Resilient dynamic programming[*]

Saverio Caminiti, Irene Finocchi, Emanuele G. Fusco
Computer Science Department
Sapienza University of Rome
{caminiti, finocchi, fusco}@di.uniroma1.it

Francesco Silvestri
Department of Information Engineering, University of Padova, Italy
IT University of Copenhagen, Denmark
silvest1@dei.unipd.it

January 26, 2015

Abstract

We investigate the design of dynamic programming algorithms in unreliable memories, i.e., in the presence of errors that lead the logical state of some bits to be read differently from how they were last written. Assuming that a limited number of memory faults can be inserted at run-time by an adversary with unbounded computational power, we obtain the first resilient algorithms for a broad range of dynamic programming problems, devising a general framework that can be applied to both iterative and recursive implementations. Besides all local dependency problems, where updates to table entries are determined by the contents of neighboring cells, we also settle challenging non-local problems, such as all-pairs shortest paths and matrix multiplication. All our algorithms are correct with high probability and match the running time of their standard non-resilient counterparts while tolerating a polynomial number of faults. The recursive algorithms are also cache-efficient and can tolerate faults at any level of the memory hierarchy. Our results exploit a careful combination of data replication, majority techniques, fingerprint computations, and lazy fault detection. To cope with the complex data access patterns induced by some of our algorithms, we also devise amplified fingerprints, which might be of independent interest in the design of resilient algorithms for different problems.

[*]Preliminary versions of this paper appeared in [14, 15]. Silvestri was supported by the University of Padova under projects STPD08JA32 and CPDA121378, by MIUR of Italy under project AMANDA, and by the European Research Council under grant 614331.

1 Introduction

Transient electronic noises and permanent hardware failures make dynamic random access memories susceptible to errors where the logical state of a bit is read differently from how it was last written. A large-scale study on Google's server fleet over a period of nearly 2.5 years observed error rates orders of magnitude higher than previously reported in laboratory conditions, with average FIT rates (failures in time per billion device hours) of 50,000 per Mbit [45]: this implies that a cluster of 1000 computers with 4 Gigabytes per node, for example, can experience one bit error every three seconds, with each node experiencing an error every 40 minutes. Fault-tolerant memory chips available on the market – based on parity checks and error-correcting codes – have limited fault coverage, typically correcting single bit errors and detecting two bits errors. Furthermore, they have large manufacturing and power costs, can cause significant delays, and can even determine interruptions of service upon detection of uncorrectable errors. Due to low supply voltage and low critical charge per cell, caches are also rather prone to bit flips [34], while sophisticated error-correction algorithms are prohibitive for on-chip memories due to tight constraints on die size.

If memory errors are not corrected, they can lead to applications using corrupted data and eventually to machine crashes. Silent data corruptions are a major concern in the reliability of modern systems: even a few of them may be harmful to both the correctness and the performance of software. Coping with memory faults appears to be of particular importance for all those applications handling massive data sets, for long-living processes, and for safety-critical avionics applications. A few works have also shown that bit flips can cause serious security vulnerabilities [7, 32].

A recent trend to cope with silent data corruptions is to design applications that are more tolerant to faults [48]: this "robustification" of software involves re-writing it so that dealing with faults simply causes the execution to take longer. Unfortunately, most algorithms and data structures are far from being robust: since the contents of memory locations are supposed to remain unchanged throughout the execution unless explicitly written by the program, wrong steps may be taken upon reading corrupted values, yielding unpredictable results: as an example, if only one single key in a sorted array of size n is corrupted, there can be as many as $\Theta(n)$ uncorrupted keys that could not be found by the standard binary search algorithm.

Related work. Algorithmic research related to memory errors spans more than thirty years. Starting from the "twenty questions game" posed by Rényi and Ulam in the late 70's, many results have been obtained in the liar model: see, e.g., [3, 9, 24, 40] and the extensive survey in [41]. More recently, sorting and selection have been studied in the "just noticeable difference model", where the outcome of comparisons is unpredictable if the compared values are within a fixed threshold [2]. These works assume transient comparator failures, but no corruption of data. Destructive faults have been first investigated in the context of fault-tolerant sorting networks [4, 38], and many subsequent works have focused on the design of resilient data structures in a variety of (hardly comparable) models: e.g., pointer-based data structures are the subject of [5], and error-correcting data structures for membership problems, based on locally decodable error-correcting codes, are presented in [16]. The more restrictive problem of checking (but not recovering) the behavior of large data structures that reside in an unreliable memory has also received considerable attention [8, 23]: checkers are typically correct with high probability and use an amount of reliable memory that is much smaller than the memory required by the data structure being checked.

A variety of resilient algorithms have been designed in the faulty-memory random access machine (faulty RAM), where an adaptive adversary can corrupt at most δ memory cells of a large unreliable memory during the execution of an algorithm [29]. Resiliency is achieved if a problem is solved correctly (at least) on the set of uncorrupted values. This relaxed definition of correctness fits naturally many problems addressed so far in this model, which include sorting [27, 29], dictionaries [11, 28], priority queues [35], selection [6, 37], counting [13], K-d trees [31], interval trees and suffix trees [22]. The efficiency and efficacy of some resilient algorithms have also been experimentally evaluated [42, 44, 26, 25]. A two-level faulty external memory model has been introduced in [12] to investigate the connection between fault tolerance and I/O-efficiency [1, 47]. To the best of our knowledge, no algorithms that are both resilient and cache-efficient across the entire memory hierarchy, in the spirit of cache-oblivious algorithms [30], are known in the literature.

Our results. In spite of the wealth of results summarized above, it remains an open question whether powerful algorithmic techniques, such as those based on dynamic programming (DP), can be made to work in the presence of faults: even checking DP computations has been regarded as an elusive goal for many years, especially for problems with non-local dependencies such as all-pairs shortest paths. In this paper we provide the first positive answers to this question, focusing on the faulty RAM model and obtaining the first resilient algorithms for dynamic programming. Our algorithms can tolerate, with high probability, destructive faults at any level of the memory hierarchy, while still incurring a low (additive) overhead on the running time and a small number of cache misses.

Throughout the paper we denote by $\alpha \leq \delta$ the *actual* number of faults occurring during a computation. Furthermore, we assume that the input is stored *reliably* so that correct input values can be recovered at any time by majority techniques. This can be obtained by replicating each input value $2\delta + 1$ times upon its first reading (corruptions that take place before the first reading cannot be detected). Our bounds, which cover a broad range of dynamic programming problems, are summarized in Table 1.

To illustrate our results, let us first consider the problem of computing a longest common subsequence (LCS) between two reliable n-length sequences. A simple-minded resilient implementation of the standard dynamic programming algorithm could be based on replicating all DP table values $2\delta+1$ times: this would result in a multiplicative $\Theta(\delta)$ overhead on both space usage and running time, yielding a $\Theta(\delta n^2)$-time algorithm. Hence, only a constant number of faults could be tolerated while matching the standard $\Theta(n^2)$ time bound of the non-resilient approach. In contrast, our algorithm IterLCS (Section 3) runs in $O(n^2 + n\delta + \alpha\delta^{1+\varepsilon})$ time, for any arbitrarily small constant $\varepsilon \in (0, 1]$: it can therefore tolerate an almost *linear* number of faults (i.e., up to $\delta = O(n^{1-\gamma})$ faults, for any arbitrarily small constant $\gamma > 0$) while maintaining the same asymptotic running time of its non-resilient counterpart.

The techniques used in Section 3 are independent of the specific recurrence relation that defines the table entries and can be applied to all local dependency dynamic programming problems, where updates to entries in the DP table are determined by the contents of $O(1)$ neighboring cells: this class includes, e.g., edit distance and certain kinds of sequence alignment [33], but excludes many practically relevant computations such as Floyd-Warshall all-pairs shortest paths algorithm. Moreover, algorithm IterLCS has poor temporal locality and is not cache-efficient. In the remainder of the paper we overcome these limitations as

Algorithm	Running time	Cache misses	Private memory	Section
IterLCS	$O(n^2 + \delta n + \alpha\delta^{1+\varepsilon})$	$\Omega\left(\frac{n^2}{B}\right)$	$O(1)$	§3
RecLCS	$O(n^2 + \delta n \log n)$	$O\left(\frac{n^2}{MB} + \frac{\delta n \log n}{B}\right)$	$O(\log n)$	§4
RecGEP	$O(n^3 + \delta n^2 \log n)$	$O\left(\frac{n^3}{B\sqrt{M}} + \frac{\delta n^2 \log n}{B}\right)$	$O(\log n)$	§5
ShallowLCS	$O(n^2 + \delta n^{1+c/P} P)$	$O\left(\frac{n^2}{MB} + \frac{\delta n^{1+c/P} P}{B}\right)$	P	§6
ShallowGEP	$O(n^3 + \delta n^{2+c/P} P)$	$O\left(\frac{n^3}{B\sqrt{M}} + \frac{\delta n^{2+c/P} P}{B}\right)$	P	§6

Table 1: Summary of our bounds for longest common subsequence (LCS) and Gaussian Elimination Paradigm (GEP).

follows:

- In Section 4 we prove that, if we can afford to use $O(\log n)$ *private* memory words (incorruptible and hidden from the adversary), then LCS can be solved resiliently and cache-obliviously via a recursive approach in $O(n^2 + \delta n \log n)$ time and $O(\frac{n^2}{MB} + \frac{\delta n \log n}{B})$ cache misses, where M is the (unreliable) cache size and B is the number of words in a cache line. As long as $\delta = O(\frac{n}{\log n})$, algorithm RecLCS runs in $O(n^2)$ time and is either cache-optimal, if δ is also bounded by $O(\frac{n}{M \log n})$, or at most a factor of $\log n$ away from optimal. Any non-resilient algorithm must indeed perform $\Omega(\frac{n^2}{MB})$ cache misses [18]. $\Omega(\frac{\delta n}{B})$ additional misses are required to write the output sequence reliably.
- In Section 5 we settle challenging non-local problems that fit in the Gaussian Elimination Paradigm (GEP), by exploiting a recursive framework introduced in [17, 18]. The GEP class includes problems solvable by triply-nested `for` loops of the type that occur in the standard Gaussian elimination algorithm, most notably matrix multiplication and Floyd-Warshall all-pairs shortest paths. Both the run-time overhead and the number of cache misses of algorithm RecGEP are close to optimal: any non-resilient algorithm must indeed perform $\Omega(\frac{n^3}{B\sqrt{M}})$ cache misses [18] and the reliable output matrix takes $\Omega(\delta n^2)$ space.
- In Section 6 we remove the logarithmic private memory assumption of Section 4 and Section 5, obtaining parametric algorithms that can adapt to private memories of different sizes. For instance, algorithm ShallowLCS solves LCS resiliently and cache-obliviously in $O(n^2 + \delta n^{1+c/P} P)$ time and $O(\frac{n^2}{MB} + \frac{\delta n^{1+c/P} P}{B})$ cache misses, where P is the private memory size (bounded by $O(\log n)$) and $c < P$ is a small constant. Notice that $n^{c/P} = \Theta(1)$ when $P = \Theta(\log n)$. This algorithm matches the $\Theta(n^2)$ time of its non-resilient counterpart as long as $\delta = O(\frac{n^{1-c/P}}{P})$, offering a full spectrum of tradeoffs between private memory size and number of faults. Similar tradeoffs can be achieved also for GEP problems.

To obtain our results we introduce some novel techniques which might be of independent interest in the design of resilient algorithms for different problems, as sketched below.

Techniques. Similarly to previous works in the field, we exploit knowledge of δ and a small (at most logarithmic) number of private memory words, but do not rely on any cryptographic assumptions. Since the faulty RAM model does not provide fault detection capabilities, we use *read-and-write Karp-Rabin fingerprints* to guarantee that values read throughout the computation were not tampered since they were last written. Such fingerprints have been effectively used for checking the correctness of a computation, but they alone are not powerful enough in the faulty RAM model, where we also need to recover the computation upon fault detection without restarting it from scratch. We overcome this issue by carefully combining fingerprinting with *data replication* and *majority techniques*.

Since working at resiliency δ throughout the entire computation would asymptotically increase the running time, we sometimes store data *semi-resiliently*, by replicating variables less than $2\delta+1$ times: semi-resilient data could be corrupted by the adversary, but at the cost of a large number of faults, allowing us to amortize the cost of a subproblem recomputation.

In the iterative algorithm, fingerprints and semi-resilient variables are associated to an asymmetric, hierarchical decomposition of the dynamic programming table into rectangular slices of height δ and decreasing width. In the recursive algorithms, fingerprints are associated to the input and output data of each recursive call and the resiliency level is tied with the depth of the call: calls that are deeper in the recursion tree correspond to smaller subproblems and have a lower level of resiliency. Fingerprints, prime numbers used for their generation, and information about recursive calls are all stored in the private memory. Since $\Omega(1)$ data are necessary per recursion level, in the parametric algorithms of Section 6 we limit the depth of the recursion tree by recursively solving a non-constant number of subproblems, using a *lazy fault detection* strategy. This might force the algorithm to perform entire subtree computations on wrong data, but we can prove that the wasted computation time can be appropriately amortized.

Combining read-and-write fingerprint computations with techniques aimed at achieving cache-efficiency also poses some unique challenges: read and write data access patterns may be different from each other and data may be discarded to improve spatial locality as soon as they are no longer needed, obtaining the discarded parts, when necessary, by appropriately repeating computations. To cope with these issues, we devise opportunely crafted *amplified fingerprints*, coupled with the complex data access patterns induced by multiple, out-of-order read and write operations. To the best of our knowledge, this is the first paper introducing the notion of amplified fingerprint, which might be of independent interest in the design of resilient algorithms for different problems.

2 Preliminaries

In this section we introduce preliminary definitions and concepts that will be useful throughout the paper, describing our hierarchical faulty memory model and presenting tools for resiliency that we will use as building blocks in our algorithms.

