Dynamic programming in faulty memory hierarchies (cache-obliviously)*

Saverio Caminiti[1], Irene Finocchi[1], Emanuele G. Fusco[1], and Francesco Silvestri[2]

1 **Computer Science Department, *Sapienza* University of Rome**
{caminiti, finocchi, fusco}@di.uniroma1.it

2 **Department of Information Engineering, University of Padova**
silvest1@dei.unipd.it

Abstract

Random access memories suffer from transient errors that lead the logical state of some bits to be read differently from how they were last written. Due to technological constraints, caches in the memory hierarchy of modern computer platforms appear to be particularly prone to bit flips. Since algorithms implicitly assume data to be stored in reliable memories, they might easily exhibit unpredictable behaviors even in the presence of a small number of faults. In this paper we investigate the design of dynamic programming algorithms in faulty memory hierarchies. Previous works on resilient algorithms considered a one-level faulty memory model and, with respect to dynamic programming, could address only problems with local dependencies. Our improvement upon these works is two-fold: (1) we significantly extend the class of problems that can be solved resiliently via dynamic programming in the presence of faults, settling challenging non-local problems such as all-pairs shortest paths and matrix multiplication; (2) we investigate the connection between resiliency and cache-efficiency, providing cache-oblivious implementations that incur an (almost) optimal number of cache misses. Our approach yields the first resilient algorithms that can tolerate faults at any level of the memory hierarchy, while maintaining cache-efficiency. All our algorithms are correct with high probability and match the running time and cache misses of their standard non-resilient counterparts while tolerating a large (polynomial) number of faults. Our results also extend to Fast Fourier Transform.

1998 ACM Subject Classification B.8 [Performance and reliability]; F.2 [Analysis of algorithms and problem complexity]; I.2.8 [Dynamic programming].

Keywords and phrases Unreliable memories, fault-tolerant algorithms, dynamic programming, cache-oblivious algorithms, Gaussian elimination paradigm.

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2011.433

1 Introduction

Random access memories suffer from failures that lead the logical state of some bits to be read differently from how they were last written. A recent study has analyzed the memory-error sensitivity of Google's fleet of commodity servers over a period of nearly two years, observing an incidence of errors much higher than previously reported in laboratory conditions [26]. Due to low supply voltage and low critical charge per cell, caches in the memory hierarchy

* This work was supported in part by the Italian Ministry of Education, University, and Research (MIUR) under PRIN 2008TFBWL4 national research project AlgoDEEP. The last author was also supported by the University of Padova under Projects STPD08JA32 and CPDA099949/09.

© S. Caminiti, I. Finocchi, E.G. Fusco, and F. Silvestri;
licensed under Creative Commons License NC-ND

of modern computer platforms appear to be even more prone to bit flips than dynamic random access memories [22], while sophisticated error-correction algorithms are prohibitive for on-chip caches due to tight constraints on die size. The effect of memory errors is an important consideration in system design, especially for long-running and large-scale applications that work on massive data sets. When hardware techniques to detect bit flips are not available, it is important to design algorithms and data structures that are resilient to memory faults without incurring significant space/time penalties.

So far, algorithmic research related to memory faults mainly focused on fault-tolerant sorting networks [25] and on the design of resilient data structures in different (hardly comparable) models [2, 3, 9]. A variety of results have been recently obtained in a faulty RAM model introduced in [19], where an adversary can corrupt at most δ memory cells of a large unreliable memory during the execution of an algorithm. Problems solved resiliently in the faulty RAM model include sorting [17, 19], dictionaries [5, 18], priority queues [23], counting [7], K-d Trees [21], dynamic data structures [15], and local-dependency dynamic programming [8]. All these works focus on a one-level faulty memory: the only exception is [6], which investigates the connection between fault tolerance and I/O-efficiency in the external memory model [1], addressing resilient dictionaries, priority queues, and sorting. We remark that external memory (and, similarly, cache-aware) algorithms often crucially depend on the knowledge of hardware parameters, such as block or cache line size, and may not adapt well to different memory hierarchies. Cache-oblivious algorithms [20] overcome this issue: they are designed in a two-level ideal-cache model with no explicit dependencies on hardware parameters, and can therefore adapt simultaneously to all levels of the memory hierarchy (see [16] for a survey). To the best of our knowledge, no work but [6] proposes algorithms that are both resilient and cache-efficient. These two requirements pose indeed conflicting challenges: resilient algorithms typically use data replication, which might result in poor spatial locality, and perform error detection and recovery strategies throughout the computation, which might result in poor temporal locality.

Hierarchical faulty memory model. To analyze the resiliency of algorithms to faults that take place anywhere in the memory hierarchy, we combine the notions of fault-tolerance and cache-obliviousness extending the faulty RAM model [19] and the faulty external memory model [6] in a natural way. We assume the existence of a multilevel unreliable memory. Overall, at most δ (adversarial) corruptions can take place: each fault can be inserted at any time during the execution of an algorithm and at any level of the memory hierarchy. The algorithms can exploit knowledge of δ, which is a parameter of the model, but are oblivious to hardware parameters of the memory hierarchy and require neither error detection capabilities nor cryptographic assumptions. Following [20], we analyze the cache complexity in a two-level ideal-cache model, where both levels may be faulty: a fully associative cache of size M is partitioned into lines, each consisting of B consecutive words which are always moved together to/from main memory according to an optimal off-line replacement strategy (these choices are justified in [20]). Similarly to previous works [3, 6, 17, 18], we assume the existence of P private memory words that are incorruptible and hidden from the adversary: the private memory can be used, e.g., in the case of randomized algorithms to store random values and their derivatives. If $P = \Theta(1)$, the private memory can be implemented by a constant number of dedicated registers and accessed without incurring cache misses. If P is not constant (e.g., $P = \Theta(\log n)$, where n is the input size), we assume the existence of a private memory hierarchy whose largest level has size P: at each hierarchy level, private and public (unreliable) memory have the same cache line size.

