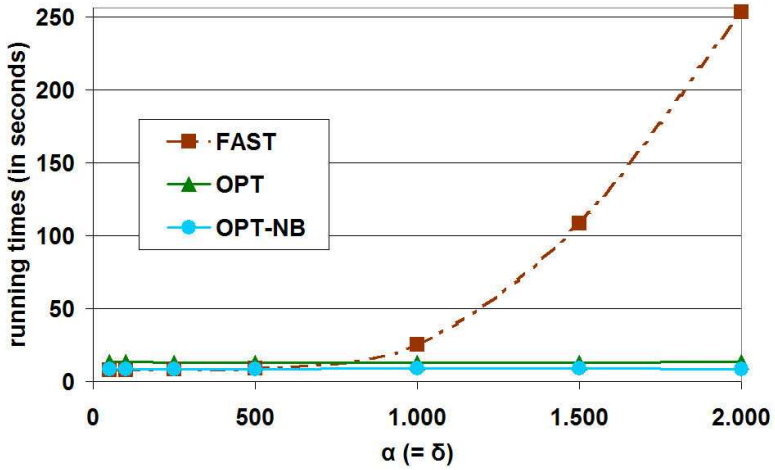
[1]The average length is obtained by first finding the arithmetic mean of the lengths of the fail sequences obtained during a single execution of an algorithm on a specific instance, and then averaging these values over five executions (with different fault sequences) and ten different instances.




(a)




(b)




(c)

Figure 10: Increasing number of faults concentrated in (a) the initial 20%, (b) the middle 20%, and (c) the final 20% of the algorithms' running times ($n = 20 \cdot 10^6$ in this experiment).

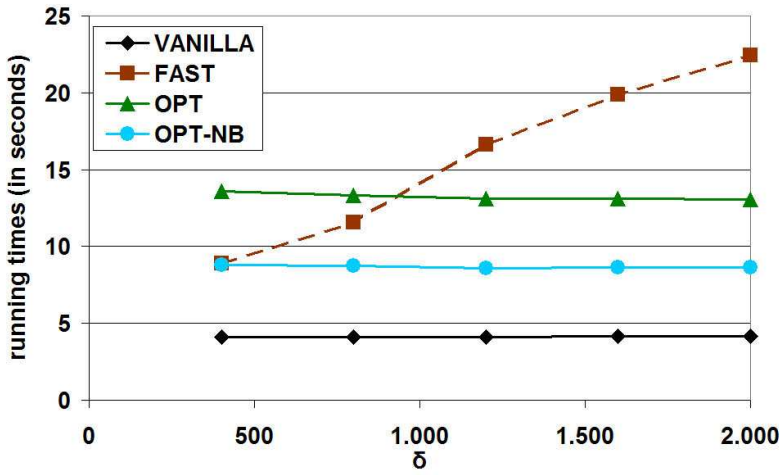



Figure 11: Sensitivity to $\delta$: in this experiment $n = 20 \cdot 10^6$, $\alpha = 400$, and $\delta \leq 2000$.

resilient algorithms, faults appearing at the beginning or at the end of the execution may produce quite different running times. To check this, we performed experiments in which we injected faults only in the initial 20%, middle 20%, and final 20% of the running time. The outcome of one of those experiments, presented in Figure 10, confirms that faults occurring early during the execution of FAST are processed more quickly than faults occurring late. The effect is instead negligible for algorithms OPT and OPT-NB, whose running time seems to be quite independent of the fault generation time. This fact is related to the result of Figure 8, that shows that not only the average, but also the maximum length of the fail sequences computed by these algorithms is always small.

### 4.3 Sensitivity to $\delta$

All the resilient algorithms considered in our experiments need an explicit upper bound $\delta$ on the number of faults that may happen throughout their execution. This is not a problem when $\delta$ is known in advance. However, if the rate of faults is unknown, the algorithms need at least an estimate on $\delta$ to work properly, and a bad estimate on $\delta$ may affect their running times or even their correctness. In this section we discuss issues related to finding a good estimate for $\delta$ in the upper bound model.

