Designing Reliable Algorithms in Unreliable Memories*

Irene Finocchi[1], Fabrizio Grandoni[1], and Giuseppe F. Italiano[2]

[1] Dipartimento di Informatica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy. [finocchi,grandoni]@di.uniroma1.it
[2] Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Via del Politecnico 1, 00133 Roma, Italy. italiano@disp.uniroma2.it

Abstract. Some of today's applications run on computer platforms with large and inexpensive memories, which are also error-prone. Unfortunately, the appearance of even very few memory faults may jeopardize the correctness of the computational results. An algorithm is resilient to memory faults if, despite the corruption of some memory values before or during its execution, it is nevertheless able to get a correct output at least on the set of uncorrupted values. In this paper we will survey some recent work on reliable computation in the presence of memory faults.

1 Introduction

The inexpensive memories used in today's computer platforms are not fully secure, and sometimes the content of a memory unit may be temporarily or permanently lost or damaged. This may depend on manufacturing defects, power failures, or environmental conditions such as cosmic radiation and alpha particles [15, 23, 30, 32]. Unfortunately, even very few memory faults may jeopardize the correctness of the underlying algorithms, and thus the quest for reliable computation in unreliable memories arises in an increasing number of different settings.

Many large-scale applications require the processing of huge data sets that can easily reach the order of Terabytes: for instance, NASA's Earth Observing System generates Petabytes of data per year, while Google currently reports to be indexing and searching over 8 billion Web pages. In all such applications processing massive data sets, there is an increasing demand for large, fast, and inexpensive memories, at any level of the memory hierarchy: this trend is witnessed, e.g., by the large popularity achieved in recent years by commercial Redundant Arrays of Inexpensive Disks (RAID) systems [7, 18], which offer enhanced I/O bandwidths, large capacities, and low cost. As the memory size becomes larger, however, the mean time between failures decreases considerably: assuming standard soft error rates for the internal memories currently on the market [30], a system with Terabytes of memory (e.g., a cluster of computers with a few

* Work supported by the Italian MIUR Project ALGO-NEXT "Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments".

Gigabytes per node) would experience one bit error every few minutes. Error checking and correction circuitry added at the board level could contrast this phenomenon, but would also impose non-negligible costs in performance and money: hence, it is not a feasible solution when speed and cost are both at prime concern.

A different application domain for reliable computation is fault-based cryptanalysis. Some recent optical and electromagnetic perturbation attacks [4, 29] 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. Differently from the almost random errors affecting the behavior of large size memories, in this context the errors are introduced by a malicious adversary that can assume some knowledge of the algorithm's behavior.

In order to protect the computation against destructive memory faults, data replication would be a natural approach. However, it can be very inefficient in highly dynamic contexts or when the objects to be managed are large and complex: copying such objects can indeed be very costly, and in some cases we might not even know how to do this (for instance, when the data is accessed through pointers, which are moved around in memory instead of the data itself, and the algorithm relies on user-defined access functions). In these cases, we can assume neither the existence of *ad hoc* functions for data replication nor the definition of suitable encoding mechanisms to maintain a reasonable storage cost. Instead, it makes sense to assume that it is the algorithms themselves in charge of dealing with memory faults.

Informally, we have a *memory fault* when the correct value that should be stored in a memory location gets altered because of a failure. 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 nevertheless able to get a correct output (at least) on the set of uncorrupted values. In this paper we survey some recent work on the design and analysis of resilient algorithms by focusing on the two basic problems of sorting and searching, which are fundamental in many large scale applications. For instance, the huge data sets processed by Web search engines are typically stored in low cost memories by means of 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 few bit flips in the index, that may become responsible of erroneous answers to keyword searches [16]. In Section 2 we will discuss different models and approaches proposed in the literature to cope with unreliable information. In Section 3 we will highlight a faulty memory model and overview known results and techniques.