Our results. We investigate the design of dynamic programming algorithms in the hierarchical faulty memory model. Previous work on resilient dynamic programming (in short, DP) [8] only applies to *local dependency DP problems*, where updates to entries in the DP table are determined by the contents of $O(1)$ neighboring cells: this class of problems includes, e.g., longest common subsequence and certain kinds of sequence alignment, but excludes many practically relevant problems such as Floyd-Warshall all-pairs shortest paths. Furthermore, algorithms in [8] are designed in a one-level memory model and are not cache-efficient. The contribution of this paper is two-fold. As a first result, we remove the local dependency assumption, significantly extending, w.r.t. [8], the class of problems that can be solved resiliently via dynamic programming in the presence of faults. Hinging upon a recursive framework introduced in [10, 11], we design resilient algorithms for all problems that can be solved by triply-nested loops of the type that occur in the standard Gaussian elimination algorithm, most notably all-pairs shortest paths and matrix multiplication. Similar results also apply to the Fast Fourier Transform. We remark that even checking the correctness of dynamic programming computations for non-local problems has been regarded as an elusive goal for many years. All our algorithms are correct with high probability, are parametric in the private memory size, and can tolerate a polynomial number of faults while still matching the running time of their non-resilient counterparts. As a second contribution, our approach yields the first resilient and cache-oblivious algorithms that can tolerate faults at any level of the memory hierarchy, while incurring an (almost) optimal number of cache misses. To obtain our results we introduce some novel techniques which might be of independent interest in the design of resilient algorithms for different problems.

To exemplify our bounds, consider a classical local-dependency DP problem, i.e., computing a longest common subsequence (LCS) of two sequences of length m and n, with $m \geq n$. We solve LCS resiliently and cache-obliviously in $O(nm + \delta n^{c/P} mP)$ time and $O(nm/(MB) + \delta n^{c/P} mP/B)$ cache misses, where M is the unreliable cache size, P is the private memory size (bounded by $O(\log n)$), B is the number of words in a cache line, δ is an upper bound on the number of faults, and $c < P$ is a small constant. Notice that $\Omega(nm/(MB) + \delta m/B)$ is a lower bound on the number of cache misses in the hierarchical faulty memory model [11], and that $n^{c/P} = \Theta(1)$ when $P = \Theta(\log n)$. Our algorithm matches the $\Theta(mn)$ running time of its non-resilient counterpart as long as $\delta = O(n^{1-c/P}/P)$, offering a full spectrum of tradeoffs between private memory size and number of faults. For instance, when $P = \Theta(\log n)$, we can tolerate up to $\delta = O(n/\log n)$ faults and incur a number of cache misses that is either optimal, if δ is also bounded by $O(n/(M \log n))$, or at most a factor of $\log n$ away from optimal. Even when $P = \Theta(1)$, the algorithm can still tolerate a polynomial number of faults within the same bounds of its non-resilient counterpart. Note that the resilient LCS algorithm from [8] incurs $\Theta(nm/B)$ cache misses, even without faults.

Paper organization. After some preliminaries, in Section 3 we introduce the main tools that will allow us to achieve resiliency and cache-efficiency simultaneously: this section is intended as an overview of our techniques, while the algorithms are detailed in Section 4 (which focuses on local-dependency DP problems) and in Section 5 (devoted to the extension to non-local problems). Due to lack of space, proofs of several results and some detailed description are omitted and will appear in the full version of the paper.

2 Preliminaries

Recursive dynamic programming [10, 11]. Our approach hinges upon a recursive framework for dynamic programming, introduced in [10, 11], that we briefly describe here

for completeness. We refer to [10, 11] for a detailed description and analysis. Let X and Y be two sequences of length n and m, respectively (w.l.o.g., let $m \geq n$). As an example we consider the LCS problem, whose standard DP solution is based on the following recurrence:

$$\ell[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \ell[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max\{\ell[i,j-1], \ell[i-1,j]\} & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases} \quad (1)$$

where $\ell[i,j]$ is the length of a longest common subsequence of prefixes $\langle x_1,\ldots,x_i\rangle$ and $\langle y_1,\ldots,y_j\rangle$. Values ℓ can be stored in a DP table C of size $(n+1)\times(m+1)$, and a longest common subsequence of X and Y can be obtained by computing a traceback path starting from entry $C[n,m]$, according to Equation 1.

Let $C[i,j][h,k]$ be the subtable of the dynamic programming table C ranging from row i to row j and from column h to column k. Let vectors $L = C[i-1,j][h-1,h-1]$, $R = C[i,j][k,k]$, $T = C[i-1,i-1][h-1,k]$, and $D = C[j,j][h,k]$ be the *left*, *right*, *top*, and *down boundaries* of the subtable, respectively. Moreover, let $\langle x_i,\ldots,x_j\rangle$ and $\langle y_h,\ldots,y_k\rangle$ be the *projections* of the input sequences X and Y on $C[i,j][h,k]$. The algorithm presented in [10, 11] is implemented by two recursive functions, Boundary and Traceback-Path, that use a divide-and-conquer strategy, logically splitting table C into four quadrants. Boundary performs a forward computation by recursively solving four subproblems: it returns the output boundaries (R and D) of a quadrant, starting from the projections of X and Y on the quadrant and the input boundaries (L and T). Traceback-Path finds the traceback path π through the DP table C by recursively finding the fragments of the path through the quadrants it traverses: given the entry point of path π on the output boundaries of a quadrant, Traceback-Path calls function Boundary to compute the input boundaries of (at most three) subquadrants, and recursively calls itself to compute the fragments of π traversing the subquadrants. In [12] the authors also provide a multicore version of the algorithm: this extension exploits a tiling sequence, depending on the recursion depth, that determines a subdivision of the DP table into a (not necessarily constant) number of quadrants.