2.1 Model of computation

The faulty-RAM model assumes a unit cost RAM with wordsize w, distinguishing between an *unreliable* and a *private* memory. Up to δ unreliable memory words may be silently corrupted during the execution of an algorithm by an adaptive adversary with unlimited computational power, whereas the private memory consists of P memory words that are incorruptible and hidden from the adversary. No reliable computation would be possible without at least a constant number of incorruptible memory words [29]. Moreover, in the case of randomized algorithms, we assume that the random bits are not accessible to the adversary and that reading a memory word (in the unreliable memory) is an atomic operation, that is, the adversary cannot corrupt a memory word after the reading process has started. Without the last two assumptions, most of the power of randomization would be lost in our setting. Previous works in the faulty-RAM model have mainly assumed $P = \Theta(1)$. Different models of faulty memories, however, allow the presence of a non-constant private memory, as long as this is small with respect to the unreliable one (see, e.g., [8, 23, 46]). Moreover, the availability of a non-constant memory that cannot be corrupted is supported by recent research on hybrid systems that integrate algorithmic resiliency with the amount of memory protected by hardware ECC [39].

To analyze cache-efficiency, we extend the faulty-RAM and the faulty external memory [12] models assuming the existence of a *multilevel unreliable memory*: each of the $\alpha \leq \delta$ faults can take place at any level of the hierarchy. We also assume the existence of a *multilevel private memory* whose largest level has size P: at each hierarchy level, private and unreliable memory have the same cache line size. If $P = \Theta(1)$, we regard the private memory as implemented by a constant number of dedicated registers that can be accessed without incurring cache misses. Following [30], we focus on cache-oblivious algorithms and 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 [30]).

2.2 Tools for resiliency

Resilient variables. An *r-resilient variable x* consists of $2r+1$ copies of a standard variable, stored contiguously in the unreliable memory [28]. A *reliable write* operation on x means assigning the same value to each copy. Similarly, a *reliable read* means calculating the majority value: using the algorithm in [10], this can be done in $O(1)$ private memory words, $\Theta(r)$ time, and $\Theta(r/B + 1)$ cache misses, where B is the number of words in a cache line. The majority value is guaranteed to be correct if $r \geq \delta$, since at most δ copies can be corrupted. An r-resilient variable with $r < \delta$ can be corrupted by the adversary, but at the cost of at least $r + 1$ faults. Hence, we will say that such variables are *semi-resilient*.

Generation of random primes. Primes can be generated in the faulty-RAM model using a variant of a well-known algorithm based on iterated Miller-Rabin tests [43]. The following lemma is proved in the appendix:

Lemma 1. *For any constants γ, $c > 0$, it is possible to independently select k (not necessarily distinct) prime numbers in $I = [n^{c-1}, n^c]$, uniformly at random, with error probability bounded by k/n^γ. Each prime selection requires time polylogarithmic in n using a constant number of memory words.*

Read and write Karp-Rabin fingerprints. Given a vector $A = \langle a_0, \ldots, a_k \rangle$ and a prime number p, a Karp-Rabin fingerprint [36] can be defined as $\sum_{i=0}^{k} a_i 2^{w(k-i)} \mod p$, where w is the memory word size and each a_i fits into a memory word. All fingerprints and prime numbers will be stored in private memory. If A is revealed sequentially, its fingerprint can be incrementally computed in $O(k)$ time and $O(1)$ private memory: when a new value a_i is given, the fingerprint can be updated in $O(1)$ time using Horner's rule and simple modular arithmetics [36]. To perform error detection, we associate any vector A with a *write fingerprint* φ_A and a *read fingerprint* $\overline{\varphi}_A$: φ_A and $\overline{\varphi}_A$ are based on values written to and read from 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.

We define the fingerprint of an $n \times n$ matrix X to be the fingerprint of the vector X' containing its Z-order representation (our results can be easily adapted to a standard row-major representation). The Z-order representation of a matrix X is defined recursively: if $n = 1$ then $X' = X$, otherwise X' is given by the concatenation of the Z-order representations of $X_{0,0}$, $X_{0,1}$, $X_{1,0}$, and $X_{1,1}$, which denote the four quadrants of X (respectively, top-left, top-right, down-left, and down-right).

2.3 Amplified fingerprints

When read and write data access patterns to vector A are not sequential (and possibly different from each other), updating fingerprints φ_A and $\overline{\varphi}_A$ could require logarithmic time per access, due to exponentiation. Moreover, if values of A are accessed $\omega(1)$ times, they should appear in fingerprints tied with different exponents. We define an *amplified write fingerprint* as

$$\varphi_A = \sum_{i=0}^{k} \left(a_i \sum_{j=1}^{\mu_i} 2^{w f_{i,j}} \right) \mod p$$

where μ_i is the amplifying factor for element a_i (i.e., the multiplicity of read operations on a_i), and values $f_{i,j}$ are distinct positive integers tailored to the read and write patterns.

The correctness of read data can be verified by updating an *amplified read fingerprint* $\overline{\varphi}_A$ adding $a_i 2^{w f_{i,j}}$ during the j-th reading of a_i. The overhead of amplified fingerprints computations depends on the specific access pattern. In our algorithms, we will exploit regularities in data access patterns to compute factors $2^{w f_{i,j}}$ in $O(1)$ amortized time.

As a warmup example, in this section we analyze the simple case in which the amplifying factor is 1 and vector A is read in *reversed order* with respect to the order in which it was written. The reversed order pattern is exploited several times throughout the paper, namely in Section 4 and in Section 5. Amplified fingerprints for more complex patterns will be discussed in Section 6.

Let $k+1$ be the length of vector $A = \langle a_0, \ldots, a_k \rangle$. Entries a_i of vector A are read backwards, i.e., for $t = k$ downto 0. Item a_i is thus read at time $k - i$ (during the write fingerprint computation a_i was instead written at time i). We can describe this access pattern by choosing $f_{i,j} = k - i$: at time t during the read phase, we read item a_{k-t} and we thus have $f_{k-t,j} = t$. Notice that parameter j is not relevant since each entry is accessed only once, i.e., $\mu_i = 1$. We maintain in private memory a running value $v_t = 2^{wt} \mod p$, which is initialized to 1 and multiplied by 2^w at each new read operation. Upon reading a_{k-t} at time t, the amplified read fingerprint $\overline{\varphi}_A$ can be updated in constant time by adding $v_t a_{k-t}$. This yields the following result:

Lemma 2. *Amplified fingerprints for the reversed order pattern with amplifying factor 1 can be maintained in $O(1)$ worst-case time.*

3 Iterative local-dependency dynamic programming

In a local-dependency dynamic programming problem, the value of each cell of the DP table only depends on a constant number of neighboring cells and input symbols, according to a fixed access pattern. In this section we show how to make iterative algorithms for local-dependency DP problems resilient to memory faults.

We consider the computation of a longest common subsequence (LCS) as a canonical example of a problem with local dependencies. Given two input sequences X and Y, the LCS problem asks to find a subsequence that is common to both X and Y and has maximum length. The classic dynamic programming solution is based on the following recurrence:

$$\ell_{i,j} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \ell_{i-1,j-1} + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{\ell_{i,j-1}, \ell_{i-1,j}\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases} \tag{1}$$

where $\ell_{i,j}$ is the length of a longest common subsequence of prefixes $\langle x_1, \ldots, x_i \rangle$ and $\langle y_1, \ldots, y_j \rangle$. The length of an LCS of X and Y is thus given by $\ell_{n,m}$, where n and m are the lengths of X and Y, respectively (w.l.o.g., let $m \geq n$). In the classic non-resilient dynamic programming algorithm, values ℓ are stored in a DP table C of size $n \times m$ whose entries can be computed, e.g., in row-major order, in $\Theta(mn)$ time. An LCS of X and Y can be obtained by computing a traceback path starting from entry $C[n, m]$ [33].

In the rest of this section we first describe a basic algorithm to compute $\ell_{n,m}$ resiliently with $O(\alpha\delta^2)$ error-recovery time (Section 3.1). In Section 3.2 we show how to decrease this overhead to $O(\alpha\delta^{1+\varepsilon})$, for any arbitrarily small constant $\varepsilon \in (0, 1]$. The resilient computation of a traceback path in C is addressed in Section 3.3.

3.1 A simple algorithm with $O(\alpha\delta^2)$ error-recovery time

Algorithm IterLCS mimics the behavior of the standard non-resilient dynamic programming approach, performing additional work to cope with memory faults. During the computation of table C, we compute fingerprints that allow us to determine whether some memory fault occurred in a given set of memory words. Since detected faults should not force us to recompute the entire table, we divide it into square blocks of side length δ, writing the block boundaries as δ-resilient variables (see Figure 1a). Blocks are smaller (and not necessarily square) on the boundaries, whenever n or m are not divisible by δ: in this case the last row and/or column may not be written reliably. Blocks are computed in column-major order. Upon detection of a failure, we recompute only the current block.

Let $B_{i,j}$ be an internal block (boundary blocks can be treated similarly). Entries of $B_{i,j}$ are processed in column-major order. Column 1 of $B_{i,j}$, together with a Karp-Rabin write fingerprint φ_1, is computed reliably from the δ-resilient variables of neighboring blocks. In general, when computing a column k, for $k > 1$, we already have a write fingerprint φ_{k-1} obtained during the calculation of column $k - 1$. While scanning k top-down, we compute a write fingerprint φ_k and a read fingerprint $\overline{\varphi}_{k-1}$, which is then compared to φ_{k-1}. If the *fingerprint test* succeeds, $\overline{\varphi}_{k-1}$ and φ_{k-1} are discarded and the computation of column $k+1$ begins. Otherwise, the computation of the block is restarted. The prime numbers used for fingerprint computation are chosen uniformly at random at the beginning of the execution of the algorithm and after each fault detection.

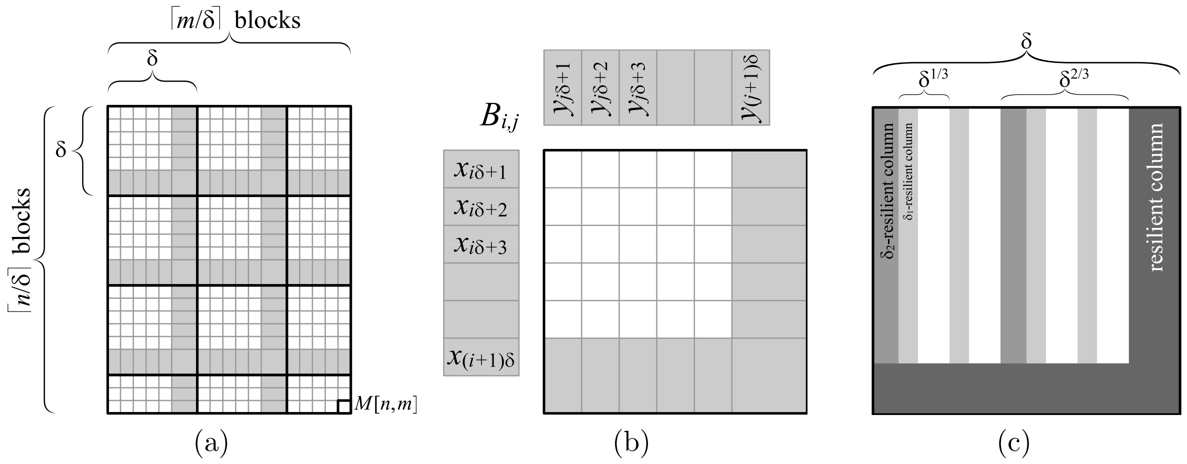

Figure 1: a) Table decomposition and resilient block boundaries (gray color) when $m \geq n \geq \delta$; b) block computation with unreliable input; c) semi-resilient columns in the hierarchical table decomposition for $k = 3$.

Computing table C in $O(mn)$ time requires each comparison of the input symbols in Equation 1 to be performed in constant (amortized) time. As shown in Figure 1b, the input values required to compute block $B_{i,j}$ are $x_{i\delta+1}, \ldots, x_{(i+1)\delta}$ and $y_{j\delta+1}, \ldots, y_{(j+1)\delta}$. The only character of string Y needed in the computation of column k is $y_{j\delta+k}$: we can thus afford to read each $y_{j\delta+k}$ reliably. Conversely, computing each column of $B_{i,j}$ requires all characters $x_{i\delta+1}, \ldots, x_{(i+1)\delta}$. These characters are thus read reliably once, producing a write fingerprint φ_x. Successive read operations are performed unreliably (i.e., considering only one copy) and, similarly to block values, produce a read fingerprint that is compared against φ_x to detect faults in X. If an input fingerprint test fails, all δ-resilient variables $x_{i\delta+1}, \ldots, x_{(i+1)\delta}$ are refreshed using their majority values before restarting the computation of $B_{i,j}$.

We now prove that algorithm IterLCS is correct with high probability and analyze its running time.

Lemma 3. *For any constant $\beta > 0$, algorithm* IterLCS *can be tuned to be correct with probability larger than $1 - 1/m^{\beta}$, when δ is polynomial in m.*

Proof. The algorithm fails whenever a fingerprint test does not detect a memory fault. Consider a block that gets corrupted during its computation and assume for the time being that the boundaries of its neighboring blocks are correct. The reliable read operations guarantee that column 1 is also written correctly. Consider now the concatenation of values written to and read from a column k, and let τ be the difference between those values. Assuming that random numbers used in fingerprint computations are prime, a fingerprint test on column $k+1$ fails to detect a fault in column k if and only if $\varphi_k = \overline{\varphi}_k$ even if $\tau \neq 0$. This happens when $\tau \equiv 0 \mod p$. Since the length of a block column is bounded by $d = \min\{\delta, m\}$ and the word size $w = O(\log m)$, then $\tau \leq 2^{wd} = O(m^d)$. This implies that, for any constant $c > 0$, τ has $O(m)$ prime divisors of value $\Theta(m^c)$. By the prime number theorem, there are $\Theta(m^c/\log m)$ prime numbers of value $\Theta(m^c)$. Hence, a randomly chosen prime $p \in I = [m^{c-1}, m^c]$ divides τ with probability at most $(\log m)/(\sigma m^{c-1}) < 1/(\sigma m^{c-2})$, for some constant $\sigma > 0$. The same argument holds for the fingerprint computed on the input symbols.