2 Models and Related Work

The problem of computing with unreliable information or in the presence of faulty components dates back to the 50's [33]. Due to the heterogeneous nature

of faults (e.g., permanent or transient) and to the large variety of components that may be faulty in computer platforms (e.g., processors, memories, network nodes or links), many different models have been proposed in the literature. In this section we will briefly survey only those models that appear to be most relevant to the problem of computing with unreliable memories.

The liar model. Two-person games in the presence of unreliable information have been the subject of extensive research since Rényi [28] and Ulam [31] posed the following "twenty questions problem":

> Two players, Paul and Carole, are playing the game. Carole thinks of a number between one and one million, which Paul has to guess by asking up to twenty questions with binary answers. How many questions does Paul need to get the right answer if Carole is allowed to lie once or twice?

Many subsequent works have addressed the problem of searching by asking questions answered by a possibly lying adversary [1, 6, 11, 12, 19, 24–26]. These works consider questions of different formats (e.g., comparison questions or general yes-no questions such as "Does the number belong to a given subset of the search space?") and different degrees of interactivity between the players (in the adaptive framework, Carole must answer each question before Paul asks the next one, while in the non-adaptive framework all questions must be issued in one batch). We remark that the problem of finding optimal searching strategies has strong relationships with the theory of error correcting codes. Furthermore, different kinds of limitations can be posed on the way Carole is allowed to lie: e.g., fixed number of errors, probabilistic error model, or linearly bounded model in which the number of lies can grow proportionally with the number of questions. Even in the very difficult linearly bounded model, searching is now well understood and can be solved to optimality: Borgstrom and Kosaraju [6], improving over [1, 11, 25], designed an $O(\log n)$ searching algorithm. We refer to the excellent survey [26] for an extensive bibliography on this topic.

Problems such as sorting and selection have instead drastically different bounds. Lakshmanan *et al.* [20] proved that $\Omega(n \log n + k \cdot n)$ comparisons are necessary for sorting when at most k lies are allowed. The best known $O(n \log n)$ algorithm tolerates only $O(\log n / \log \log n)$ lies, as shown by Ravikumar in [27]. In the linearly bounded model, an exponential number of questions is necessary even to test whether a list is sorted [6]. Feige *et al.* [12] studied a probabilistic model and presented a sorting algorithm correct with probability at least $(1 - q)$ that requires $\Theta(n \log(n/q))$ comparisons. Lies are well suited at modeling transient ALU failures, such as comparator failures. Since memory data get never corrupted in the liar model, fault-tolerant algorithms may exploit query replication strategies. We remark that this is not the case in faulty memories.

Fault-tolerant sorting networks. Destructive faults have been first investigated in the context of fault-tolerant sorting networks [2, 21, 22, 34], in which comparators can be faulty and can possibly destroy one of the input values. Assaf and Upfal [2] present an $O(n \log^2 n)$-size sorting network tolerant (with high probability) to random destructive faults. Later, Leighton and Ma [21] proved

that this bound is tight. The Assaf-Upfal network makes $\Theta(\log n)$ copies of each item, using data replicators that are assumed to be fault-free.

Parallel computing with memory faults. Multiprocessor platforms are even more prone to hardware failures than uniprocessor computers. A lot of research has been thus devoted to deliver general simulation mechanisms of fully operational parallel machines on their faulty counterparts. The simulations designed in [8–10, 17] are either randomized or deterministic, and operate with constant or logarithmic slowdown, depending on the model (PRAM or Distributed Memory Machine), on the nature of faults (static or dynamic, deterministic or random), and on the number of available processors. Some of these works also assume the existence of a special fault-detection register that makes it possible to recognize memory errors upon access to a memory location.