Resilient variables [18]. An *r-resilient variable* x consists of $2r+1$ copies of a standard variable. A *reliable write* operation on x means assigning the same value to each copy. Similarly, a *reliable read* means calculating the majority value, which can be done in $\Theta(r)$ time and $O(1)$ space [4], incurring $O(r/B+1)$ cache misses. The majority value is guaranteed to be correct if $r \geq \delta$, since at most δ copies can be corrupted. If $r < \delta$, an r-resilient variable can be corrupted by the adversary, but at the cost of at least $r+1$ faults.

Karp-Rabin fingerprints [24]. Given a vector $A = \langle a_0,\ldots,a_k\rangle$ and a prime number p, a Karp-Rabin fingerprint can be defined as $\varphi_A = \sum_{i=0}^{k} a_i 2^{w(k-i)} \mod p$, where w is the memory word size. Fingerprint φ_A can be incrementally computed in $O(k)$ time and $O(1)$ private memory: when a new number a_i is revealed, φ_A can be updated in $O(1)$ time using Horner's rule and simple modular arithmetics [24].

3 Overview of our techniques

In this section we describe the main tools that will allow us to adapt some cache-oblivious algorithms to run in the presence of memory faults, while keeping the number of cache misses bounded. In Section 4 we will show how to combine these tools to make functions Boundary and Traceback-Path resilient.

Read and write fingerprints. The hierarchical faulty memory model does not provide fault detection capabilities: hence, we need to guarantee that values read throughout the

computation (from the input sequences and from the DP table) were not tampered since they were last written. To this aim, we use read and write Karp-Rabin fingerprints. Since Boundary and Traceback-Path are recursive, we will associate fingerprints to the input and output data of each call (i.e., to quadrant boundaries and to sequence projections). We use an independently generated prime number p_d for each recursion depth d and denote the write and read fingerprints of a vector A as φ_A and $\overline{\varphi}_A$, respectively. The correctness of data stored in A can be checked by reading A, computing $\overline{\varphi}_A$, and comparing its value against the write fingerprint φ_A produced when A was previously written: if $\varphi_A \neq \overline{\varphi}_A$, a fault occurred.

Bounding private data. We store fingerprints, primes, and information about recursive calls in the private memory, whose amount is limited to P memory words. Since $\Omega(1)$ data are necessary per recursion level, we need to limit the depth of the recursion tree, depending on P. Let c be the number of local variables used by the algorithm and let ρ be the largest integer such that $c\,\rho \leq P$: notice that $\rho = \Theta(P)$. At each call, we split the table into $\lambda \times \lambda$ quadrants, where $\lambda = \lceil n^{1/\rho} \rceil$: this guarantees that private data fits into P memory words.

Lazy fault detection. With non-constant λ and $O(1)$ fingerprints per recursion level, checking the correctness of the input for all the recursive calls would result in non-negligible time overhead. Hence, we detect faults lazily and perform fingerprint tests only when all subproblems have been recursively solved.

Data replication at decreasing resiliency levels. To resume computation after the detection of a fault, we use r-resilient variables. Since working at resiliency level δ throughout the computation would asymptotically increase the running time, we exploit a hierarchy of *decreasing resiliency levels*, tied with the depth of the recursive call: calls that are deeper in the recursion tree correspond to smaller subproblems and have a lower level of resiliency. It is worth noticing that the corruption of r-resilient variables, with $r < \delta$, together with lazy fault detection, might force the algorithm to perform entire subtree computations on wrong data. The corruption will be detected only by fingerprints at level r, but the wasted computation time will be amortized on $\Theta(r)$ faults. This will be crucial to bound the total cost of error recovery.

Amplified fingerprints. We will see in Section 4 that, during the execution of the algorithm, read and write data access patterns do not necessarily coincide. This is an issue, since updating a fingerprint $\overline{\varphi}_A$ while reading a vector $A = \langle a_0, \ldots, a_k \rangle$ according to an arbitrary pattern could require logarithmic time per access, due to exponentiation (see Section 2). Moreover, some values are possibly read $\omega(1)$ times and should appear in fingerprints tied with different exponents. To address these issues, we exploit regularities in data access patterns. We define an *amplified write fingerprint* as $\varphi_A = \sum_{i=0}^{k}(a_i \sum_{j=1}^{s_i} 2^{w f_{i,j}}) \mod p_d$, where s_i is the amplifying factor for element a_i (i.e., the number of times a_i will be read), and values $f_{i,j}$ are distinct positive integers characterizing the access pattern. The correctness of read data can be verified by updating the *amplified read fingerprint* $\overline{\varphi}_A$ adding $a_i 2^{w f_{i,j}}$ during the j-th reading of a_i. In our algorithm, factors $2^{w f_{i,j}}$ can be computed in $O(1)$ amortized time: these computations depend on the access pattern and will be therefore discussed later.

4 Recursive local-dependency dynamic programming

We now describe in more details how functions Boundary and Traceback-Path can be made resilient. We present a cache-efficient implementation in Section 4.1 and its analysis in Section 4.2. We assume that the input sequences are δ-resilient and that both of them have length n: the latter assumption can be removed by splitting the longer sequence Y into $\lceil m/n \rceil$ segments.