Now consider a game with two players. The game is divided into rounds. At each round player 1 (the algorithm) chooses uniformly at random a prime $p \in I$ and player 2 (the adversary) chooses a number $\tau \leq 2^{wd}$. If p divides τ, then player 2 wins, otherwise the next round begins. Player 1 wins if player 2 does not win in α rounds. This game models the behavior of algorithm IterLCS, provided that no composite number is generated instead of a prime. Namely, the probability for algorithm IterLCS of being correct is lower bounded by the probability for player 1 of winning the game. Let p_i and τ_i be the numbers chosen by the two players at round i. Let D_i be the event "player 2 does not win at round i". If player 2 did not win in rounds $1, \ldots, i-1$, the probability of D_i equals the probability that p_i does not divide τ_i. From the discussion above, we have $Pr\left\{D_i | \bigcap_{j=1}^{i-1} D_j\right\} \geq 1 - 1/(\sigma m^{c-2})$. The probability that player 1 wins is equal to $Pr\left\{\bigcap_{i=1}^{\alpha} D_i\right\}$. By the chain rule of conditional probability, we have:

$$Pr\left\{\bigcap_{i=1}^{\alpha} D_i\right\} = \prod_{i=1}^{\alpha} Pr\left\{D_i \mid \bigcap_{j=1}^{i-1} D_j\right\} \geq \left(1 - \frac{1}{\sigma m^{c-2}}\right)^{\alpha} \geq 1 - \frac{\alpha}{\sigma m^{c-2}}$$

We conclude by taking into account the probability for algorithm IterLCS of generating at some round a composite number instead of a prime. It follows from Lemma 1 that, for any constant $\gamma > 0$, the probability that all the α numbers are prime can be made at least as large as $1 - \alpha/m^{\gamma}$. Hence, algorithm IterLCS is correct with probability larger than or equal to $(1 - \alpha/(\sigma m^{c-2}))(1 - \alpha/m^{\gamma})$. Since $\alpha \leq \delta$ is polynomial in m, by appropriately choosing values c and γ the correctness probability can be made larger than $1 - 1/m^{\beta}$, for any constant $\beta > 0$. $\square$

Lemma 4. *The length of a longest common subsequence between two sequences of length n and m (with $m \geq n$) can be correctly computed, with high probability, in $O\left(mn + m\delta + \alpha\delta^2\right)$ worst-case time, when δ is polynomial in m and $\alpha \leq \delta$ is the actual number of faults occurring during the computation.*

Proof. Let us distinguish between *successful* and *unsuccessful block computations*. Unsuccessful block computations account for the time spent by the algorithm computing blocks that are then discarded due to the detection of a memory fault. This time also includes the generation of random primes, except for the first one. Successful block computations account for the remaining time, including the calculation of fingerprints.

Successful computations. Computing the first and last columns of a block requires constantly-many reliable reads or writes for each entry, i.e., $O(\delta^2)$ time. Computing any internal column requires $O(\delta)$ time, since fingerprints can be updated in $O(1)$ amortized time (see Section 2.2). The total time spent in a block is thus $O(\delta^2)$ and the overall time for successful block computations is $O(mn + m\delta)$, because the number of blocks is $\lceil n/\delta \rceil \times \lceil m/\delta \rceil$ and the entire input strings are read reliably.

Unsuccessful computations. Each block recomputation is due to a fingerprint mismatch, that can only be caused by a memory fault (either in the table cells or in some input symbol from string X). Since all block cells are recomputed and input symbols are refreshed upon fault detection, each block recomputation can be charged to a distinct fault. It follows that at most α block computations can be discarded during the entire execution of algorithm IterLCS. Refreshing δ input values and computing a block take time $O(\delta^2)$, yielding overall $O(\alpha\delta^2)$ time for unsuccessful computations. The generation of (at most α) prime numbers does not affect this asymptotic running time (see Lemma 1). $\square$

3.2 Faster error recovery via long distance fingerprints

Our improved algorithm uses an asymmetric decomposition of table C (see Figure 1c) and $k = \lceil 1/\varepsilon \rceil$ different *resiliency levels*, where ε is an arbitrarily small constant in $(0, 1]$. Let $\delta_i = \lceil \delta^{i/k} \rceil$. At each level $i \in [1, k]$, we use δ_i-resilient variables. Consider a given $\delta \times \delta$ block Q. Every δ_i columns, we write a δ_i-resilient column (and all δ_j-resilient versions of this column for $j < i$). In particular, the last column of each internal block is written at all resiliency levels. The non-resilient columns of table C are regarded as having resiliency level 0. During the computation of block Q, for each resiliency level i we keep (in the private memory) the fingerprint of the last δ_i-resilient column. These *long distance* write fingerprints, similarly to those described in Section 3.1, are computed while writing column values. For each resiliency level, we independently select a prime number for computing the fingerprints.

Upon detection of a fault, error recovery is done starting from the last δ_1-resilient column: its values are read by majority computing a read fingerprint at level 1 which is then compared with the corresponding write fingerprint. If the test fails, the recovery starts again from resiliency level 2, i.e., from the last δ_2-resilient column. In general, a level i fingerprint mismatch induces a recovery starting from the last δ_{i+1}-resilient column. When a fingerprint mismatch arises at resiliency level i, we generate a new random prime for level i, we read by majority all values of the last δ_{i+1}-resilient column, and we use these values to refresh all δ_j-resilient versions of this column, for $j \leq i$, recomputing their write fingerprints. Input symbols can be read efficiently by storing δ_i-resilient copies of symbols in X at all resiliency levels and combining long distance fingerprints with the technique described in Section 3.1.

If the adversary inserts no fault, algorithm IterLCS computes the same values for the DP table as any non-resilient LCS algorithm, and is thus correct. In the proof of Lemma 3 we proved correctness with high probability, assuming that the adversary can play α rounds of the game. This is a worst-case assumption also for the improved version of algorithm IterLCS. Indeed, the only fingerprint tests that correspond to a round in the game are those performed on corrupted data. If data is read δ_i-resiliently, more than one fault is needed for the read data to be corrupted. This implies that the additional read operations (and the corresponding fingerprint tests) used to recover computation from δ_i-resilient data have no impact in the failure probability analysis. Lemma 3 thus applies to the improved version of algorithm IterLCS, as well.

Theorem 1. *Algorithm* IterLCS *requires* $O(mn + m\delta + \alpha\delta^{1+\varepsilon})$ *time in the worst case, where* m *and* n *(with* $m \geq n$*) are the lengths of the input sequences,* ε *is an arbitrarily small constant in* $(0, 1]$*, the upper bound* δ *on the number of faults is polynomial in* m*, and* $\alpha \leq \delta$ *is the actual number of faults occurring during the execution.*

Proof. To analyze the running time, similarly to Lemma 4 we distinguish between successful and unsuccessful computations. The running time of successful block computations is not asymptotically affected by the additional fingerprint computations and by the δ_i-resilient columns. Indeed ε is a constant and the additional work, required to compute each δ_i resilient column, is amortized over the computation of $\Omega(\delta_i)$ columns. This settles the mn and $m\delta$ terms in the time complexity formula.

Now consider the unsuccessful computations. Recovery at resiliency level i discards at most $\delta \times \delta_i$ entries of table C and requires computing $O(\delta)$ majority values through δ_i-resilient read operations. Each such operation takes time $O(\delta^{i/k})$, hence restarting the computation takes the same asymptotic time as recomputing the $\delta \times \delta_i$ discarded table entries. The overall time required for a recovery at resiliency level i is thus $O(\delta^{1+i/k})$. A recovery at resiliency level i is due to at least $\delta_{i-1}+1$ errors, either on the input symbols or in table C: a fingerprint mismatch at resiliency level $i-1$ may indeed arise only if the majority value of some δ_{i-1}-resilient variable has been corrupted by the adversary. This gives $O(\max_{1\le i\le k}\{\delta^{1+i/k}/\delta^{(i-1)/k}\}) = O(\delta^{1+1/k})$ amortized time per fault, which proves the theorem since $k = \lceil 1/\varepsilon \rceil$. $\square$

3.3 Tracing back

We now describe how the traceback process can be made resilient. The computation of the LCS proceeds backward block by block starting from cell $C[n, m]$, which we already proved to be with high probability the length of a longest common subsequence. Within each block traversed by the traceback path, we compute the corresponding segment S of the LCS. To bound the traceback cost by $O(\alpha\delta^{1+\varepsilon})$, we proceed incrementally by increasing the resiliency level of S from 1 up to $k = \lceil 1/\varepsilon \rceil$, exploiting the δ_i-resilient columns written by the forward computation. Notice that column fingerprints cannot be exploited, since they are no longer available at this time.

In more details, we regard each segment S as being divided into (at most) $\delta^{1/k}$ subsegments, computed at resiliency level $k-1$. This subdivision proceeds hierarchically, down to resiliency level 1. As a base step, subsegments at resiliency level 0, corresponding to single arcs of the traceback path, are computed from the cells of table C. Except for boundary cases, a subsegment S_i at resiliency level i spans two δ_i-resilient columns and is computed by combining all the $\delta^{1/k}$ subsegments at resiliency level $i-1$ in which S_i is logically divided. Subsegments are read, proceeding right to left, δ_{i-1}-resiliently, and their soundness is verified against the corresponding input symbols, which are read δ_i-resiliently: we call this a *consistency check*. A consistency check at resiliency level i fails for a subsequence symbol z, obtained by tracing back over cell $C[h, k]$, if the δ_i-resilient read operation of either x_h of y_k does not return symbol z. During this process, the resiliency level each subsegment composing S_i is is increased from δ_{i-1} to δ_i.

If a consistency check fails at a given cell c some data has been corrupted. The algorithm attempts an optimistic error-recovery: assuming that the data-corruption identified by δ_i-resilient read operations was indeed due the corruption of the majority values of δ_{i-1}-resilient variables, the algorithm rolls back the computation of the current subsegment S_i and proceed to recovery as follows: the input symbols corresponding to the row and column of c are refreshed and, if either endpoint of S lies on a resilient row, the corresponding cell is also refreshed (these refreshes are done at full resiliency). The recovery then starts from the closest δ_i-resilient column to the left of c: all δ_j-resilient versions of this column, for $j < i$, are refreshed from the δ_i-resilient values (read by majority) and the block slice is recomputed by applying the improved version of algorithm IterLCS. At the end of the slice computation, we check if the new values stored on the closest δ_i-resilient column to the right of c match the old ones: if this is not the case, recovery restarts at resiliency level $i+1$. Notice that at this point we have no guarantee that the computation of the slice used correct data. If data corruption was due to corrupted δ_{i+j}-resilient variables (for $j \ge 0$), the table elements of the recomputed slice could have been miscalculated, however this data corruption would be identified when checking δ_{i+j}-resilient subsegments against δ_{i+j+1}-resilient variables. At the end of the recovery phase, the computation of S_i restarts from subsegments at resiliency level 0.

When the computation of a subsegment S_i is completed, the algorithm checks if the length of the subsegment matches the difference between the cell values in table C corresponding to its endpoints. These cells lie on δ_i-resilient columns (or on resilient rows on the block boundaries) and their values are read δ_i-resiliently. Apart from refreshing the input values, upon detection of a mismatch recovery is performed as described above.

We remark that, if the forward computation failed (which is a low-probability event), the traceback algorithm may find inconsistencies in δ-resilient variables: in this case the algorithm terminates without reconstructing the LCS.

Theorem 2. *Let C be the table computed by algorithm* IterLCS. *A common subsequence of length $C[n, m]$, if any, can be reconstructed from C with high probability in time $O(m\delta + \alpha\delta^{1+\varepsilon})$.*

Proof. The correctness of each symbol included in the common subsequence is verified at all resiliency levels by consistency checks. The length of each segment is also verified against the values reliably stored in the block boundaries of table C. Since during error recovery no δ_{i+j}-resilient value is modified starting from δ_i-resilient reads, for any $j \geq 0$, memory faults are never propagated to higher levels of resiliency. This implies that a subsegment at resiliency level i may be wrong only if at least $\delta^{i/k} + 1$ memory faults occurred. Since the adversary can insert at most δ faults, the δ-resilient common subsequence, if returned, is correct and has length $C[n, m]$. This subsequence may not be optimal or the algorithm may not be able to reconstruct it only if a fingerprint test failed to detect a memory fault, which is a low probability event (see Lemma 3). We now analyze the running time.

Successful computations. The time spent to combine all δ_{k-1}-resilient subsegments is asymptotically higher than the time spent at all lower resiliency levels. This time is $O(m\delta)$, because the traceback path traverses $O((n + m)/\delta)$ blocks and each block costs time $O(\delta^2)$.

Unsuccessful computations. Consider a consistency check failure arising while computing a subsegment at resiliency level $i + 1$. Such a failure is due to at least $\delta_i + 1$ faults and costs $O(\delta^{1+(i+1)/k})$ time for recomputing $\delta \times \delta_{i+1}$ table cells. If the δ_{i+1}-resilient column used for recovery was correct, detected errors are removed from the table with an amortized $O(\delta^{1+1/k})$ cost per memory fault. If the δ_{i+1}-resilient column used for recovery was corrupted, the adversary must have inserted at least $\delta_{i+1} + 1$ faults and the recomputed cells of the table may still contain incorrect values after recovery. Two cases may happen: either the forward recomputation of the table slice finds an inconsistency with the following δ_{i+1}-resilient column, or no inconsistency is detected and a possibly wrong δ_{i+1}-resilient subsegment is computed. In both cases, the number of memory faults inserted by the adversary is large enough to obtain an amortized $O(\delta^{1+1/k})$ cost per fault, with recovery done at a higher resiliency level. □

4 Recursive local-dependency dynamic programming

Algorithm IterLCS presented in Section 3 can be applied to all problems with local dependencies, but incurs in a large number of cache misses. In this section we show that, if we can afford a private memory of logarithmic size, we can achieve both resiliency and cache-efficiency. Our approach hinges upon a recursive framework for dynamic programming, introduced in [17, 18], that we briefly describe in Section 4.1 focusing on the LCS problem. We assume that both input sequences have length n and, without loss of generality, that n and δ are powers of two (we will then show how to remove the former assumption).

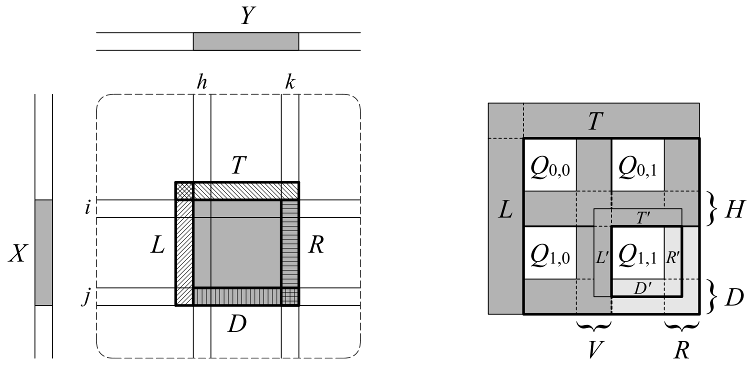

Figure 2: (a) Boundaries of a subtable and input projections; (b) subquadrants and auxiliary vectors used by function Boundary.