Implementing data structures in unreliable memory. In many applications such as file system design, it is very important that the implementation of a data structure is resilient to memory faults and provides mechanisms to recover quickly from erroneous states. Unfortunately, many pointer-based data structures are highly non-resilient: losing a single pointer makes the entire data structure unreachable. This problem has been addressed in [3], providing fault-tolerant versions of stacks, linked lists, and binary search trees: these data structures have a small time and space overhead with respect to their non-fault-tolerant counterparts, and guarantee that only a limited amount of data may be lost upon the occurrence of memory faults.

Blum et al. [5] considered the following problem: given a data structure residing in a large unreliable memory controlled by an adversary and a sequence of operations that the user has to perform on the data structure, design a checker that is able to detect any error in the behavior of the data structure while performing the user's operations. The checker can use only a small amount of reliable memory and can report a buggy behavior either immediately after an errant operation (on-line checker) or at the end of the sequence (off-line checker). Memory checkers for random access memories, stacks and queues have been presented in [5], where lower bounds of $\Omega(\log n)$ on the amount of reliable memory needed in order to check a data structure of size n are also given.

3 Designing Resilient Algorithms

Memory faults alter in an unpredictable way the correct values that should be stored in memory locations. Due to such faults, we cannot assume that the value read from a memory location is the same value that has been previously written in that location. If the algorithm is not prepared to cope with memory faults, it may take wrong steps upon reading of corrupted values and errors may propagate over and over. Consider for instance mergesort: during the merge step, errors may propagate due to corrupted keys having value larger than the correct one. Even in the presence of very few faults, in the worst case as many as $\Theta(n)$ keys may be out of order in the output sequence, where n is the number of keys to be

merged. There are even more subtle problems with recursive implementations: if some parameter or local variable in the recursion stack (e.g., an array index) gets corrupted, the mergesort algorithm may recurse on wrong subarrays and entire subsequences may remain unordered.

3.1 The Faulty Memory Model

Memory faults may happen at any time during the execution of an algorithm, at any memory location, and even simultaneously. The last assumption is motivated by the fact that an entire memory bank may dismiss to work properly, and thus all the data contained in it may get lost or corrupted at the same time. In order to model this scenario, in [14] we introduced a *faulty-memory random access machine*, i.e., a random access machine whose memory locations may experience memory faults which corrupt their content. In this model corrupted values are indistinguishable from correct ones. We also assumed that the algorithms can exploit only $O(1)$ *reliable* memory words, whose content gets never corrupted: this is not restrictive, since at least registers can be considered fault-free.

Let δ denote an upper bound on the total number of memory faults that can happen during the execution of an algorithm (note that δ may be a function of the instance size). We can think of the faulty-memory random access machine as controlled by a malicious adaptive adversary: at any point during the execution of an algorithm, the adversary can introduce an arbitrary number of faults in arbitrary memory locations. The only adversary's constraint is not to exceed the upper bound δ on the number of faults. If the algorithm is randomized, we assume that the adversary has no information about the sequence of random values.

In the faulty-memory model described above, if each value were replicated k times, by majority techniques we could easily tolerate up to $(k-1)/2$ faults; however, the algorithm would present a multiplicative overhead of $\Theta(k)$ in terms of both space and running time. This implies, for instance, that in order to be resilient to $O(\sqrt{n})$ faults, a sorting algorithm would require $O(n^{3/2} \log n)$ time and $O(n^{3/2})$ space. The space may be improved using error-correcting codes, but at the price of a higher running time.

3.2 Resilient Sorting and Searching

It seems natural to ask whether it is possible to design algorithms that do not exploit data replication in order to achieve resilience to memory faults: i.e., algorithms that do not wish to recover corrupted data, but simply to be correct on uncorrupted data, without incurring any time or space overhead. More formally, in [14] we considered the following resilient sorting and searching problems.

- *Resilient sorting*: we are given a set of n keys that need to be sorted. The values of some keys may be arbitrarily corrupted during the sorting process. The problem is to order correctly the set of uncorrupted keys.