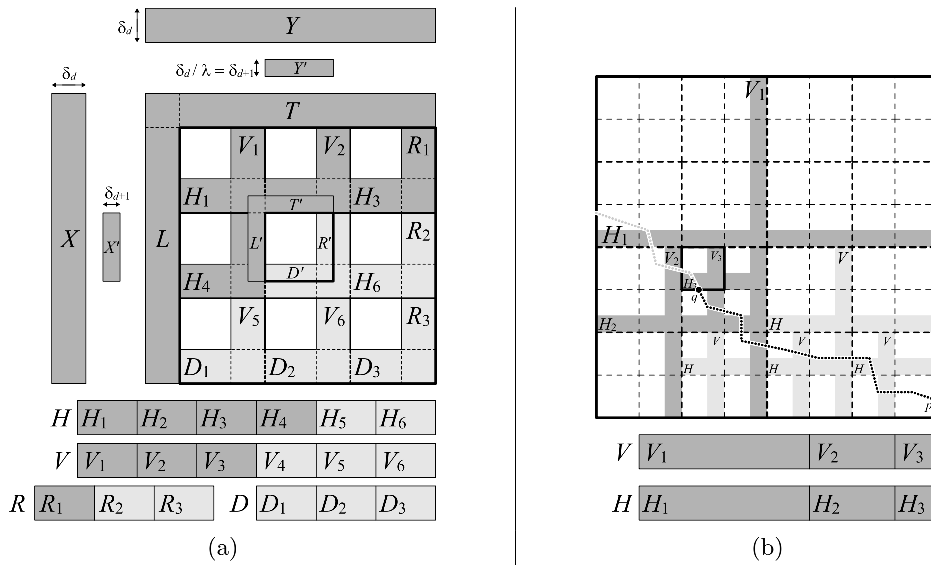

Figure 1 (a) DP quadrants, boundaries, and auxiliary vectors; (b) cache-oblivious traceback path computation with virtual quadrants.

Insert and extract. Throughout the computation, we repeatedly extract and merge vector segments, changing their resiliency level and updating their fingerprints, by means of two auxiliary functions, called `insert` and `extract`. Function `insert` combines two vectors: given as input a vector A stored at resiliency level r, a vector A' stored at resiliency level $r' \leq r$, two write fingerprints φ_A and $\varphi_{A'}$, it reads by majority values in A' and appends them to A, increasing their resiliency from r' to r (we assume that enough memory has been already allocated in A). At the same time, `insert` updates the write fingerprint φ_A with the new values and computes a read fingerprint $\overline{\varphi}_{A'}$ to check correctness of the read data: if $\overline{\varphi}_{A'} \neq \varphi_{A'}$, the function fails. Symmetrically, function `extract` takes a small vector out of a larger one.

Resilient boundary. Function Boundary, when called at recursion depth d, receives as input L, T, and the projections of X and Y, all stored at resiliency level $\delta_d = \lceil \delta/\lambda^d \rceil$, and the corresponding write fingerprints produced by its caller (initialization can be appropriately done at recursion depth 0). All the input vectors have the same length $n_d = \lceil n/\lambda^d \rceil$ and are stored in the unreliable memory, while fingerprints are private. Similarly to [12], Boundary recursively solves λ^2 subproblems in row-major order. However, the parameters of the recursive calls are vectors of length $n_{d+1} = \lceil n/\lambda^{d+1} \rceil$ stored at resiliency level $\delta_{d+1} = \lceil \delta/\lambda^{d+1} \rceil$, together with their write fingerprints, and the output is stored into two auxiliary (horizontal and vertical) vectors H and V. These vectors have length $(\lambda - 1)n_d$, resiliency δ_d, and are associated with a write and a read fingerprint (see Fig. 1.a). Appropriate fingerprint tests are also performed during the computation, as detailed below. Consider an internal quadrant $\langle i, j \rangle$, with $1 < i, j < \lambda$. During the recursive call at depth d, the algorithm works as follows:

Step 1. The input boundaries L' and T', of length n_{d+1} and resiliency δ_{d+1}, are extracted from V and H, respectively, together with their write fingerprints. The projection Y' of sequence Y on the quadrant is obtained similarly (X' is extracted from X only if $j = 1$). These `extract` operations also update the read fingerprints $\overline{\varphi}_H$, $\overline{\varphi}_V$, and $\overline{\varphi}_Y$.

Step 2. Given L', T', Y', and X' (together with their write fingerprints) computed in step

1, the recursive call on quadrant $\langle i, j \rangle$ returns the output boundaries D' and R', of length n_{d+1} and resiliency δ_{d+1}, and their write fingerprints $\varphi_{D'}$ and $\varphi_{R'}$. If this call fails, all data at resiliency level δ_{d+1} are discarded, the prime number p_{d+1} associated with recursion level $d+1$ is renewed, and the computation restarts from step 1. Backup copies of $\overline{\varphi}_V$, $\overline{\varphi}_H$, and $\overline{\varphi}_Y$ are used to restore the computation state. The projection of X is extracted once per row and requires an additional backup fingerprint based on prime number p_d.

STEP 3. Upon successful termination of the recursive call, the output boundaries D' and R' of quadrant $\langle i, j \rangle$ are merged with H and V, respectively. If `insert` fails (due to a fingerprint mismatch on δ_{d+1}-resilient data) the computation is restarted from step 1. Otherwise, the write fingerprints φ_H and φ_V are updated.

Quadrants on the first column and quadrants on the first row (i.e., $j = 1$ and $i = 1$, respectively) are handled similarly, but L' and T' are extracted from L and T (instead of V and H), respectively. When computing quadrants on the last row or column, the resulting output is inserted into the output vectors R and D with resiliency δ_d.