In all the experiments described up to this point we used $\delta = \alpha$. However, the exact number of memory faults that occur during the execution of an algorithm is not always known *a priori*. As mentioned above, this raises the question of estimating a suitable value of $\delta$ to be used by the algorithm: on the one side, rounding up $\delta$ may lead to much slower running times; on the other side, rounding it down may compromise the whole correctness of the resilient sorting algorithm. Within this framework, we analyzed the sensitivity of FAST, OPT, and OPT-NB with respect to variations of $\delta$. In the experiment illustrated in Figure 11, for instance, we run the algorithms on input sequences of length $n = 20 \cdot 10^6$, keeping the actual number of faults fixed ($\alpha = 400$) and increasing $\delta$ from 400 to 2000. When $\delta = 400$, this simulates a good estimate of $\delta$; as $\delta$ gets much larger than $\alpha$, this tend to simulate bad estimates of $\delta$. Note that, while the performances of OPT and OPT-NB are substantially unaffected by the increase on the value of $\delta$ for this choice of the parameters,

the running time of FAST seems to grow linearly with $\delta$: once again, this appears to depend on the fact that the length of the fail sequences in FAST is proportional to $\delta$, differently from what happens in the case of both OPT and OPT-NB. As a result, the experiment confirms the theoretical prediction that OPT and OPT-NB are much less vulnerable than FAST to potential bad estimates of the value $\delta$.

### 4.4 Faults per unit time per unit memory model

The faults per unit time per unit memory injection model introduces a major novelty with respect to the upper bound model, as the actual number of faults that will be generated throughout the execution of an algorithm is not bounded *a priori*. Thus, the quest for a good estimate of $\delta$ here is even more crucial. Note that in this model rounding up $\delta$ not only pushes additional overhead on the resilient sorting algorithm, but also increases the number of faults that will actually occur throughout the execution, because the running time becomes larger. In more details, let $t(n, \delta)$ denote the running time and let $\sigma$ be the error rate, per unit memory and per unit time. Then, ignoring constant factors in the running time and in the space usage, the actual number of faults that will occur is $\sigma \cdot t(n, \delta) \cdot n \log n$, since the sorting algorithms use $n \log n$ bits of memory. An algorithm is guaranteed to be correct only if

$$\delta \geq \sigma \cdot t(n, \delta) \cdot n \log n$$

Since the running time $t$ is an increasing function of $\delta$ itself, if the fault injection rate is too fast (i.e., $\sigma$ is too large) it may be possible that no value of $\delta$ satisfies the above inequality. In particular, the above inequality suggests that, given a fault rate $\sigma$, for all the resilient algorithms that we have considered there is an upper bound on the largest instance that can be faithfully sorted in the presence of faults that occur with rate $\sigma$. Similarly, given a number $n$ of keys to be sorted, there must be an upper bound on $\sigma$ such that the algorithm is guaranteed to sort faithfully $n$ keys only if the fault rate is smaller than that bound.

To investigate this issue, we tried to determine, for each algorithm and given the number $\sigma$ of faults per seconds, the smallest value of $\delta$ that is larger than the actual number $\sigma \cdot t(n, \delta) \cdot n \log n$ of faults: we will refer to this value of $\delta$ as the *correctness threshold*. Table 2 reports the correctness thresholds for the three algorithms FAST, OPT, and OPT-NB corresponding to six different values of $\sigma$ and to $n = 20 \cdot 10^6$. Since $n$ is fixed in this experiment and all the three algorithms use (asymptotically) $n \log n$ bits, it is more convenient to report the correctness thresholds as a function of $\sigma' = \sigma \cdot n \log n$. For a practical deployment of the algorithms studied in this paper in the realistic faults per unit time per unit memory model, one should provide lookup tables similar to Table 2 for several different values of $n$.