– *Resilient searching*: we are given a sequence of n keys on which we wish to perform membership queries. The keys are stored in increasing order, but some keys may be corrupted and thus may occupy wrong positions in the sequence. Let x be the key to be searched for. The problem is either to find a key equal to x, or to determine that there is no correct key equal to x.

In both cases, this is the best that we can achieve in the presence of memory faults. For sorting, we cannot indeed prevent keys corrupted at the very end of the algorithm execution from occupying wrong positions in the output sequence. For searching, memory faults can make the searched key x appear or disappear in the sequence at any time.

We remark that one of the main difficulties in designing efficient resilient sorting and searching algorithms derives from the fact that positional information is no longer reliable in the presence of memory faults: for instance, when we search an array whose correct keys are in increasing order, it may be still possible that a faulty key in position i is smaller than some correct key in position j, $j < i$, thus guiding the search towards a wrong direction.

Known results and techniques. In [14] we proved both upper and lower bounds on resilient sorting and searching. These results are shown for deterministic algorithms that do not make use of data replication and use only $O(1)$ words of reliable memory. We remark that a constant-size reliable memory may be even not sufficient for a recursive algorithm to work properly: parameters, local variables, return addresses in the recursion stack may indeed get corrupted. This is however not a problem if the recursion can be simulated by an iterative implementation using only a constant number of variables.

With respect to sorting, we 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. This lower bound implies that, if we have to sort in the presence of $\omega(\sqrt{n \log n})$ memory faults, then we should be prepared to spend more than $O(n \log n)$ time. We also designed a resilient $O(n \log n)$ comparison-based sorting algorithm that tolerates $O((n \log n)^{1/3})$ memory faults. The algorithm is based on a bottom-up iterative implementation of mergesort and hinges upon the combination of two merging subroutines with quite different characteristics. The first subroutine requires optimal linear time, but it may be unable to sort correctly all the uncorrupted keys (i.e., some correct elements may be in a wrong position in the output sequence). Such errors are recovered by using the second subroutine: this procedure may require more than linear time in general, but is especially efficient when applied to unbalanced sequences. The crux of the approach is therefore to make the disorder in the sequence returned by the first subroutine proportional to the number of memory faults that happened during its execution: this guarantees that the shorter sequence received as input by the second subroutine will have a length proportional to the actual number of corrupted keys, thus limiting the total running time. The gap between the upper and lower bounds for resilient sorting has been recently closed in [13], by designing an optimal resilient comparison-based sorting algorithm that can tolerate up to $O(\sqrt{n \log n})$ faults in $O(n \log n)$ time. In [13] we also prove that, in the

special case of integer sorting, there is a randomized algorithm that can tolerate up to $O(\sqrt{n})$ faults in expected linear time. No lower bound is known up to this time for resilient integer sorting.

With respect to searching, in [14] we designed an $O(\log n)$ time searching algorithm that can tolerate up to $O(\sqrt{\log n})$ memory faults and we proved that any $O(\log n)$ time deterministic searching algorithm can tolerate at most $O(\log n)$ memory faults. The lower bound has been extended to randomized algorithms in [13], where we also exhibit an optimal randomized searching algorithm and an almost optimal deterministic searching algorithm that can tolerate up to $O((\log n)^{1-\epsilon})$ memory faults in $O(\log n)$ worst-case time, for any small positive constant ϵ.