No fingerprint test is performed at the end of step 1 to establish the correctness of the extracted subsequences: such a test would require reading V and H from scratch, and would have a prohibitive running time. The following fingerprint tests are instead (lazily) performed, depending on the quadrant:

- $\overline{\varphi}_V = \varphi_V$ and $\overline{\varphi}_H = \varphi_H$ must hold if $\langle i, j \rangle = \langle \lambda, \lambda \rangle$;
- $\overline{\varphi}_L = \varphi_L$ and $\overline{\varphi}_T = \varphi_T$ must hold if $\langle i, j \rangle = \langle \lambda, 1 \rangle$ or $\langle 1, \lambda \rangle$, respectively;
- $\overline{\varphi}_Y = \varphi_Y$ must hold for each $i \in [1, \lambda]$ and $j = \lambda$;
- $\overline{\varphi}_X = \varphi_X$ must hold if $\langle i, j \rangle = \langle \lambda, 1 \rangle$.

A mismatch on any of the above tests implies failure of the recursive call at depth d and will be handled by higher recursive calls (see step 2).

Resilient traceback path. The resilient implementation of TRACEBACK-PATH computes the traceback path segment π traversing a quadrant, stored at resiliency level δ_d, and its write fingerprint φ_π. TRACEBACK-PATH calls resilient BOUNDARY to obtain vectors H and V, containing the output boundaries (at resiliency level δ_d) of the λ^2 quadrants of size $n_{d+1} \times n_{d+1}$. Then, it computes π backward from H and V by calling itself on (at most $2\lambda - 1$) subquadrants intersected by π. Segments of π (at resiliency level δ_{d+1}) obtained by the recursive calls are stitched and increased in resiliency using function `insert`. Fingerprint mismatches at resiliency level δ_d cause the current call of TRACEBACK-PATH to fail, while mismatches at level δ_{d+1} or failed subroutine invocations cause the computation to be repeated. Since the backward access pattern to H, V, L, and T is inverted with respect to the order in which data are written, we use amplified fingerprints as described in Section 3. E.g., the read fingerprint of vector H can be efficiently computed by saving in the private memory a running value $2^{w(|H|-i-1)} \mod p_d$ and performing a single multiplication by 2^w per update. We also notice that some quadrants may not be intersected by the traceback path, but we force the algorithm to read vector segments corresponding to these quadrants in order to correctly update the read fingerprints.

4.1 Cache-oblivious implementation

To improve temporal locality we access data in Z-order [20], and to improve spatial locality we shrink the size of data structures in the unreliable memory by recycling space as soon as written data are no longer needed. While this is quite standard in the design of cache-oblivious algorithms, it has non-trivial consequences on fingerprint computation.

Amplified fingerprints vs. Z-order. When using the Z-order, read operations on vectors H and V do not follow the write Z-order in which their write fingerprints φ_H and φ_V have been produced. We have thus to change the read fingerprint computation to reflect this different order while maintaining $O(1)$ amortized time per operation. Consider, e.g., the computation of $\overline{\varphi}_H$ for calls at recursion depth $d = \rho - 1$ (similar reasonings can be applied to the other vectors and recursion depths). Since $d = \rho - 1$, we have that $n_d \leq \lambda$, each subproblem is a single entry of the DP table C, and vector H corresponds to a portion of C of size $n_d \times n_d$. We use $\varphi_H = \sum_{x=0}^{|H|-1} H[x]2^{wx} \mod p_d$ as a write fingerprint. The computation of cell (i, j), requires values from cells $(i, j-1)$, $(i-1, j)$, and $(i-1, j-1)$. It can be shown that the ranks of these cells in vector H (say r_1, r_2, and r_3) can be computed from the rank r of cell (i, j) as $r_1 = r - \text{left}(\exp(j))$, $r_2 = r - \text{up}(\exp(i))$, and $r_3 = r - \text{diag}(\exp(i), \exp(j))$, where $\exp(k)$ is the exponent of 2 in the factorization of k, $\text{diag}(h, k) = \text{up}(h) + \text{left}(k)$, and up and left are defined as follows:

$$\text{left}(k) = \begin{cases} 1 & \text{if } k = 0 \\ 2^{2k-1} + \text{left}(k-1) & \text{otherwise} \end{cases} \qquad \text{up}(k) = \begin{cases} 2 & \text{if } k = 0 \\ 2^{2k} + \text{up}(k-1) & \text{otherwise} \end{cases}$$

Since each written value is read three times (from below, from the right, and from the bottom-right diagonal), we maintain three read fingerprints $\overline{\varphi}_{H,R}$, $\overline{\varphi}_{H,B}$, and $\overline{\varphi}_{H,D}$. While reading, e.g., cell $(i, j-1)$ to compute cell (i, j), we update $\overline{\varphi}_{H,R}$ by adding $H[r_1]2^{w(|H|-r)} \cdot 2^{-w(r-r_1)} \mod p_d$. To make this computation efficient, we precompute the inverse 2^{-w} of 2^w in the ring $\mathcal{Z}_{p_d}$, for all selected primes p_d. Computing $\text{left}(\exp(j))$ for all $j \in [0, \lambda - 1]$ requires $O(\lambda)$ sums and divisions by 2, and thus the amortized cost of each computation is constant. Within the same bound it is also possible to compute $2^{-w\,\text{left}(\exp(j))} \mod p_d$, starting from 2^{-w} and performing constantly many products for each sum in the computation of $\text{left}(\exp(j))$. Similar reasonings apply to functions up and diag.