4.1 Cache-efficient dynamic programming

As shown in Figure 2a, for any subtable Q of the DP table C we can naturally identify its *left*, *right*, *top*, and *down boundaries* (denoted by L, R, T, and D) and two *projections* of the input sequences X and Y on Q. The algorithm presented in [17, 18] 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 through table C by recursively finding its fragments through the traversed quadrants. Given the input boundaries of a quadrant Q and the path entry point on the output boundaries of Q, it first computes the input boundaries of at most three subquadrants of Q (by invoking function Boundary), and then calls itself recursively on each of the at most three traversed subquadrants.

A longest common subsequence of X and Y can be computed by initializing the left and top boundaries of table C with all zeroes and invoking function Traceback-Path with entry point $C[n, n]$. We refer to [17, 18] for a detailed description of the implementation details. As proved in [17], Boundary and Traceback-Path compute a longest common subsequence of any two n-length sequences in $O(n^2)$ time, using linear space, and incurring $O(1 + n/B + n^2/(BM))$ cache misses, where M and B are the cache and the cache line sizes, respectively. In [19], 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.

4.2 A resilient cache-efficient algorithm

We now describe how functions Boundary and Traceback-Path can be made resilient. Similarly to the iterative approach of Section 3, we combine data replication and read/write fingerprints to bound the cost of error recovery and to enable fault detection, respectively. Input and output of the recursive calls are first stored at *resiliency level* δ, i.e., through δ-resilient variables. However, since maintaining full resiliency throughout the recursion would increase the running time by a factor of δ, we halve the resiliency level at each recursive call exploiting semi-resilient variables associated with fingerprints. For each recursion level d, we use a distinct prime number p_d. Notice that both primes and the call stack can be stored in private memory since the recursion depth is $O(\log n)$.

We start by describing two auxiliary functions that extract and merge vector segments, changing their resiliency level and updating their fingerprints:

- 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.
- Function `extract` takes a small vector out of a larger one. Given as input a vector A of length k stored at resiliency level r, a read fingerprint $\overline{\varphi}_A$ computed on a prefix of A of length $s \leq k$, and three integer values s, $k' \leq k - s$, and $r' \leq r$, it extracts from A a segment A' of length k', starting from position $s + 1$. Values from A are read by majority on the $2r+1$ available copies and are written in A' at resiliency level r'. At the same time, $\overline{\varphi}_A$ is updated to include the additional k' items that have been just read and a write fingerprint $\varphi_{A'}$ is computed from scratch according to the majority values read from A.

Resilient boundary. When called on a quadrant Q at recursion depth d, function Boundary receives, together with L, T, and the input projections, the write fingerprints corresponding to these vectors. Accordingly, it returns the output boundaries associated with two write fingerprints. All the input and output vectors have the same length $n_d = n/2^d$ and are stored at resiliency level $\delta_d = \max\{\delta/2^d, 1\}$. As in [18], Boundary logically splits Q into four subquadrants ($Q_{0,0}$, $Q_{0,1}$, $Q_{1,0}$, and $Q_{1,1}$ in row-major order) and recursively solves the subproblems. The recursive calls operate at resiliency level $\delta_{d+1} = \max\{\delta/2^{d+1}, 1\}$ on input vectors L', T', X', and Y', all of length n_{d+1}, and produce D' and R' as output. The call on quadrant Q makes use of two auxiliary (horizontal and vertical) vectors H and V of length n_d, resiliency δ_d, associated with their write and read fingerprints (see Figure 2b).

During the recursive call on quadrant Q, function Boundary computes subquadrants in row-major order according to the following pattern: 1) input extraction; 2) fingerprint tests; 3) recursive call and output insertion; 4) error recovery, if needed. Steps 1, 2, and 3 slightly differ depending on the subquadrant on which the recursive call is performed. Namely,

$$L' \text{ and } T' \text{ are extracted from: } \begin{cases} L \text{ and } T & \text{on quadrant } Q_{0,0} \\ V \text{ and } T & \text{on quadrant } Q_{0,1} \\ L \text{ and } H & \text{on quadrant } Q_{1,0} \\ V \text{ and } H & \text{on quadrant } Q_{1,1} \end{cases}$$

The `extract` (step 1) also produces write fingerprints $\varphi_{L'}$ and $\varphi_{T'}$, which are passed to the recursive call, and updates - depending on the quadrant - read fingerprints $\overline{\varphi}_L$, $\overline{\varphi}_T$, $\overline{\varphi}_H$, and $\overline{\varphi}_V$, which are then used in the fingerprint tests described below. X' and Y' are always extracted from the projections of X and Y on Q, respectively. Moreover, X' is extracted only on $Q_{0,0}$ and $Q_{1,0}$, since subquadrants are processed in row-major order. Fingerprint tests take place as soon as a vector of length n_d has been read completely: due to the row-major processing order, this happens on subquadrant $Q_{0,1}$ for vectors Y and T, on $Q_{1,0}$ for vectors X and L, and on $Q_{1,1}$ for the auxiliary vectors H and V (see Figure 2b for the layout). Moreover, since input is extracted from sequence Y twice (and only once from sequence X), on quadrant $Q_{1,1}$ we also check the correctness of the second extraction from Y. In view of the above considerations, fingerprint tests (step 2) are as follows:

$$\begin{cases} \text{no test} & \text{on quadrant } Q_{0,0} \\ \overline{\varphi}_Y = \varphi_Y \text{ and } \overline{\varphi}_T = \varphi_T & \text{on quadrant } Q_{0,1} \\ \overline{\varphi}_X = \varphi_X \text{ and } \overline{\varphi}_L = \varphi_L & \text{on quadrant } Q_{1,0} \\ \overline{\varphi}_Y = \varphi_Y,\ \overline{\varphi}_H = \varphi_H, \text{ and } \overline{\varphi}_V = \varphi_V & \text{on quadrant } Q_{1,1} \end{cases}$$

A mismatch on any of the above tests implies failure of the recursive call at depth d and will be handled by higher calls (see step 4 discussed below). Notice that fingerprint tests are performed lazily once vectors at resiliency level δ_d have been read entirely. With this approach, it may happen that a recursive call operates on corrupted input data, since no check establishes the correctness of the extracted subsequences L', T', X', and Y' until computation of quadrant $Q_{1,1}$. While in this algorithm lazy and eager fingerprint tests would be equivalent, the lazy approach described here turns out to be more efficient in the extension described in Section 6.

Upon success of a recursive call, its output vectors are appropriately merged, increasing their resiliency level (step 3). Namely,

$$D' \text{ and } R' \text{ are inserted into: } \begin{cases} H \text{ and } V & \text{on quadrant } Q_{0,0} \\ H \text{ and } R & \text{on quadrant } Q_{0,1} \\ D \text{ and } V & \text{on quadrant } Q_{1,0} \\ D \text{ and } R & \text{on quadrant } Q_{1,1} \end{cases}$$

Error recovery (step 4) is performed whenever in step 3 the recursive call fails or a fingerprint mismatch (on δ_{d+1}-resilient data) arises during an `insert`: in such cases, all data at resiliency level δ_{d+1} are discarded, the prime number p_{d+1} is renewed, and the subquadrant computation restarts from step 1. A backup copy of each read and write fingerprint is needed to restore the computation state. Since each of the two projections of X is extracted only once (on subquadrants $Q_{0,0}$ and $Q_{1,0}$, respectively), input extraction upon failure of a recursive call on $Q_{0,1}$ or $Q_{1,1}$ must be handled appropriately: an additional fingerprint of X' based on prime number p_d is sufficient to check that a repeated extraction from X, needed to recompute the input for quadrants $Q_{0,1}$ or $Q_{1,1}$, is consistent with the original extraction. A fingerprint mismatch during this check implies corruption of δ_d-resilient variables, thus causing the failure of the recursive call at depth d.

Resilient traceback path. The resilient implementation of Traceback-Path computes the traceback path segment π traversing a quadrant Q, along with its write fingerprint φ_π. For calls at recursion depth d, both π and the input vectors (of length n_d) are stored at resiliency level δ_d, while the entry point of π in Q is stored in private memory. Traceback-Path first performs a forward computation by calling resilient Boundary to obtain vectors H and V, stored at resiliency level δ_d. For instance, assume that the entry point of π in Q belongs to subquadrant $Q_{1,1}$ (the other cases, where the entry point is in $Q_{0,1}$ or $Q_{1,0}$, can be treated similarly): in this scenario TRACEBACK-PATH invokes BOUNDARY on $Q_{0,0}$, $Q_{0,1}$, and $Q_{1,0}$ to obtain both H and V. Once all vectors are available at the proper resiliency level, TRACEBACK-PATH computes π backward by calling itself on (at most three) subquadrants intersected by π: in the previous example, the first recursive call is on $Q_{1,1}$, while the other calls depend on which subquadrants are traversed by π. Segments of π in the traversed subquadrants, obtained by the recursive calls and stored at resiliency level δ_{d+1}, are eventually stitched and increased in resiliency using function `insert`. Fingerprint mismatches at resiliency level δ_d cause the current call of TRACEBACK-PATH to fail. Fingerprint mismatches at resiliency level δ_{d+1} and failed calls cause data at resiliency level δ_{d+1} to be discarded and to repeat the subproblem computation.

Since the backward access pattern to H, V, L, and T is inverted with respect to the order in which data are written by function BOUNDARY in the forward computation, we exploit amplified fingerprints in reversed order as described in Section 2.3. These fingerprints can be updated in constant time as shown in Lemma 2. We also take care of forcing the algorithm to read vector segments corresponding to quadrants not intersected by the traceback path in order to correctly update the amplified read fingerprints.

Extension to sequences of different length. Input sequences of different lengths m and n, with $m \geq n$, are handled by splitting the longer sequence Y into $\lceil m/n \rceil$ segments, each (but the last one) of length n. A forward computation is first performed by applying function BOUNDARY $\lceil m/n \rceil$ times, storing all the output boundaries R in δ-resilient variables. Function TRACEBACK-PATH is then invoked $\lceil m/n \rceil$ times, starting from $C[n, m]$. In general, the entry point of a subproblem is obtained from the already computed suffix of the traceback path. We call the entire algorithm RECLCS.

4.3 Analysis

Before analyzing running time and cache complexity of algorithm RECLCS, we prove its correctness on any two sequences X and Y of length n and m, with $m \geq n$.

Lemma 5. *For any constant $\varepsilon > 0$, algorithm* RECLCS *can be tuned to be correct with probability larger than $1 - 1/m^{\varepsilon}$, when δ is polynomial in m.*

Proof. If no memory fault is introduced by the adversary, the correctness of the algorithm follows from [17]. We now prove that all faults inserted by the adversary are detected and corrected by algorithm RECLCS with high probability. The initial calls of function TRACEBACK-PATH have resiliency $\delta_0 = \delta$: this implies that majority values cannot be corrupted and computation is never aborted. The algorithm independently selects a prime number for each level of recursion and performs a new selection every time some computation is repeated due to a fingerprint mismatch. Hence, at most $\lceil \log n \rceil + \alpha$ selections can be performed during the execution and, by Lemma 1, all selected numbers in $[m^{k-1}, m^k]$ are prime with probability at least $1 - (\log n + \alpha)/m^{\gamma}$, for any constants k and γ. Similarly to the proof of Lemma 3, each fingerprint test fails to identify a corrupted variable with probability at most $1/(\sigma m^{k-2})$, for some positive constant σ, provided that all selected numbers are primes. Since no more than α variables can be corrupted by the adversary during the execution, the overall probability of detecting all faults is at least $(1-\alpha/(\sigma m^{k-2}))(1-(\log n+\alpha)/m^{\gamma})$. By appropriately choosing k and γ, this probability can be made larger than $1 - 1/m^{\varepsilon}$, for any $\varepsilon > 0$. □

Theorem 3. *Algorithm* RecLCS *solves the longest common subsequence correctly with high probability. It requires $O(mn + \delta m \log n)$ time and incurs $O(mn/(BM) + \delta m \log n/B)$ cache misses in the worst case, where m and n (with $m \geq n$) are the lengths of the input sequences, M and B are the cache and the cache line sizes, and the upper bound δ on the number of faults is polynomial in m.*

Proof. Correctness is proved in Lemma 5. Since algorithm RecLCS consists of $\lceil m/n \rceil$ calls of Traceback-Path with input size n, it is sufficient to analyze its running time on strings of equal length n, multiplying all bounds by $\lceil m/n \rceil$. Provided that no computation is repeated, the running time of Boundary is given by the following recurrence:

$$T_B(n, \delta) = \begin{cases} \Theta(\delta + 1) & \text{if } n \leq 1 \\ 4T_B(n/2, \delta/2) + \Theta(n(\delta + 1)) & \text{otherwise} \end{cases}$$

which results in a worst-case $\Theta(n^2 + \delta n \log n)$ time.

Inducing a recomputation at level $1 \leq i \leq k$, with $k = \log(\min\{n, \delta\})$, requires at least $\delta/2^i$ faults (there cannot be recomputations at level $i = 0$ since boundaries are δ-resilient). Hence, at most $\alpha 2^i/\delta$ recomputations can be induced. Since there are 4^i subproblems at level i, the following summation bounds from above the time spent in unsuccessful computations:

$$\sum_{i=1}^{k} \frac{\alpha 2^i}{\delta} \frac{T_B\left(n, \delta\right)}{4^i} \leq T_B\left(n, \delta\right) \sum_{i=1}^{k} \frac{1}{2^i} \leq T_B\left(n, \delta\right)$$

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 $T_B\left(n, \delta\right)/4^k$. Hence, the time spent in unsuccessful computations at levels $j \in [k+1, \log n]$ is upper bounded by: $\alpha T_B\left(n, \delta\right)/\delta^2 < T_B\left(n, \delta\right)$. In all cases, the time spent in unsuccessful computations does not exceed the time spent in successful computations.