References

1. J. A. Aslam and A. Dhagat. Searching in the presence of linearly bounded errors. *Proc. 23rd ACM Symp. on Theory of Computing* (STOC'91), 486–493, 1991.
2. S. Assaf and E. Upfal. Fault-tolerant sorting networks. *SIAM J. Discrete Math.*, 4(4), 472–480, 1991.
3. Y. Aumann and M. A. Bender. Fault-tolerant data structures. *Proc. 37th IEEE Symp. on Foundations of Computer Science* (FOCS'96), 580–589, 1996.
4. 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.
5. M. Blum, W. Evans, P. Gemmell, S. Kannan and M. Naor. Checking the correctness of memories. *Proc. 32th IEEE Symp. on Foundations of Computer Science* (FOCS'91), 1991.
6. R. S. Borgstrom and S. Rao Kosaraju. Comparison based search in the presence of errors. *Proc. 25th ACM Symp. on Theory of Computing* (STOC'93), 130–136, 1993.
7. P. M. Chen, E. L. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2), 145–185, 1994.
8. B. S. Chlebus, A. Gambin and P. Indyk. PRAM computations resilient to memory faults. *Proc. 2nd Annual European Symp. on Algorithms* (ESA'94), LNCS 855, 401–412, 1994.
9. 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.
10. 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.
11. 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.
12. U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM Journal on Computing*, 23, 1001–1018, 1994.
13. I. Finocchi, F. Grandoni and G. F. Italiano. Optimal resilient sorting and searching in the presence of memory faults. Manuscript, 2005.

14. I. Finocchi and G. F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). *Proc. 36th ACM Symposium on Theory of Computing* (STOC'04), 101–110, 2004.
15. S. Hamdioui, Z. Al-Ars, J. Van de Goor, and M. Rodgers. Dynamic faults in Random-Access-Memories: Concept, faults models and tests. *Journal of Electronic Testing: Theory and Applications*, 19, 195–205, 2003.
16. M. Henzinger. The past, present and future of Web Search Engines. Invited talk. *31st Int. Coll. Automata, Languages and Programming*, Turku, Finland, July 12–16 2004.
17. 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.
18. R. H. Katz, D. A. Patterson and G. A. Gibson, *Disk system architectures for high performance computing*, Proceedings of the IEEE, 77(12), 1842–1858, 1989.
19. D. J. Kleitman, A. R. Meyer, R. L. Rivest, J. Spencer, and K. Winklmann. Coping with errors in binary search procedures. *Journal of Computer and System Sciences*, 20:396–404, 1980.
20. K. B. Lakshmanan, B. Ravikumar, and K. Ganesan. Coping with erroneous information while sorting. *IEEE Trans. on Computers*, 40(9):1081–1084, 1991.
21. T. Leighton and Y. Ma. Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. *SIAM Journal on Computing*, 29(1):258–273, 1999.
22. T. Leighton, Y. Ma and C. G. Plaxton. Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults. *Journal of Computer and System Sciences*, 54:265–304, 1997.
23. T. C. May and M. H. Woods. Alpha-Particle-Induced Soft Errors In Dynamic Memories. *IEEE Trans. Elect. Dev.*, 26(2), 1979.
24. S. Muthukrishnan. On optimal strategies for searching in the presence of errors. *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms* (SODA'94), 680–689, 1994.
25. A. Pelc. Searching with known error probability. *Theoretical Computer Science*, 63, 185–202, 1989.
26. A. Pelc. Searching games with errors: Fifty years of coping with liars. *Theoretical Computer Science*, 270, 71–109, 2002.
27. B. Ravikumar. A fault-tolerant merge sorting algorithm. *Proc. 8th Annual Int. Conf. on Computing and Combinatorics* (COCOON'02), LNCS 2387, 440–447, 2002.
28. A. Rényi. *A diary on information theory*, J. Wiley and Sons, 1994. Original publication: *Napló az információelméletröl*, Gondolat, Budapest, 1976.
29. 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.
30. Tezzaron Semiconductor. *Soft errors in electronic memory - a white paper*, URL: `http://www.tezzaron.com/about/papers/Papers.htm`, January 2004.
31. S. M. Ulam. *Adventures of a mathematician*. Scribners (New York), 1977.
32. A.J. Van de Goor. *Testing semiconductor memories: Theory and practice*, ComTex Publishing, Gouda, The Netherlands, 1998.
33. 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.
34. A. C. Yao and F. F. Yao. On fault-tolerant networks for sorting. *SIAM Journal on Computing*, 14, 120–128, 1985.