Amplified fingerprints vs. virtual quadrants. To optimize cache misses, the length of vectors H and V, at recursion depth d, should be reduced from λn_d to $\Theta(n_d)$. We adapt a technique proposed in [11, 12]: at any time, only appropriate subvectors of H and V are stored, obtaining the missing parts, when necessary, by repeating forward computations. In more details, the $(n_d \times n_d)$-size DP table is split into four quadrants of size $(n_d/2 \times n_d/2)$, which are superimposed over the $\lambda \times \lambda$ submatrices. Let Q be the quadrant containing the entry point of π: the input boundaries for Q are obtained by applying function Boundary to at most three quadrants, and the traceback path π is computed by combining the output of at most three recursive calls of Traceback-Path on the intersected quadrants. It can be shown that the additional forward computations do not asymptotically increase the running time, and that the active boundaries stored in H and V (see Fig. 1.b) have length $O(n_d)$ [11, 12].

To apply this technique in our setting, two main issues need to be settled. A first technical issue is that recursion on $(n_d/2^i \times n_d/2^i)$-size quadrants cannot be explicit, since the recursion tree would exceed the amount of private memory when $P = o(\log n)$. Hence, in our implementation $(n_d/2^i \times n_d/2^i)$-size quadrants are only *virtual*: recursive calls of Traceback-Path on virtual quadrants inside an $n_d \times n_d$ quadrant are simulated iteratively (this can be done using only a constant number of indexes and variables), while real recursive calls are performed when $n_d/2^i$ becomes smaller than n_{d+1}. The resiliency is kept at level δ_d during simulated recursion, and drops to δ_{d+1} on real recursive calls.

The main issue is that, with virtual quadrants, the data access pattern becomes more complex and requires opportunely crafted amplified fingerprints to enable error detection while keeping negligible the time overhead. Consider as an example vector H and suppose for the moment that recursive calls of function Traceback-Path are performed, during simulated

recursion, on all quadrants. The access pattern on H is given by a layer of $\log_2 \lambda$ levels, where layer i corresponds to an inverted Z-order on 4^i data segments of length $n_d/2^i$. Hence, the number of elements in layer i is $n_d 2^i$ and the overall number of elements preceding the first element of layer i is given by $\sum_{j=1}^{i-1} n_d 2^i = n_d(2^i - 2)$. Write fingerprints are computed according to the subdivision in layers. Let l be the current depth of simulated recursion and let D' be the output received from a call to function Boundary. Layers involved in the fingerprint computation are layers l to $\log_2 \lambda$. As elements $h \in D'$ are inserted into H, φ_H is updated in constant amortized time by adding $h \sum_{i=l}^{\log_2 \lambda} 2^{w(n_d(2^i-2)+r_i n_d/2^i+s_i)}$ mod p_d, where r_i is the rank, in layer i, of the segment containing h and s_i is the number of elements preceding h in this segment. Vector V and the projections of the input sequences are handled in a similar way. Updates to read fingerprints are done accordingly to the current layer on quadrants intersected by the traceback path π, paying attention to work also on quadrants that are not intersected by π (to keep the read fingerprints consistent with the write fingerprints).

4.2 Analysis

Let α be the number of faults actually introduced by the adversary during an execution of the algorithm: notice that $\alpha \leq \delta$. We recall that the two input sequences have length m and n, with $m \geq n$, and that $\rho = \Theta(\log_\lambda n) = \Theta(P)$ is the recursion depth (see Section 3).

▶ **Theorem 1.** *Algorithm* Resilient-LCS *computes, with high probability, a correct longest common subsequence of the input sequences X and Y.*

Proof. If no memory fault is introduced by the adversary, the correctness of the algorithm follows from [10]. In general, the first call of function Traceback-Path has resiliency $\delta_0 = \delta$: this implies that majority values cannot be corrupted and computation is never aborted. It is not difficult to see that at most $\rho + \alpha$ selections of prime numbers are needed during an execution of the algorithm. It follows from [8] that, for any constants k and γ, it is possible to independently select $\rho + \alpha$ numbers in $[m^{k-1}, m^k]$, uniformly at random, so that all selected numbers are prime with probability at least $1 - (\rho + \alpha)/m^\gamma$. We now consider the probability that fingerprint tests do not fail. It can be shown using standard techniques that each fingerprint test fails to identify a corrupted variable with probability at most $1/(\sigma m^{k-2})$, for some positive constant σ. Since no more than α variables can be corrupted by the adversary during the execution of the algorithm, we have that the overall probability of detecting all faults is at least $1 - \alpha/(\sigma m^{k-2})$, provided that all selected numbers are primes. The probability that the algorithm computes the LCS correctly is thus at least $(1 - \alpha/(\sigma m^{k-2}))(1 - (\rho + \alpha)/m^\gamma)$. By appropriately choosing constants k and γ, this probability can be made larger than $1 - 1/m^\varepsilon$, for any $\varepsilon > 0$. ◀

The following theorem gives the running time of algorithm Resilient-LCS.

▶ **Theorem 2.** *Algorithm* Resilient-LCS *requires $O(mn + \delta mn^{c/P} P)$ time in the worst case, where P is the available private memory, $c < P$ is a small constant, m and n (with $m \geq n$) are the lengths of the input sequences, and δ is an upper bound on the number of memory faults.*

Proof. Algorithm Resilient-LCS consists of $\lceil m/n \rceil$ calls of Traceback-Path with input size n. Hence, functions Boundary and Traceback-Path are always called on two strings of length n. We first consider the time spent in successful computation and then take into account the time spent in computation discarded due to the detection of some fault. W.l.o.g

we assume n, δ and λ to be powers of two. Since at some recursion depth d the resiliency level $\delta_d = \delta/\lambda^d$ may become smaller than one, we suppose vectors to be $\Theta(\delta_d + 1)$-resilient.