Similarly, unsuccessful computations of function Traceback-Path do not exceed the running time of the successful ones. This time is given by the following recurrence:

$$T_P(n, \delta) = \begin{cases} \Theta(\delta + 1) & \text{if } n \leq 1 \\ 3T_P(n/2, \delta/2) + 4T_B(n/2, \delta/2) + \Theta(n(\delta + 1)) & \text{otherwise} \end{cases}$$

which solves to $\Theta(n^2 + \delta n \log n)$.

Analogous considerations apply to the number of cache misses. Provided that no computation is repeated, the cache complexities $Q_B(n, \delta)$ of function Boundary and $Q_P(n, \delta)$ of function Traceback-Path are given by:

$$Q_B(n, \delta) = \begin{cases} O(n(\delta + 1)/B + 1) & \text{if } n(\delta + 1) \leq M \text{ or } n \leq 1 \\ 4Q_B(n/2, \delta/2) + O(n(\delta + 1)/B) & \text{otherwise} \end{cases}$$

and

$$Q_P(n, \delta) = \begin{cases} O(n(\delta + 1)/B) & \text{if } n(\delta + 1) \leq M \text{ or } n \leq 1 \\ 3Q_P(n/2, \delta/2) + 4Q_B(n/2, \delta/2) + \Theta(n(\delta + 1)/B) & \text{otherwise} \end{cases}$$

Both recurrences solve to $\Theta(n^2/(BM) + \delta n \log n/B)$ and unsuccessful computations do not affect this bound. □

```
Input: $n \times n$ matrix $A$, function $f : \mathcal{S}^4 \to \mathcal{S}$, set $\Sigma_f$ of triplets $\langle i,j,k \rangle$, with $i,j,k \in [0,n)$.
Output: transformation of $A$ defined by $f$ and $\Sigma_f$.
1. for $k \leftarrow 0$ to $n-1$ do
2.    for $i \leftarrow 0$ to $n-1$ do
3.       for $j \leftarrow 0$ to $n-1$ do
4.          if $\langle i,j,k \rangle \in \Sigma_f$ then $A[i,j] \leftarrow f(A[i,j], A[i,k], A[k,j], A[k,k])$;
```

Figure 3: Gaussian Elimination Paradigm (GEP).

5 Non-local dynamic programming

The techniques described in Section 4 can be used to extend significantly the class of problems that are efficiently solvable in the presence of memory faults. In this section we present resilient (and cache-efficient) algorithms for non-local problems that fit in the Gaussian Elimination Paradigm [20] and for the Fast Fourier Transform.

5.1 Gaussian Elimination Paradigm

Let A be an $n \times n$ matrix with entries from an arbitrary domain $\mathcal{S}$, and let $f : \mathcal{S}^4 \to \mathcal{S}$ be an arbitrary function. The computation in Figure 3 is known as *Gaussian Elimination Paradigm* (GEP) [20]: the algorithm modifies A by applying a given set of *updates*, denoted by $\langle i,j,k \rangle$ for $i,j,k \in [0,n)$. We let Σ_f denote the set of updates performed by the algorithm. Many problems can be solved by GEP, including Floyd-Warshall's all-pairs shortest path, Gaussian Elimination, LU decomposition without pivoting, and matrix multiplication. I-GEP [20] is a subclass of GEP including all the aforementioned problems, where the updates to matrix A can be appropriately reordered without compromising correctness of the final result, even if intermediate states are different.

The cache-oblivious algorithm for I-GEP provided in [20] incurs $O(n^3/B\sqrt{M})$ cache misses and extends to parallel and multicore machines [19, 21]. The algorithm consists of four recursive functions $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$ that take as input four matrices $X = A[I,J]$, $U = A[I,K]$, $V = A[K,J]$ and $W = A[K,K]$, where I,J,K denote suitable intervals in $[0,n)$. These functions differ according to the amount of overlap among X, U, V, and W: function $\mathcal{A}$ assumes completely overlapping matrices (i.e., $X = V = U = W$), $\mathcal{B}$ (resp., $\mathcal{C}$) assumes that $X = V$ and $U = W$ (resp., $X = U$ and $V = W$), and $\mathcal{D}$ assumes completely non-overlapping matrices. Other types of overlapping are not possible. Each function modifies matrix X by means of eight mutually recursive calls (using suitable quadrants of X, U, V, and W as inputs) performing updates $\langle i,j,k \rangle$ in Σ_f such that $A[i,j] \in X$, $A[i,k] \in U$, $A[k,j] \in V$, and $A[k,k] \in W$. The initial call is $\mathcal{A}(A,A,A,A)$ and base cases arise on input matrices of size 1. We refer to [20] for the proof of the correctness of the algorithm and for a more detailed pseudocode.

A resilient cache-efficient algorithm for GEP. The resilient algorithm RecGEP inherits the recursive organization of the aforementioned I-GEP algorithm, pairing it with data replication and read/write fingerprints. For each of the $\log n$ recursion levels, we use a distinct prime number to compute fingerprints of the involved matrices (assuming a private memory of logarithmic size). We recall that the fingerprint of a matrix is computed as explained in Section 2.2.

As in Section 4.2, let $\delta_d = \max\{\delta/2^d, 1\}$ and $n_d = n/2^d$, for $0 \le d \le \log n$. At recursion depth d, RecGEP receives the $n_d \times n_d$ matrices X, U, V, and W stored δ_d-resiliently and their write fingerprints computed using prime p_d. It then updates X together with its write fingerprint, explicitly checking that each recursive call receives correct input quadrants. (We remark that fault detection could be made lazy by exploiting amplified fingerprints: this allows reducing the private memory size and can be done using techniques similar to those in Section 4.) In more details, the algorithm first reads δ_d-resiliently all the input matrices, producing for each matrix a read fingerprint and four write fingerprints, one per quadrant. All these fingerprints are based on prime p_d and the correctness of read data is verified via fingerprint tests against the input write fingerprints. Then, RecGEP recursively solves eight subproblems in two rounds, as specified in I-GEP [20]. For each call:

1. it extracts the δ_{d+1}-resilient quadrants, computing their write fingerprints based on prime p_{d+1} and their read fingerprints based on prime p_d;
2. it tests whether the latter fingerprints match the write fingerprints computed at the beginning. If a fingerprint test fails (i.e., a δ_d-resilient variable has been corrupted), the call at level d aborts;
3. it performs a recursive call to solve the subproblem;
4. it replaces the suitable quadrant of X (together with its write fingerprint) with the updated quadrant returned by the call at step 3, increasing its resiliency level to δ_d;
5. it performs error recovery whenever the call of step 3 fails or a fault is recognized in step 4: in this case, prime p_{d+1} is renewed and the subproblem computation is restarted.

Theorem 4. *Algorithm* RecGEP *solves $n\times n$ I-GEP problems correctly with high probability. It requires $O(n^3 + \delta n^2 \log n)$ time and incurs $O(n^3/(B\sqrt{M}) + \delta n^2 \log n/B)$ cache misses in the worst case, where M is the cache size, B is the cache line size, and the upper bound δ on the number of faults is polynomial in n.*

Proof. If there are no faults, RecGEP is equivalent to the non-resilient algorithm. Otherwise, memory faults are fixed by exploiting data replication. Correctness follows from the fact that faults are detected with high probability via fingerprints (this can be proved as in Lemma 5).

If there is no recomputation, the algorithm performs a constant number of scans of the input matrices to prepare subproblem inputs and fingerprints. Hence, its running time $T_{\texttt{RecGEP}}(n, \delta)$ is upper bounded by the following recurrence:

$$T_{\texttt{RecGEP}}(n,\delta) = \begin{cases} \Theta(\delta+1) & \text{if } n \le 1 \\ 8T_{\texttt{RecGEP}}(n/2, \delta/2) + \Theta(n^2\delta + n^2) & \text{otherwise} \end{cases}$$

which solves to $\Theta(n^3 + \delta n^2 \log n)$. Inducing a recomputation at level $1 \le i \le k$, with $k = \log(\min\{n, \delta\})$, requires at least $\delta/2^i$ faults and costs at most $T_{\texttt{RecGEP}}(n,\delta)/8^i + \Theta(n^2\delta/4^i)$, where the second term is the cost of input extraction at level $i-1$. At most $\alpha 2^i/\delta$ recomputations can be therefore induced at level i. The following summation bounds from above the time spent in unsuccessful computations:

$$\sum_{i=1}^{k} \frac{\alpha 2^i}{\delta}\left(\frac{T_{\texttt{RecGEP}}(n,\delta)}{8^i} + \Theta\left(\frac{n^2\delta}{4^i}\right)\right) \le \sum_{i=1}^{k} \frac{\alpha}{\delta}\left(\frac{T_{\texttt{RecGEP}}(n,\delta)}{4^i} + \Theta\left(\frac{n^2\delta}{2^i}\right)\right) < \Theta(T_{\texttt{RecGEP}}(n,\delta))$$

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 $T_{\texttt{RecGEP}}(n,\delta)/8^k$. Hence, the time spent in unsuccessful computations at levels $j \in [k+1, \log n]$ is upper bounded by: $\alpha T_{\texttt{RecGEP}}(n,\delta)/8^k \leq \alpha T_{\texttt{RecGEP}}(n,\delta)/\delta^3 < T_{\texttt{RecGEP}}(n,\delta)$. In all cases, the time spent in unsuccessful computations does not exceed the time spent in the successful ones.

Similarly, the cache complexity of RecGEP without recomputation is upper bounded by the following recurrence:

$$Q_{\texttt{RecGEP}}(n,\delta) = \begin{cases} \Theta(n^2\delta/B + 1) & \text{if } n^2\max\{\delta,1\} \leq M \\ 8Q_{\texttt{RecGEP}}(n/2,\delta/2) + \Theta(n^2\delta/B + n^2/B) & \text{otherwise} \end{cases}$$

which solves to $O(n^3/(B\sqrt{M}) + \delta n^2 \log n/B + 1)$. Unsuccessful computations do not affect this bound. □

5.2 Fast Fourier Transform

The techniques to add resiliency to recursive algorithms described above also apply to the Fast Fourier Transform. Building on the cache-oblivious FFT algorithm in [30], we derive a resilient (cache-oblivious) algorithm, named RecFFT, which returns the correct result with high probability, using a private memory of size $\Theta(\log\log n)$. The cache-oblivious algorithm computes recursively the FFT of an input vector A of size n by performing two rounds of recursive calls: in each round, vector A is permuted (according to a matrix transposition) and then split into $\sqrt{n}$ segments of size $\sqrt{n}$. The FFT of each segment is then computed recursively. The recursion depth is therefore $\Theta(\log\log n)$.

In the description of the resilient implementation, for the sake of simplicity we assume $\log n$ and $\log\delta$ to be powers of two (RecFFT can be easily generalized to arbitrary values of n and δ). Input vectors at recursion level d, with $0 \leq d \leq \log\log n$, have size $n_d = n^{1/2^d}$, are stored at resiliency level $\delta_d = \max\{\delta^{1/2^d}, 1\}$, and are associated with a write fingerprint computed with prime p_d. When subproblems are extracted from the input vector, their resiliency level is decreased to δ_{d+1} and their correctness is checked lazily at the end of each round. If a subproblem fails, the prime p_{d+1} is renewed, the extraction of the subproblem input is repeated (together with the computation of its write fingerprint), and the subproblem computation is restarted. Matrix transposition can be performed using the cache-oblivious algorithm in [30], which can be easily extended in order to return the write fingerprint of the new vector and to update the read fingerprint of the input vector (these two fingerprints are computed on different permutations of the same values).

Theorem 5. *Algorithm* RecFFT *correctly computes the FFT of a vector of size n with high probability. It requires $O(n\log n + \delta n)$ time and incurs $O((n\log_M n)/B + \delta n/B + 1)$ cache misses in the worst case, where M is the cache size, B is the cache line size, and $\delta \leq n$ is an upper bound on the number of faults.*

Proof. Correctness follows from the same arguments used in the previous sections. If there is no recomputation, the running time of RecFFT is upper bounded by

$$T_{\texttt{RecFFT}}(n,\delta) = \begin{cases} \Theta(\delta + 1) & \text{if } n \leq 4 \\ 2\sqrt{n}T_{\texttt{RecGEP}}(\sqrt{n},\sqrt{\delta}) + \Theta(\delta n + n) & \text{otherwise} \end{cases}$$

which solves to $\Theta(n\log n+\delta n)$. Let $k=\log\log\delta$. Inducing a recomputation at level $1\le i\le k$ requires at least $\delta^{1/2^i}$ faults. Hence, at most $\alpha/\delta^{1/2^i}$ recomputations can be induced at level i. Since the cost of recomputing a subproblem is at most $T_{\texttt{RecFFT}}(n,\delta)/(2^i n^{1-1/2^i})+\Theta(n^{1/2^i}\delta^{1/2^{i-1}})$, where the second term is due to input extraction at the $(i-1)$-st recursive level, we have

$$\sum_{i=1}^{k}\frac{\alpha}{\delta^{1/2^i}}\left(\frac{T_{\texttt{RecFFT}}(n,\delta)}{2^i n^{1-1/2^i}}+\Theta(n^{1/2^i}\delta^{1/2^{i-1}})\right)=O\left(\sum_{i=1}^{k}\frac{\alpha n}{\delta}\frac{\log n}{2^i}+\alpha(n\delta)^{1/2^i}\right).$$

This is in $O(T_{\texttt{RecFFT}}(n,\delta))$, because $\sum_{i=1}^{k}\alpha(n\delta)^{1/2^i}\le\sum_{i=0}^{k-1}\alpha(n)^{1/2^i}=O(\alpha n)$. Recursive calls at levels deeper than k are all done at resiliency level 1, since $\delta\le n$. The adversary can induce up to α recomputations at these levels, each having cost in $O(T_{\texttt{RecFFT}}(n^{1/\log\delta},1))$. Hence, the time spent in unsuccessful computations at levels $j\in[k+1,\log\log n]$ is in $O(\alpha T_{\texttt{RecFFT}}(n^{1/\log\delta},1))$. This solves to $O(\alpha n^{1/\log\delta}\log_\delta n)$, which is in $O(n\log n)$. In all cases, successful computations dominate the running time.