As one may expect, the correctness thresholds increase with $\sigma'$, and thus with $\sigma$: this is because larger values of $\sigma$ yield larger total numbers of injected faults. The experiment also confirms our intuition that a correctness threshold may not always exist, limiting the possibility of using the algorithms in the faults per unit time per unit memory model only when the fault injection rate is small enough. In particular, it is remarkable that algorithm FAST has no correctness threshold already for $\sigma' = 50$, i.e., it cannot tolerate more than 50 faults per second when the space usage is equal to $n \log n$ bits with $n = 20 \cdot 10^6$. Algorithms OPT and OPT-NB appear to be more robust and can tolerate much higher fault injection rates. This is in line with the previous experiments, where we observed that the running time of FAST grows much more quickly than the running times of OPT and OPT-NB as the number of faults increases.

| Algorithm | $\sigma' = 10$ | $\sigma' = 20$ | $\sigma' = 30$ | $\sigma' = 40$ | $\sigma' = 50$ | $\sigma' = 60$ |
|---|---|---|---|---|---|---|
| FAST | 100 | 180 | 290 | 410 | - | - |
| OPT-NB | 100 | 210 | 300 | 400 | 500 | 590 |
| OPT | 160 | 310 | 445 | 620 | 770 | 880 |

Table 2: Correctness thresholds for $n = 20 \cdot 10^6$ and $\sigma' = \sigma \cdot n \log n \in [10, 60]$. Algorithm FAST cannot be run when $\sigma' \geq 50$ (no correctness thresholds exist).

| Algorithm | $\sigma' = 10$ | $\sigma' = 20$ | $\sigma' = 30$ | $\sigma' = 40$ | $\sigma' = 50$ | $\sigma' = 60$ |
|---|---|---|---|---|---|---|
| FAST | 8178 | 8285 | 8576 | 9097 | - | - |
| OPT-NB | 8894 | 8866 | 8931 | 8957 | 8912 | 8992 |
| OPT | 13878 | 13714 | 13725 | 13671 | 13738 | 13672 |

Table 3: Running times (measured in milliseconds) of algorithms FAST, OPT, and OPT-NB in the faults per unit time per unit memory model when $n = 20 \cdot 10^6$ and $\sigma' = \sigma \cdot n \log n \in [10, 60]$. For each value of $\sigma'$, the algorithms have been run using the correctness thresholds given in Table 2.

We used the correctness thresholds shown in Table 2, when defined, to compare FAST, OPT, and OPT-NB in the faults per unit time per unit memory model. The outcome of one of these experiments is shown in Table 3. The experiment confirmed the relative performances observed in the upper bound model. If the fault injection rate is small, the running times of OPT-NB and FAST are comparable, with FAST slightly better when $\sigma' \leq 30$: this confirms that FAST might be preferable to OPT-NB for small numbers of faults (Figure 7 and Figure 9 give the corresponding results in the upper bound model). When $\sigma' \geq 50$, however, FAST cannot be run since no correctness threshold exists. OPT and OPT-NB, instead, can tolerate such higher fault injection rates and still guarantee the correctness of their output. Notice that, similarly to the upper bound model, OPT is about 1.5 times slower than OPT-NB.

# 5 Conclusions

In this paper we have addressed the problem of sorting in the presence of faults that may arbitrarily corrupt memory locations. We have experimentally investigated the impact of memory faults both on the correctness and on the running times of mergesort-based sorting algorithms, including the algorithms FAST and OPT presented in [18, 20], a naive approach, and a carefully engineered version of OPT, named OPT-NB, introduced in this paper. We have performed experiments using a variety of fault injection strategies and different instance families. The full software package used in our study is publicly available at the URL: `http://www.dsi.uniroma1.it/~finocchi/experim/faultySort/`.

Our experiments give evidence that simple-minded approaches to the resilient sorting problem are largely impractical, while the design of more sophisticated algorithms seems worth the effort: the algorithms presented in [18, 20] are not only theoretically efficient, but also fast in practice. In particular, our engineered implementation of OPT is robust to different memory fault patterns and appears to be the algorithm of choice for most parameter choices. Algorithm FAST, however, might be preferable for rather small values of $\delta$.

The model in which the algorithms under investigation have been designed assumes that there exist an upper bound $\delta$ on the maximum number of faults that may happen throughout the execution of the algorithm. However, in practice algorithms with larger execution times are likely to incur a larger number of memory faults. We have thus adapted the algorithms to work in a different faults per unit time per unit memory model. Even in this more realistic scenario, we have observed the same relative performances of the algorithms: Opt and Opt-Nb appear to be more robust than Fast and can tolerate higher fault rates.