Consider a successful computation of function Boundary at recursion depth d with input size n_d and resiliency level δ_d. If $n_d \leq \lambda$, the function requires $T_B(n_d, \delta_d) = O\left(n_d^2(\delta_d + 1)\right)$ time. If $n_d > \lambda$, the time $T_B(n_d, \delta_d)$ becomes $O\left(n_d^2 + \delta_d n_d \lambda \log_\lambda n_d\right)$ since Boundary performs λ^2 recursive calls with input size n_d/λ and resiliency δ_d/λ, and each call requires $O\left(n_d(\delta_d + 1)/\lambda\right)$ time for preparing inputs and fingerprints. Therefore $T_B(n, \delta) = O(n^2 + \delta n \lambda \log_\lambda n) = O(n^2 + \delta n n^{c/P} P)$, by definition of λ.

Inducing a recomputation at level $1 \leq i \leq k$, with $k = \log_\lambda \min\{n, \delta\}$, requires δ/λ^i faults (there cannot be recomputation at level $i = 0$ since boundaries are δ-resilient). Hence at most $\alpha\lambda^i/\delta$ recomputations can be induced. Since there are λ^{2i} subproblems at level i, the following summation bounds from above the time spent in unsuccessful computation:

$$\sum_{i=1}^{k} \frac{\alpha\lambda^i}{\delta} \frac{T_B\left(n, \delta\right)}{\lambda^{2i}} \leq T_B\left(n, \delta\right) \sum_{i=1}^{k} \frac{1}{\lambda^i} \leq T_B\left(n, \delta\right)$$

If $\delta < n$, recursive calls done at levels deeper than k are all done at resiliency level 1. The adversary can induce up to α recomputations at these levels, each of which has cost bounded by $T_B\left(n, \delta\right)/\lambda^{2k}$. Hence the time spent in unsuccessful computation at levels $j \in [k+1, \log_\lambda n]$ is upper bounded by: $\alpha T_B\left(n, \delta\right)/\lambda^{2k} = \alpha T_B\left(n, \delta\right)/\delta^2 < T_B\left(n, \delta\right).$ In all cases, this time does not exceed the time spent in successful computation.

Consider a successful computation of Traceback-Path at recursion depth d with input size n_d and resiliency level δ_d. If $n_d \leq \lambda$ the function requires $O\left(n_d^2(\delta_d + 1)\right)$ time. If $n_d > \lambda$, the time is $O\left(n_d^2 + n_d\delta_d\lambda^{\log_2 3}\log_\lambda n_d\right)$ since the function performs at most $2\lambda - 1$ recursive calls with input size n_d/λ and resiliency δ_d/λ, and calls at most 3^j times function Boundary with input size $n_d/2^j$ and resiliency $\delta_d + 1$, for each $1 \leq j \leq \log_2 \lambda$. As done previously, it can be shown that the time spent in unsuccessful computation does not exceed the time spent in successful computation. Multiplying these bounds by $\lceil m/n \rceil$ calls and recalling that $\lambda = n^{1/\Theta(P)}$, the theorem follows. ◀

Theorem 2 implies that, when $P = \Theta\left(\log n\right)$ and $\delta = O(n/\log n)$, the running time of algorithm Resilient-LCS is $O\left(nm\right)$, matching the running time of the non resilient cache-oblivious algorithm given in [10]. Furthermore, for any small constant private memory, the algorithm can still tolerate a polynomial number of faults within the non-resilient bounds. Theorem 3 gives the cache complexity of algorithm Resilient-LCS.

▶ **Theorem 3.** *Algorithm* Resilient-LCS *incurs* $O(mn/(BM) + \delta m n^{c/P} P/B)$ *cache misses in the worst case, where* $c < P$ *is a small constant.*

It follows from Theorem 3 that, when the available private memory is $\Theta\left(\log n\right)$ and $\delta = O(n/(M \log n))$, algorithm Resilient-LCS incurs $O(nm/(BM))$ misses, which is optimal for algorithms based on Equation (1), as proved in [10]. Notice that, in the hierarchical faulty memory model, we have an additional lower bound $\Omega\left(\delta m/B\right)$, given by the number of cache misses needed to read the input sequences at resiliency level δ. Hence, in any case, the number of cache misses is at most a $\log n$ factor away from optimal.

5 Extension to non-local problems

The techniques described in previous sections can be used to extend significantly the class of problems that are efficiently solvable in the presence of memory faults. Here, we sketch

resilient and cache-efficient algorithms for problems that fit in the Gaussian Elimination Paradigm [11] and for the Fast Fourier Transform. These algorithms compute the correct solution with high probability, when up to δ faults are inserted by an adversary.

Gaussian Elimination Paradigm (GEP). This paradigm includes all problems that can be solved by a triply nested `for` loop which updates each entry of an $n \times n$ input matrix C at most n times. Some notable examples are matrix multiplication, Gaussian elimination and LU decomposition without pivoting, and Floyd-Warshall all-pairs shortest paths. I-GEP [13] is a subclass of GEP which includes all the aforementioned problems. In I-GEP, it is possible to perform certain reorderings of the updates of matrix C guaranteeing that the final result remains correct, despite the fact that the intermediate states of C are different. In [12, 13, 14], cache-oblivious algorithms for I-GEP are provided for single processors, parallel, and multicore machines. Our algorithm Resilient-I-GEP is based on the multicore version [12]: it works recursively and relies on a subdivision of the input matrix into a varying number of square subproblems, depending on the recursion depth.