Similar arguments apply to the cache complexity, where the number of cache misses due to successful computations is given by:

$$Q_{\texttt{RecFFT}}(n,\delta)=\begin{cases}\Theta(n\delta/B+1) & \text{if } n\max\{\delta,1\}\le M\\ 2\sqrt{n}Q_{\texttt{RecGEP}}(\sqrt{n},\sqrt{\delta})+\Theta(n\delta/B+n/B) & \text{otherwise}\end{cases}$$

which solves to $O((n\log_M n)/B+\delta n/B+1)$. □

6 Bounding the private memory

In this section we show how to remove the logarithmic private memory assumption of Section 4, obtaining an algorithm that is parametric in the private memory size P (similar techniques can be also adapted to the algorithms of Section 5). Using rather standard techniques, we reduce the recursion depth so that stack, primes, and fingerprints fit in P memory words (Section 6.1). The most challenging problem with a small private memory, however, is to maintain cache-efficiency: we address temporal and spatial locality issues in Section 6.2 and Section 6.3, respectively. A full analysis is given in Section 6.4. We assume both input sequences X and Y to be of length n: this assumption can be removed as in Section 4.2.

6.1 Shallow recursion tree with lazy fault detection

Let v be the number of local variables used by algorithm RecLCS and let ρ be the largest integer such that $v\rho\le P$. Notice that $\rho=\Theta(P)$. We modify RecLCS by splitting a quadrant, at each call, into $\lambda\times\lambda$ subquadrants, where $\lambda=\lceil n^{1/\rho}\rceil$: this guarantees that private data fits into P memory words since the recursion depth is $\rho=\Theta(\log_\lambda n)=\Theta(P)$. We index quadrants by pairs $\langle i,j\rangle\in[0,\lambda-1]\times[0,\lambda-1]$ and we say that quadrant $\langle i,j\rangle$ has row i and column j. A quadrant is *internal* if and only if $1\le i,j<\lambda-1$. Besides the subdivision into λ^2 quadrants, the resilient implementation of function Boundary is modified as follows:

- Input and output vectors used in a call at recursion depth d have length $n_d=\lceil n/\lambda^d\rceil$ and are stored in the unreliable memory at resiliency level $\delta_d=\lceil\delta/\lambda^d\rceil$. Auxiliary vectors

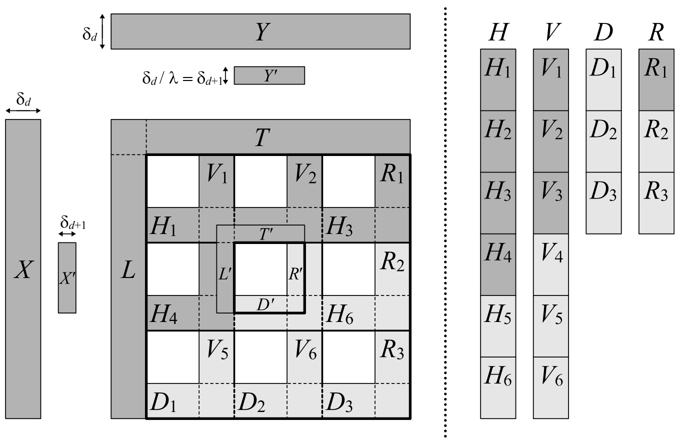

Figure 4: Quadrants, boundaries, and auxiliary vectors in the $\lambda \times \lambda$ table decomposition.

H and V also have resiliency δ_d: they are obtained by juxtaposing the down boundaries $H_{\langle i,j \rangle}$ of quadrants $\langle i, j \rangle$ and the right boundaries of quadrants $\langle i, j \rangle$, respectively. Hence, their length is λn_d. Notice that the last row of H and the last column of V correspond to vectors D and R, respectively (see Figure 4). Each vector has its own (private) read and write fingerprints.

- The λ^2 subproblems are recursively solved in row-major order. Each subquadrant is processed according to the same pattern described in Section 4.2. Input extraction (from L/V and T/H) and output insertion (into R/V and D/H) can be appropriately modified depending on the row and column indexes of the subquadrant. The projection of X is extracted once per row: similarly to Section 4.2, this requires an additional backup fingerprint based on prime number p_d to verify the correctness of repeated extractions needed for error recovery.

- With non-constant λ and $O(1)$ fingerprints per recursion level, checking the input correctness for all the recursive calls would result in non-negligible time overhead: hence, detecting faults lazily, i.e., performing fingerprint tests related to each vector only when the vector has been completely scanned by the algorithm, becomes mandatory with the $\lambda \times \lambda$ subdivision. The following fingerprint tests are thus (lazily) performed, depending on the quadrant:

  – $\overline{\varphi}_V = \varphi_V$ and $\overline{\varphi}_H = \varphi_H$ must hold if $\langle i, j \rangle = \langle \lambda - 1, \lambda - 1 \rangle$;
  – $\overline{\varphi}_L = \varphi_L$ and $\overline{\varphi}_T = \varphi_T$ must hold if $\langle i, j \rangle = \langle \lambda - 1, 0 \rangle$ or $\langle 0, \lambda - 1 \rangle$, respectively;
  – $\overline{\varphi}_Y = \varphi_Y$ must hold for each $i \in [0, \lambda - 1]$ and $j = \lambda - 1$;
  – $\overline{\varphi}_X = \varphi_X$ must hold if $\langle i, j \rangle = \langle \lambda - 1, 0 \rangle$.

Error recovery works as in Section 4.2, either repeating a subquadrant computation (in case of a fingerprint mismatch on data at resiliency level δ_{d+1}) or making the recursive call at depth d fail (in case of a fingerprint mismatch on data at resiliency level δ_d).

Function Traceback-Path calls Boundary on at most $\lambda^2 - 1$ quadrants, and recursively calls itself on the (at most $2\lambda - 1$) subquadrants intersected by the traceback path π. Fingerprint computations during the forward and backward phases can be made consistent by reversing the access pattern and using amplified fingerprints, as in Section 4.2. Vector segments corresponding to quadrants not intersected by π still need to be scanned in order to correctly update their read fingerprints.

We will use two main ingredients to reduce the number of cache misses: (1) to improve temporal locality, instead of row-major order we access data in Z-order (see Section 6.2); and (2) 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 (see Section 6.3). While this is quite standard in the design of cache-oblivious algorithms, it has non-trivial consequences on fingerprint computation.

6.2 Z-order fingerprints

Computation of quadrant $\langle i,j\rangle$ requires input from the output boundaries of neighboring quadrants $\langle i,j-1\rangle$, $\langle i-1,j\rangle$, and $\langle i-1,j-1\rangle$. However, when using the Z-order, read operations on vectors H and V do not follow the write Z-order in which these vectors have been written: the output boundaries of the neighboring quadrants must be therefore retrieved from appropriate positions. Let $r_{\langle i,j\rangle}$ be the (write) rank of a quadrant $\langle i,j\rangle$ in Z-order, with $0 \le r_{\langle i,j\rangle} < \lambda^2$. We have the following lemma:

Lemma 6. *The write ranks of $\langle i,j-1\rangle$, $\langle i-1,j\rangle$, and $\langle i-1,j-1\rangle$ can be obtained from $r_{\langle i,j\rangle}$ as follows:*

$$\begin{cases} r_{\langle i,j-1\rangle} &= r_{\langle i,j\rangle} - (2^{2exp(j)+1}+1)/3 \\ r_{\langle i-1,j\rangle} &= r_{\langle i,j\rangle} - (2^{2exp(i)+2}+2)/3 \\ r_{\langle i-1,j-1\rangle} &= r_{\langle i,j\rangle} - (2^{2exp(j)+1}+2^{2exp(i)+2}+3)/3 \end{cases} \tag{2}$$

where $\exp(k)$ *is the exponent of 2 in the factorization of k.*

Proof. By the definition of Z-order we have that $r_{\langle i,j\rangle} = r_{i,j,n}$, where $r_{i,j,n}$ is recursively defined as:

$$r_{i,j,n} = \begin{cases} r_{(i \mod (n/2)),(j \mod (n/2)),n/2} + \lfloor \frac{i}{n/2} \rfloor \frac{n^2}{2} + \lfloor \frac{j}{n/2} \rfloor \frac{n^2}{4} & \text{if } n > 1 \\ 0 & \text{if } n = 1 \end{cases} \tag{3}$$

from which we get $r_{\langle i,j\rangle} = \sum_{k=1}^{\log n} \lfloor (i \mod 2^k)/2^{k-1} \rfloor 2^{2k-1} + \lfloor (j \mod 2^k)/2^{k-1} \rfloor 2^{2k-2}$. By definition of $exp(j)$, we have that j is divisible by $2^{exp(j)}$ and thus $\lfloor (j \mod 2^k)/2^{k-1} \rfloor = 1$ when $k = exp(j)+1$ and $\lfloor (j \mod 2^k)/2^{k-1} \rfloor = 0$ when $1 \le k \le exp(j)$. On the contrary, for $j-1$ we get $\lfloor (j-1 \mod 2^k)/2^{k-1} \rfloor = 0$ when $k = exp(j)+1$ and $\lfloor (j-1 \mod 2^k)/2^{k-1} \rfloor = 1$ when $1 \le k \le exp(j)$. It follows that $r_{\langle i,j\rangle} - r_{\langle i,j-1\rangle} = 2^{2exp(j)} - \sum_{k=1}^{exp(j)} 2^{2k-2} = (2^{2exp(j)+1} + 1)/3$, and the first case follows.

Similarly, $r_{\langle i,j\rangle} - r_{\langle i-1,j\rangle} = 2^{2exp(j)+1} - \sum_{k=1}^{exp(i)} 2^{2k-1} = (2^{2exp(i)+2}+2)/3$, from which we get the second case. Finally, the third case follows since $r_{\langle i,j\rangle} - r_{\langle i-1,j-1\rangle} = r_{\langle i,j\rangle} - r_{\langle i,j-1\rangle} + r_{\langle i,j-1\rangle} - r_{\langle i-1,j-1\rangle}$. □

During the Z-order forward computation, output boundaries of the computed quadrants are inserted into H and V, producing write fingerprints as described below. Let us consider the write and read operations on vector H at recursion depth d. We regard H as being divided into λ^2 segments, each of length n_{d+1}: the k-th segment contains the output down buffer of the k-th subproblem solved in accordance with the Z-order (similarly, V contains the right boundaries). We associate H with a down write fingerprint φ_H and a diagonal write fingerprint $\varphi_{H,1}$. The down fingerprint φ_H is computed from all the elements in the segments, which are accessed in Z-order, getting $\varphi_H = \sum_{x=0}^{\lambda^2 n_{d+1}-1} H[x]2^{wx} \mod p_d$. The diagonal fingerprint $\varphi_{H,1}$ is computed only from the last element in each segment, from which we get

$$\varphi_{H,1} = \sum_{x=0}^{\lambda^2-1} H[(n_{d+1}(x+1)-1)]2^{wx} \mod p_d$$

Since H stores down boundaries according to the Z-order, it is easy too see that the write fingerprints φ_H and $\varphi_{H,1}$ can be efficiently updated any time a new boundary is inserted into H. Segments of H corresponding to vector D will never be written: leaving these values to all zeroes allows us to skip corresponding updates to the fingerprints.

We now consider input extraction from H, needed for initializing a recursive call on quadrant $\langle i, j\rangle$. To produce vector T' (see Figure 4), data is extracted from H starting from position $n_{d+1}r_{\langle i-1,j\rangle}$. While reading from H the element corresponding to position x in T', with $0 \le x < n_{d+1}$, the read fingerprint $\overline{\varphi}_H$ is updated as follows:

$$\overline{\varphi}_H = \overline{\varphi}_H + H[n_{d+1}r_{\langle i-1,j\rangle} + x]2^{w(n_{d+1}r_{\langle i-1,j\rangle}+x)} \mod p_d$$

While reading from H the last element of the output boundary of quadrant $\langle i-1, j-1\rangle$, at position $r_{\langle i-1,j-1\rangle}n_{d+1} + n_{d+1} - 1$, the read fingerprint $\overline{\varphi}_{H,1}$ is updated as follows:

$$\overline{\varphi}_{H,1} = \overline{\varphi}_{H,1} + H[(r_{\langle i-1,j-1\rangle}+1)n_{d+1} - 1]2^{w\, r_{\langle i-1,j-1\rangle}} \mod p_d.$$

Vector V can be handled similarly, using a right write fingerprint $\varphi_V = \sum_{x=0}^{\lambda^2 n_{d+1}-1} V[x]2^{wx} \mod p_d$, and updating the corresponding read fingerprint $\overline{\varphi}_V$ as follows:

$$\overline{\varphi}_V = \overline{\varphi}_V + V[n_{d+1}r_{\langle i,j-1\rangle} + x]2^{w(n_{d+1}r_{\langle i,j-1\rangle}+x)} \mod p_d$$

while reading from V the element corresponding to position x in L', with $0 \le x < n_{d+1}$.

To make these computations efficient, we precompute the inverse 2^{-w} of 2^w in ring $\mathcal{Z}_{p_d}$, for all selected primes p_d. We also maintain throughout the quadrant computation the running value $2^{w\, n_{d+1}\, r_{\langle i,j\rangle}}$ and exploit Equations (2). It is easy to see that the above write and read fingerprints match if no faults occur.

The computation of a quadrant $\langle i, j\rangle$ requires the segment of X and Y containing entries $x_{\lfloor i/n_{d+1}\rfloor+1}, \ldots, x_{\lceil i/n_{d+1}\rceil+n_{d+1}}$ and $y_{\lfloor j/n_{d+1}\rfloor+1}, \ldots, y_{\lfloor j/n_{d+1}\rfloor+n_{d+1}}$ (recall that indexes in X and Y start from 1). Being quadrants accessed in Z-order, we get that the read fingerprint of X is:

$$\overline{\varphi}_X = \sum_{i=0}^{\lambda-1}\sum_{j=0}^{\lambda-1}\sum_{x=0}^{n_{d+1}-1} X[n_{d+1}j + x + 1]2^{w(n_{d+1}r_{\langle i,j\rangle}+x)} \mod p_d.$$

Every time a value of X in position $n_{n_{d+1}j+x+1}$ is read for the i-th time, the read fingerprint can be easily updated as follows:

$$\overline{\varphi}_X = \overline{\varphi}_X + X[n_{d+1}j + x + 1]2^{w(n_{d+1}r_{\langle i,j\rangle}+x)} \mod p_d.$$

A similar approach applies for $\overline{\varphi}_Y$. In order to compute the write fingerprint φ_X of X, we observe that the terms in the read fingerprint can be reordered in such a way that each term of X appears just once multiplied by a suitable value. Hence, we get the following amplified write fingerprint:

$$\varphi_X = \sum_{j=0}^{\lambda-1} \sum_{x=0}^{n_{d+1}-1} X[n_{d+1}j + x + 1] \sum_{i=0}^{\lambda-1} 2^{w(n_{d+1}r_{\langle i,j \rangle}+x)} \mod p_d.$$

By using the statement of Lemma 6, the write fingerprint can be efficiently computed by scanning once vector Y. A similar write fingerprint applies for Y.