The design and the theoretical analysis of resilient algorithms in the faults per unit time per unit memory model certainly deserve additional investigation. In particular, it would be interesting to understand whether it is possible to obtain fault-oblivious resilient algorithms, i.e., resilient algorithms that do not assume any knowledge on the maximum number $\delta$ of memory faults. In [10, 21, 25] the problem of designing resilient dictionaries and priority queues has been also addressed: an experimental study of the performances of these resilient data structures would represent a valuable contribution.

## Acknowledgments

We thank the anonymous referees for the many useful comments that improved the presentation of this paper.

## References

[1] R. Anderson and M. Kuhn. Tamper resistance – a cautionary note. *Proc. 2nd Usenix Workshop on Electronic Commerce*, 1–11, 1996.

[2] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. *Proc. International Workshop on Security Protocols*, 125–136, 1997.

[3] J. A. Aslam and A. Dhagat. Searching in the presence of linearly bounded errors. *Proc. 23rd STOC*, 486–493, 1991.

[4] S. Assaf and E. Upfal. Fault-tolerant sorting networks. *SIAM J. Discrete Math.*, 4(4), 472–480, 1991.

[5] Y. Aumann and M. A. Bender. Fault-tolerant data structures. *Proc. 37th IEEE Symp. on Foundations of Computer Science* (FOCS'96), 580–589, 1996.

[6] J. Bentley and R. Sedgewick. Quicksort is optimal. Invited talk at Stanford University, January 2002. Slides available at the URL `http://www.cs.princeton.edu/~rs/talks/QuicksortIsOptimal.pdf`.

[7] J. Blömer and J.-P. Seifert. Fault based cryptanalysis of the Advanced Encryption Standard (AES). *Proc. 7th International Conference on Financial Cryptography* (FC'03), LNCS 2742, 162–181, 2003.

[8] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. *Proc. EUROCRYPT*, 37–51, 1997.

[9] R. S. Borgstrom and S. Rao Kosaraju. Comparison based search in the presence of errors. *Proc. 25th STOC*, 130–136, 1993.

[10] G. S. Brodal, R. Fagerberg, I. Finocchi, F. Grandoni, G. F. Italiano, A. G. Jørgensen, G. Moruz and T. Mølhave. Optimal Resilient Dynamic Dictionaries. *Proc. 15th Annual European Symposium on Algorithms* (ESA'07), 347–358, 2007.

[11] B. S. Chlebus, A. Gambin and P. Indyk. Shared-memory simulations on a faulty-memory DMM. *Proc. 23rd International Colloquium on Automata, Languages and Programming* (ICALP'96), 586–597, 1996.

[12] B. S. Chlebus, L. Gasieniec and A. Pelc. Deterministic computations on a PRAM with static processor and memory faults. *Fundamenta Informaticae*, 55(3-4), 285–306, 2003.

[13] C. R. Cook and D. J. Kim. Best sorting algorithms for nearly sorted lists. *Comm. of the ACM*, 23, 620–624, 1980.

[14] A. Dhagat, P. Gacs, and P. Winkler. On playing "twenty questions" with a liar. *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms* (SODA'92), 16–22, 1992.

[15] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surveys*, 24, 441–476, 1992.

[16] U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM J. on Comput.*, 23, 1001–1018, 1994.

[17] U. Ferraro Petrillo, I. Finocchi, G. F. Italiano. The Price of Resiliency: a Case Study on Sorting with Memory Faults. *Proc. 14th Annual European Symposium on Algorithms* (ESA'06), 768–779, 2006.

[18] I. Finocchi and G. F. Italiano. Sorting and searching in faulty memories. *Algorithmica*, 52(3), 309–332, 2008. Preliminary version in *Proc. 36th ACM STOC*, 101–110, 2004.

[19] I. Finocchi, F. Grandoni, and G. F. Italiano. Designing reliable algorithms in unreliable memories. *Computer Science Review*, 1(2), 77–87, 2007.

[20] I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal sorting and searching in the presence of memory faults. *Proc. 33rd ICALP*, LNCS 4051, 286–298, 2006. To appear in *Theoretical Computer Science*.