Let $\lambda = \lceil n^{1/\rho} \rceil$ be defined as in Section 3, where $\rho = \Theta(P)$. At recursion depth d, Resilient-I-GEP receives as input four δ_d-resilient $n_d \times n_d$ submatrices of C, which are divided into λ^2 submatrices of size $n_d/\lambda \times n_d/\lambda$. Initially, the four matrices coincide with C, which is stored δ-resiliently. The algorithm performs λ passes on these submatrices, solving λ^3 subproblems in total: the four input matrices of each subproblem are stored $\lceil \delta_d/\lambda \rceil$-resiliently. The execution of the λ^3 subproblems follows the order described in [13], which guarantees cache efficiency. For each of the $\Theta(P)$ recursive levels, the algorithm stores in the private memory $O(1)$ fingerprints which are opportunely crafted to reflect the execution order of the λ^3 subproblems and are similar in spirit to the amplified fingerprints used in Resilient-LCS. Algorithm Resilient-I-GEP solves I-GEP problems correctly with high probability: its running time is $O\left(n^3 + n^{2+c/P}\delta P\right)$ and the number of cache misses is $O(n^3/(B\sqrt{M}) + n^{2+c/P}\delta P/B)$, where $c < P$ is a suitable small constant. If $P = \Theta(\log n)$, the algorithm matches the running time of its non-resilient counterpart when $\delta = O(n/\log n)$. Optimal cache efficiency is achieved for $\delta = O(n/(M \log n))$.

Fast Fourier Transform (FFT). Algorithm Resilient-FFT is built on the cache-oblivious FFT algorithm in [20]. It computes the FFT of a δ-resilient n-size vector by computing $2\sqrt{n}$ FFTs on $\lceil \delta/\sqrt{n} \rceil$-resilient $\sqrt{n}$-size vectors. As usual, each input vector is associated with a fingerprint stored in private memory. The algorithm is correct with high probability and requires $\Theta(\log \log n)$ private memory. Its running time is $O(n \log n + n\delta)$, while the cache complexity is $O((n \log_M n)/B + n\delta/B + 1)$. The running time matches the corresponding bound of the non-resilient algorithm when $\delta = O(\log n)$. Optimal cache efficiency is achieved for $\delta = O(\log_M n)$. For larger values of δ, the algorithm matches the resilient lower bounds given by the misses and time required for reading the input vector resiliently.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.
2. Y. Aumann and M. A. Bender. Fault tolerant data structures. In *Proc. 37th FOCS*, pages 580–589, 1996.
3. M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2–3):225–244, 1994.
4. R. S. Boyer and J. S. Moore. MJRTY: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.

5 G. S. Brodal, R. Fagerberg, I. Finocchi, F. Grandoni, G. F. Italiano, A. G. Jørgensen, G. Moruz, and T. Mølhave. Optimal resilient dynamic dictionaries. In *Proc. 15th ESA*, volume 4698 of *LNCS*, pages 347–358, 2007.

6 G. S. Brodal, A. G. Jørgensen, and T. Mølhave. Fault tolerant external memory algorithms. In *Proc. 11th WADS*, volume 5664 of *LNCS*, pages 411–422, 2009.

7 G. S. Brodal, A. G. Jørgensen, G. Moruz, and T. Mølhave. Counting in the presence of memory faults. In *Proc. 20th ISAAC*, volume 5878 of *LNCS*, pages 842–851, 2009.

8 S. Caminiti, I. Finocchi, and E. G. Fusco. Local dependency dynamic programming in the presence of memory faults. In *STACS*, volume 9 of *LIPIcs*, pages 45–56, 2011.

9 V. Chen, E. Grigorescu, and R. de Wolf. Efficient and error-correcting data structures for membership and polynomial evaluation. In *Proc. 27th STACS*, volume 5 of *LIPIcs*, pages 203–214, 2010.

10 R. A. Chowdhury, H. S. Le, and V. Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *Trans. Comput. Biology Bioinform.*, 7(3):495–510, 2010.

11 R. A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proc. 17th SODA*, pages 591–600, 2006.

12 R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proc. 20th SPAA*, pages 207–216, 2008.

13 R. A. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. *Theor. Comput. Syst.*, 47:878–919, 2010.

14 R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *Proc. 24th IPDPS*, 2010.

15 Paul Christiano, Erik D. Demaine, and Shaunak Kishore. Lossless fault-tolerant data structures with additive overhead. In *Proc. 14th WADS*, volume 6844 of *LNCS*, 2011.

16 E. D. Demaine. Cache-oblivious algorithms and data structures. Lecture Notes from the EEF Summer School on Massive Data Sets, BRICS, 2001.

17 I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal resilient sorting and searching in the presence of memory faults. *Theor. Comput. Sci.*, 410(44):4457–4470, 2009.

18 I. Finocchi, F. Grandoni, and G. F. Italiano. Resilient dictionaries. *ACM Trans. on Algorithms*, 6(1), 2009.

19 I. Finocchi and G. F. Italiano. Sorting and searching in faulty memories. *Algorithmica*, 52(3):309–332, 2008.

20 M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th FOCS*, pages 285–298, 1999.

21 Fabian Gieseke, Gabriel Moruz, and Jan Vahrenhold. Resilient k-d trees: K-means in space revisited. In *ICDM*, pages 815–820, 2010.

22 B. L. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.

23 A. G. Jørgensen, G. Moruz, and T. Mølhave. Priority queues resilient to memory faults. In *Proc. 10th WADS*, volume 4619 of *LNCS*, pages 127–138, 2007.

24 R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

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

26 B. Schroeder, E. Pinheiro, and W. D. Weber. DRAM errors in the wild: a large-scale field study. *Commun. ACM*, 54(2):100–107, 2011.