6.3 Virtual recursion with amplified fingerprints

To improve spatial locality, we reduce the length of the auxiliary vectors, at recursion depth d, from λn_d to $\Theta(n_d)$. We extend a technique proposed in [18, 19]: at any time, only appropriate subvectors are stored, obtaining the missing parts, when necessary, by repeating forward computations. The main issue that needs to be settled in our setting is that repeated computations imply multiple extractions of the same data, and the correctness of each data read must be appropriately checked: to this aim, we devise opportunely crafted amplified fingerprints, coupled with the data access pattern induced by multiple extractions.

Virtual quadrants. We superimpose over the $\lambda \times \lambda$ subdivision of an $n_d \times n_d$ quadrant a $\log_2 \lambda$-depth hierarchical decomposition into *virtual quadrants* (for simplicity we assume that λ is a power of 2). The i-th level of the decomposition, for $1 \leq i \leq \log_2 \lambda$, consists of 4^i virtual quadrants of size $n_d/2^i \times n_d/2^i$. At the first level, virtual quadrants correspond to $Q_{0,0}$, $Q_{0,1}$, $Q_{1,0}$, and $Q_{1,1}$, and each of them is recursively split into four parts.

The auxiliary vectors H and V used with virtual recursion are logically split into $\log_2 \lambda$ segments of decreasing length, all stored at resiliency level δ_d. The i-th segment H_i of vector H, for $1 \leq i \leq \log_2 \lambda$, has length $n_d/2^{i-1}$ and contains the output boundaries of two upper virtual quadrants at the i-th level of the hierarchical decomposition. V has a similar structure and both vectors have length $\Theta(n_d)$. Since the i-th virtual level contains 4^i quadrants, the space allocated for segments H_i and V_i will be reused throughout the computation as described below (see also Figure 5).

Virtual recursion. We mimic the behavior of algorithm RecLCS described in Section 4 by exploiting segments H_i and V_i to store the output boundaries of virtual quadrants at level i in the hierarchical decomposition. Function Boundary still works in Z-order on the $\lambda \times \lambda$ subdivision, except for recycling space in vectors H and V by overwriting boundaries that are no longer needed to complete the forward computation. Function Traceback-Path works recursively on virtual quadrants. However, recursion on the $n_d/2^i \times n_d/2^i$ virtual quadrants cannot be explicit, since the recursion depth would exceed the amount of private memory when $P = o(\log n)$. Hence, 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 $O(1)$ indexes and variables, since intermediate data of non-constant size is stored in H_i and V_i. Real recursive calls are performed when $n_d/2^i$ becomes equal to n_{d+1} (i.e., when $i = \log_2 \lambda$). The resiliency is kept at level $\delta_d = \lceil \delta/\lambda^d \rceil$ during virtual recursion, and drops to $\delta_{d+1} = \lceil \delta/\lambda^{d+1} \rceil$ on real recursive calls.

Since vectors H and V no longer maintain boundaries of all $n_{d+1} \times n_{d+1}$ quadrants, each virtual recursive call of Traceback-Path requires recomputing the $n_d/2^i \times n_d/2^i$ virtual

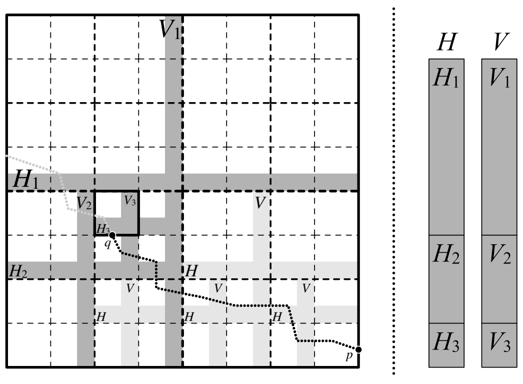

Figure 5: Traceback path computation with virtual recursion on an $(n_d \times n_d)$-size quadrant.

quadrant boundaries by invoking function Boundary. Hence, with respect to the algorithm of Section 6.1, at recursion level d the forward computation of a quadrant can be performed up to $\log_2 \lambda$ times due to virtual recursive calls of Traceback-Path. As shown in [18, 19], in the absence of faults the additional forward computations do not asymptotically increase the running time. However, reading $\omega(1)$ times the same memory location requires more powerful fingerprints.

In the example of Figure 5, the entry point p of the traceback path π is contained in the $n_d/2 \times n_d/2$ virtual quadrant $Q_{1,1}$. Boundary is invoked on $Q_{0,0}$, $Q_{0,1}$, and $Q_{1,0}$, and the (first-level) simulated recursive calls of Traceback-Path are performed on $Q_{1,1}$, $Q_{1,0}$, and $Q_{0,0}$. In this example, $n_d/8 = n_{d+1}$ and q is the entry point in a real recursive call. The forward computation of the $n_{d+1} \times n_{d+1}$ quadrant containing q is repeated three times, once per virtual recursion level. Dark grey indicates subvectors currently stored in H and V. Light grey indicates subvectors whose values have been already overwritten to save space. The data layout of vectors H and V is shown on the right hand side of the picture.

Amplified fingerprints. We now show how to devise amplified fingerprints suitable for the data access pattern induced by virtual recursion of function Traceback-Path: in our approach, if a memory location is read k times, the value written to that location contributes to a write fingerprint with k different exponents (values $f_{i,j}$ in Section 2.2).

Let us focus on vector H. At the i-th virtual recursion level, with $1 \leq i \leq \log_2 \lambda$, read accesses to H follow an inverted Z-order on 4^i data segments of length $n_d/2^i$. To describe the write fingerprint φ_H, we regard H as projected on a longer vector $\overline{H}$, divided into $\log_2 \lambda$ layers: layer i contains the 4^i virtual boundaries of length $n_d/2^i$. In each layer, elements in each virtual boundary are consecutive, while boundaries are arranged in Z-order. Layer i has thus length $2^i n_d$ and starts from position $\sum_{t=1}^{i-1} 2^t n_d = n_d(2^i - 2)$. Algorithmically, let l be the current depth of virtual recursion and let D' be the $(n_d/2^l)$-length output received from a call to function Boundary. Consider a generic element $h \in D'$ that is written into vector H: h would belong to layers $[l, \log_2 \lambda]$ of vector $\overline{H}$. We denote by r_j the rank in Z-order of the $(n_d/2^j)$-length boundary of layer j to which h belongs, and by s_j the number of elements preceding h in this boundary. Then the write fingerprint φ_H is updated as follows:

$$\varphi_H = \varphi_H + h \sum_{j=l}^{\log_2 \lambda} 2^{w(n_d(2^j-2)+r_j \frac{n_d}{2^j}+s_j)} \mod p_d$$

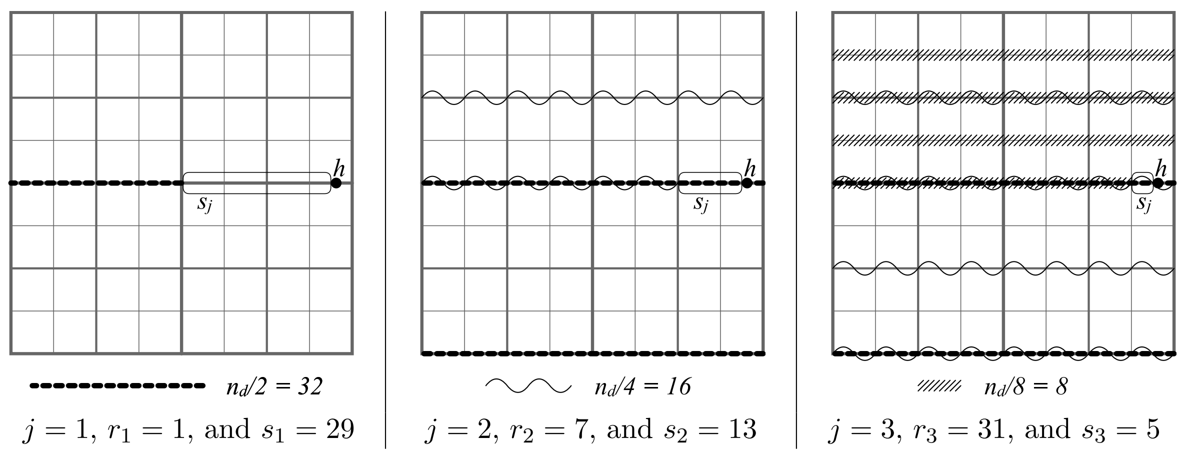
h
s_j
$n_d/2 = 32$
$j = 1$, $r_1 = 1$, and $s_1 = 29$
h
s_j
$n_d/4 = 16$
$j = 2$, $r_2 = 7$, and $s_2 = 13$
h
s_j
$n_d/8 = 8$
$j = 3$, $r_3 = 31$, and $s_3 = 5$

Figure 6: Amplified fingerprint computation. In this example $n_d = 64$, $\lambda = 8$ (from which $n_{d+1} = 8$), and h is in row 31 and column 61 of the $n_d \times n_d$ quadrant.

Figure 6 shows an example where h is the 6-th element of an n_{d+1}-length boundary and appears in all layers of $\overline{H}$, detailing values r_j and s_j for all $j \in [1, \log_2 \lambda]$.

Read fingerprints during calls to Traceback-Path at level l are updated similarly:

$$\overline{\varphi}_H = \overline{\varphi}_H + h2^{w(n_d(2^l-2)+r_l\frac{n_d}{2^l}+s_l)} \mod p_d$$

Virtual quadrants of level l not intersected by π imply updating the read fingerprint for all levels $[l, \log_2 \lambda]$ at once by exploiting the same formula used for the write fingerprint. Both vector V and the input sequences can be handled similarly.

6.4 Analysis

Theorem 6. *Algorithm* ShallowLCS *computes the longest common subsequence correctly with high probability. It requires $O(mn+\delta mn^{c/P}P)$ time and incurs $O(mn/(BM)+\delta mn^{c/P}P/B)$ cache misses in the worst case, where m and n (with $m \geq n$) are the lengths of the input sequences, P is the available private memory, c is a small constant, M and B are the cache and the cache line sizes, and the upper bound δ on the number of faults is polynomial in m.*

Proof. Since the recursion depth $\rho = \Theta(\log_\lambda n) = O(\log n)$, the same arguments used in Lemma 5 prove that all faults are detected with high probability. The analysis of running time and cache misses exploits arguments similar to the proof of Theorem 3. The main differences are due to the $\lambda \times \lambda$ subdivision and to virtual recursion (fingerprint computations can be amortized as described in Section 6.2).

Recall that $\lambda = \lceil n^{1/\rho} \rceil$. The running time $T_B(n, \delta)$ of successful computations of function Boundary is given by the following recurrence:

$$T_B(\hat{n}, \hat{\delta}) = \begin{cases} O(\hat{n}^2(\hat{\delta}+1)) & \text{if } \hat{n} \leq \lambda \\ \lambda^2 T_B(\hat{n}/\lambda, \hat{\delta}/\lambda) + O(\hat{n}(\hat{\delta}+1)\lambda) & \text{if } \hat{n} > \lambda \end{cases}$$

Indeed, in the base case $\hat{n} \times \hat{n}$ table entries are computed at resiliency level $\hat{\delta}$. Otherwise, λ^2 recursive calls are performed and each call requires $O(\hat{n}(\hat{\delta}+1)/\lambda)$ time for preparing input vectors and fingerprints. Standard arguments show that $T_B(n, \delta) = O(n^2 + \delta n \lambda \log_\lambda n)$. We remark that virtual recursion only affects the order in which actual recursive calls on the λ^2 subquadrants are performed (changing it from row-major to Z-order). The running time of unsuccessful computations is not affected by this order and does not exceed $T_B(n, \delta)$: this can be shown by mimicking the proof of Theorem 3, replacing constant 2 with λ.

Similarly to previous analyses, the cost of function TRACEBACK-PATH is dominated by successful computations, whose running time $T_P(n, \delta)$ is given by the following recurrence:

$$T_P(\hat{n}, \hat{\delta}) = \begin{cases} O(\hat{n}^2(\hat{\delta}+1)) & \text{if } \hat{n} \le \lambda \\ (2\lambda - 1)T_P(\hat{n}/\lambda, \hat{\delta}/\lambda) + O(\hat{n}^2 + \hat{n}(\hat{\delta}+1)\lambda^{\log_2 3}\log_\lambda \hat{n}) & \text{if } \hat{n} > \lambda \end{cases}$$

The additive term for the case $\hat{n} > \lambda$ is due to calls to function BOUNDARY: indeed, besides the $2\lambda - 1$ recursive calls, TRACEBACK-PATH invokes at most 3^j times BOUNDARY with input size $\hat{n}/2^j$ and resiliency $\hat{\delta} + 1$, for each $1 \le j \le \log_2 \lambda$. It follows that $T_P(n, \delta) = O(n^2 + \delta n \lambda^{\log_2 3} \log_\lambda n)$.

By definition of ρ, we have that $v/P \ge 1/2\rho$, where v is the number of local variables used by the algorithm (see Section 6.1). Hence, function TRACEBACK-PATH requires $O(n^2 + \delta n^{1+c/P} P)$ time, for any constant $c \ge 2v \log_2 3$.