[21] I. Finocchi, F. Grandoni, and G. F. Italiano. Resilient search trees. *Proc. 18th ACM-SIAM Symposium on Discrete Algorithms* (SODA'07), 547–553, 2007.

[22] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. *Proc. IEEE Symposium on Security and Privacy*, 154–165, 2003.

[23] M. Henzinger. The past, present and future of Web Search Engines. Invited talk. *31st ICALP*, 2004.

[24] P. Indyk. On word-level parallelism in fault-tolerant computing. *Proc. 13th Annual Symp. on Theoretical Aspects of Computer Science* (STACS'96), 193–204, 1996.

[25] A. G. Jørgensen, G. Moruz and T. Mølhave. Priority queues resilient to memory faults. *Proc. 10th Workshop on Algorithms and Data Structures* (WADS'07), 127–138, 2007.

[26] D. J. Kleitman, A. R. Meyer, R. L. Rivest, J. Spencer, and K. Winklmann. Coping with errors in binary search procedures. *J. Comp. Syst. Sci.*, 20:396–404, 1980.

[27] H. Kopetz. Mitigation of Transient Faults at the System Level – the TTA Approach. *Proc. 2nd Workshop on System Effects of Logic Soft Errors*, 2006.

[28] K. B. Lakshmanan, B. Ravikumar, and K. Ganesan. Coping with erroneous information while sorting. *IEEE Trans. on Computers*, 40(9):1081–1084, 1991.

[29] T. Leighton and Y. Ma. Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. *SIAM J. on Comput.*, 29(1):258–273, 1999.

[30] T. Leighton, Y. Ma and C. G. Plaxton. Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults. *J. Comp. Syst. Sci.*, 54:265–304, 1997.

[31] C. Lu and D. A. Reed. Assessing fault sensitivity in MPI applications. *Proc. 2004 ACM/IEEE Conf. on Supercomputing* (SC'04), 37, 2004.

[32] T. C. May and M. H. Woods. Alpha-Particle-Induced Soft Errors In Dynamic Memories. *IEEE Trans. Elect. Dev.*, 26(2), 1979.

[33] B. Panzer-Steindel. Data integrity. Available at `http://indico.cern.ch/getFile.py/access?contribId=3&sessionId=0&resId=1&materialId=paper&confId=13797`, april 2007.

[34] A. Pelc. Searching with known error probability. *Theoretical Computer Science*, 63, 185–202, 1989.

[35] A. Pelc. Searching games with errors: Fifty years of coping with liars. *Theoret. Comp. Sci.*, 270, 71–109, 2002.

[36] E. Pinheiro, W. Weber, and L. A. Barroso. Failure trends in large disk drive populations. *Proc. 5th USENIX Conference on File and Storage Technologies*, 2007.

[37] B. Ravikumar. A fault-tolerant merge sorting algorithm. *Proc. 8th COCOON*, LNCS 2387, 440–447, 2002.

[38] D. A. Reed, C. Lu and C. L. Mendes. Reliability challenges in large systems. *Future Gener. Comput. Syst.*, 22(3), 293–302, 2006.

[39] G. K. Saha. Software based fault tolerance: a survey. *Ubiquity*, 7(25), 1, 2006.

[40] S. Skorobogatov and R. Anderson. Optical fault induction attacks. *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems* (CHES'02), LNCS 2523, 2–12, 2002.

[41] Tezzaron Semiconductor. *Soft errors in electronic memory - a white paper*, URL: `http://www.tezzaron.com/about/papers/Papers.htm`, January 2004.

[42] A. J. Van de Goor. *Testing semiconductor memories: Theory and practice*, ComTex Publishing, Gouda, The Netherlands, 1998.

[43] J. Von Neumann, Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, C. Shannon and J. McCarty eds., Princeton University Press, 43–98, 1956.

[44] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer. An experimental study of security vulnerabilities caused by errors. *Proc. International Conference on Dependable Systems and Networks*, 421–430, 2001.

[45] A. C. Yao and F. F. Yao. On fault-tolerant networks for sorting. *SIAM J. on Comput.*, 14, 120–128, 1985.