We now analyze cache complexity. We neglect cache misses due to fingerprint computations since they are irrelevant as long as $P = \Theta(1)$ or the private cache size is $\Omega(\log M)$. We first bound the number of cache misses due to successful computations (similarly to the running time, it can be proved that unsuccessful computations are dominated by this bound). The cache complexity $Q_B(n, \delta)$ of function BOUNDARY is described by the following recurrence:

$$Q_B(\hat{n}, \hat{\delta}) = \begin{cases} O(\hat{n}^2\hat{\delta}/B) & \text{if } \hat{n} \le \lambda \text{ and } \hat{\delta} > M \\ O(\hat{n}^2(\hat{\delta}+1)^2/(BM) + \hat{n}(\hat{\delta}+1)/B + 1) & \begin{array}{l}\text{if } \hat{n} \le \lambda \text{ and } \hat{\delta} \le M, \text{ or} \\ \quad \hat{n} > \lambda \text{ and } \hat{n}(\hat{\delta}+1) \le \lambda M\end{array} \\ \lambda^2 Q_B(\hat{n}/\lambda, \hat{\delta}/\lambda) + O(\hat{n}(\hat{\delta}+1)\lambda/B) & \text{if } \hat{n} > \lambda \text{ and } \hat{n}(\hat{\delta}+1) > \lambda M \end{cases} \tag{4}$$

Besides the problem size, the different cases also depend on the relationship between the resiliency level $\hat{\delta}$ and the cache size M. We reach a base case either when we are at the last recursion level (i.e., $\hat{n} \le \lambda$) or when a subproblem fits in cache (i.e., $\hat{n}(\hat{\delta}+1)/\lambda \le M$). In the first case, the cache is too small to exploit temporal locality and $\Theta(\hat{\delta}/B)$ cache misses are required for each table access. The second and third base cases take instead advantage of the temporal locality: thanks to the Z-order, subproblems are partitioned into $O(\hat{n}\hat{\delta}/M)^2$ groups, and processing each group costs $O(M/B)$ cache misses. In the last case BOUNDARY performs λ^2 recursive calls, each requiring $n_d(\delta_d + 1)/(\lambda B)$ misses for preparing input vectors and fingerprints. It can be proved that $Q_B(n, \delta) = O(n^2/(BM) + (n\delta\lambda/B)\log_\lambda(n\delta/M))$.

The recurrence for the cache complexity $Q_P(n, \delta)$ of function TRACEBACK-PATH can be derived similarly:

$$Q_P(\hat{n}, \hat{\delta}) = \begin{cases} O(\hat{n}^2\hat{\delta}/B) & \text{if } \hat{n} \le \lambda \text{ and } \hat{\delta} > M \\ O(\hat{n}^2(\hat{\delta}+1)^2/(BM) + \hat{n}(\hat{\delta}+1)\lambda^{\log_2(3/2)}/B + 1) & \begin{array}{l}\text{if } \hat{n} \le \lambda \text{ and } \hat{\delta} \le M, \text{ or} \\ \quad \hat{n} > \lambda \text{ and } \hat{n}(\hat{\delta}+1) \le \lambda M\end{array} \\ \begin{array}{l}(2\lambda-1)Q_P(\hat{n}/\lambda, \hat{\delta}/\lambda) + \\ \quad + O(\hat{n}^2/(BM) + (\hat{n}\hat{\delta}\lambda^{\log_2 3}\log_\lambda \hat{n})/B)\end{array} & \text{if } \hat{n} > \lambda \text{ and } \hat{n}(\hat{\delta}+1) > \lambda M \end{cases}$$

In the first case, forward computations performed by function BOUNDARY dominate the number of cache misses: at the i-th virtual recursion level, at most 3^i subquadrants of size $\hat{n}^2/4^i$

are recomputed. The second and third cases are obtained by summing up, for $1 \leq j \leq \log_2 \lambda$, the cache misses of 3^j calls to function BOUNDARY (which are given by the corresponding cases in Equation 4). In the fourth case, besides 3^j calls to BOUNDARY with input size $n_d/2^j$ and resiliency $(\delta + 1)$, for each virtual recursion level j, TRACEBACK-PATH also performs $2\lambda - 1$ recursive calls with input size $\hat{n}/\lambda$ and resiliency $\hat{\delta}/\lambda$. It can be proved that the recurrence solves to $Q_P(n, \delta) = O(n^2/(BM) + (n\delta\lambda^{\log_2 3} \log_\lambda n)/B) = O(n^2/(BM) + \delta n^{1+c/P} P/B)$, where constant c was defined in the running time analysis.

The theorem follows by multiplying the running time and cache complexity of function TRACEBACK-PATH by $\lceil m/n \rceil$, in order to deal with sequences of different lengths. □

Similar techniques and analyses can be adapted to I-GEP problems, extending the bounds given in Section 5 accordingly.

7 Concluding remarks

In this paper we have devised the first resilient algorithms for dynamic programming problems. This has been regarded as an elusive goal for many years in a variety of faulty-memory models, especially for problems with non-local dependencies such as all-pairs shortest paths. Under plausible assumptions, we can correctly solve, with high probability, both local dependency DP problems and more challenging problems that fit in the Gaussian Elimination Paradigm. The asymptotic time and space bounds of our resilient algorithms match those of the standard non-resilient counterparts up to a polynomial number of faults. Our recursive algorithms can also tolerate destructive faults at any level of the memory hierarchy, while still incurring a small number of cache misses. These results can be cast into a general framework based on a careful combination of fingerprinting with data replication and majority computations, which might be of independent interest in the design of resilient algorithms for different problems.

Computing in the presence of memory faults still poses many challenging questions. This paper, as well as previous works in the faulty RAM model, crucially rely on the knowledge of the maximum number δ of memory faults: we regard the design of δ-oblivious algorithms as an interesting research direction. Moreover, many real-world applications deal with huge graphs that can easily consist of billions of nodes and are thus particularly vulnerable to soft memory errors. Unfortunately, most fundamental graphs problems, including graph traversals, are completely unsolved in faulty memories.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] M. Ajtai, V. Feldman, A. Hassidim, and J. Nelson. Sorting and selection with imprecise comparisons. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming*, pages 37–48, 2009.

[3] J. A. Aslam and A. Dhagat. Searching in the presence of linearly bounded errors. In *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pages 486–493, 1991.

[4] S. Assaf and E. Upfal. Fault tolerant sorting networks. *SIAM Journal of Discrete Mathematics*, 4(4):472–480, 1991.

[5] Y. Aumann and M. A. Bender. Fault tolerant data structures. In *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science*, pages 580–589, 1996.

[6] M. A. Babenko and I. Pouzyrevsky. Resilient quicksort and selection. In *Proceedings of the 7th International Computer Science Symposium in Russia*, volume 7353 of *LNCS*, pages 6–17, 2012.

[7] J. Blömer and J. Seifert. Fault based cryptanalysis of the advanced encryption standard (AES). In *Financial Cryptography*, pages 162–181, 2003.

[8] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2–3):225–244, 1994.

[9] R. S. Borgstrom and S. R. Kosaraju. Comparison-based search in the presence of errors. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 130–136, 1993.

[10] 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.

[11] 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 *Proceedings of the 15th European Symposium on Algorithms*, volume 4698 of *LNCS*, pages 347–358, 2007.

[12] G. S. Brodal, A. G. Jørgensen, and T. Mølhave. Fault tolerant external memory algorithms. In *Proceedings of the 11th Workshop on Algorithms and Data Structures*, volume 5664 of *LNCS*, pages 411–422, 2009.

[13] G. S. Brodal, A. G. Jørgensen, G. Moruz, and T. Mølhave. Counting in the presence of memory faults. In *Proceedings of the 20th International Symposium on Algorithms and Computation*, volume 5878 of *LNCS*, pages 842–851, 2009.

[14] S. Caminiti, I. Finocchi, and E. G. Fusco. Local dependency dynamic programming in the presence of memory faults. In *Proceedings of the 28th Symposium on Theoretical Aspects of Computer Science*, volume 9 of *LIPIcs*, pages 45–56, 2011.

[15] S. Caminiti, I. Finocchi, E. G. Fusco, and F. Silvestri. Dynamic programming in faulty memory hierarchies (cache-obliviously). In *Proceedings of the 31st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 13 of *LIPIcs*, pages 433–444, 2011.

[16] V. Chen, E. Grigorescu, and R. de Wolf. Error-correcting data structures. *SIAM Journal on Computing*, 42(1):84–111, 2013.

[17] R. A. Chowdhury, H. S. Le, and V. Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 7(3):495–510, 2010.

[18] R. A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms*, pages 591–600, 2006.

[19] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 207–216, 2008.

[20] R. A. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems*, 47:878–919, 2010.

[21] R. A. Chowdhury, V. Ramachandran, F. Silvestri, and B. Blakeley. Oblivious algorithms for multicores and networks of processors. *Journal of Parallel and Distributed Computing*, 73(7):911–925, 2013.

[22] P. Christiano, E. D. Demaine, and S. Kishore. Lossless fault-tolerant data structures with additive overhead. In *Proceedings of the 14th International Symposium on Algorithms and Data Structures*, volume 6844 of *LNCS*, pages 243–254, 2011.

[23] M. Chu, S. Kannan, and A. McGregor. Checking and spot-checking the correctness of priority queues. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming*, volume 4596 of *LNCS*, pages 728–739, 2007.

[24] U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM Journal on Computing*, 23(5):1001–1018, 1994.

[25] U. Ferraro Petrillo, I. Finocchi, and G. F. Italiano. The price of resiliency: a case study on sorting with memory faults. *Algorithmica*, 53(4):597–620, 2009.

[26] U. Ferraro Petrillo, F. Grandoni, and G. F. Italiano. Data structures resilient to memory faults: An experimental study of dictionaries. *ACM Journal of Experimental Algorithmics*, 18:1.6:1.1–1.6:1.14, 2013.

[27] I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal resilient sorting and searching in the presence of memory faults. *Theoretical Computer Science*, 410(44):4457–4470, 2009.

[28] I. Finocchi, F. Grandoni, and G. F. Italiano. Resilient dictionaries. *ACM Transactions on Algorithms*, 6(1):1:1–1:19, 2009.

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

[30] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4:1–4:22, 2012.

[31] F. Gieseke, G. Moruz, and J. Vahrenhold. Resilient k-d trees: k-means in space revisited. *Frontiers of Computer Science*, 6(2):166–178, 2012.

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

[33] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.

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

[35] A. G. Jørgensen, G. Moruz, and T. Mølhave. Priority queues resilient to memory faults. In *Proceedings of the 10th Workshop on Algorithms and Data Structures*, volume 4619 of *LNCS*, pages 127–138, 2007.

[36] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[37] T. Kopelowitz and N. Talmon. Selection in the presence of memory faults, with applications to in-place resilient sorting. In *Proceedings of the 23rd International Symposium on Algorithms and Computation*, volume 7676 of *LNCS*, pages 558–567, 2012.

[38] F. 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(2):265–304, 1997.

[39] D. Li, Z. Chen, P. Wu, and J. S. Vetter. Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 44:1–44:12, 2013.

[40] S. Muthukrishnan. On optimal strategies for searching in presence of errors. In *Proceedings of the 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 680–689, 1994.

[41] A. Pelc. Searching games with errors - fifty years of coping with liars. *Theoretical Computer Science*, 270(1–2):71–109, 2002.

[42] L.L. Pilla, P. Rech, F. Silvestri, C. Frost, P.O.A Navaux, M. Sonza Reorda, and L. Carro. Software-based hardening strategies for neutron sensitive FFT algorithms on GPUs. *IEEE Transactions on Nuclear Science*, 61(4):1874–1880, 2014.

[43] M. O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[44] P. Rech, L. Pilla, F. Silvestri, P. Navaux, and L. Carro. Neutron sensitivity and software hardening strategies for matrix multiplication and FFT on graphics processing units. In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale*, pages 13–20, 2013.

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

[46] L. De Stefani and F. Silvestri. Exploiting non-constant safe memory in resilient algorithms and data structures. Arxiv 1305.3828, 2013.

[47] J. S. Vitter. *Algorithms and Data Structures for External Memory.* Now Publishers Inc., 2008.

[48] M. Ward. Mistakes in silicon chips to help boost computer power. *BBC News*, May 25, 2010. `http://news.bbc.co.uk/2/hi/technology/10134655.stm`.

Appendix

Primes can be generated in the faulty-RAM model using a variant of a well-known algorithm, which we call `Random-Prime`. The algorithm generates a random prime p in a given interval $I = [n^{c-1}, n^c]$, where c is a constant and p can be represented in $O(1)$ memory words of size w, as follows:

1. select uniformly at random an odd number $p \in I$;
2. test p for primality by applying t times the Miller-Rabin test [43];
3. if p is composite then repeat step 1, else return p.

Differently from the usual implementation, which has expected running time, after at most r iterations of steps 1–3 we return p, even if the primality test fails. The choice of parameters t and r is discussed below.

The Miller-Rabin test has one-sided error: when it determines a number composite then the result is always true, but when it asserts that a number is prime there is a provably small probability of error. Although the probability of failure in our case does not uniquely depend on the Miller-Rabin test, it is not difficult to prove that this probability remains small and that the algorithm can be executed in our model:

Claim 1. *For any integers r, $t > 0$ and any constant $c \geq 1$, algorithm* `Random-Prime` *selects uniformly at random a prime number p in $I = [n^{c-1}, n^c]$ with error probability upper bounded by $(1 - 1/(\vartheta \log n))^r + r4^{-t}$, for some positive constant ϑ. The algorithm requires $O(rt \log n)$ time and $O(1)$ private memory words.*

Proof. It is well known that t iterations of the Miller-Rabin primality test require $O(t \log n)$ mod-n multiplications (using modular exponentiation). Thus, in the faulty-memory RAM model we can check in $O(t \log n)$ time whether a number p in $\Theta(n^c)$ is prime, for any constant c. The test can be implemented using a constant number of memory words, which can be all stored in the private memory. The probability of failure of the Miller-Rabin test, i.e., the probability for a composite number to be mistakenly recognized as a prime, is bounded by 4^{-t} [43]. We now analyze the error probability of algorithm `Random-Prime`.

By the prime number theorem, there are $\Theta(n^c/\log n)$ prime numbers in interval I. Since $|I| = \Theta(n^c)$, the probability of picking a prime when selecting an odd number uniformly at random in I is $1/(\vartheta \log n)$, for some positive constant ϑ. Let us now define the following events, for each i such that $1 \leq i \leq r$:

- L_i is the event "the i-th generated number is composite and the primality test mistakenly identifies it as a prime";
- D_i is the event "the i-th generated number is composite and the primality test correctly identifies it as composite".

The probability of each event L_i is at most $(1-1/(\vartheta \log n))4^{-t}$. The probability of each event D_i is upper bounded by $(1-1/(\vartheta \log n))$. The overall probability that a run of subroutine `Random-Prime` returns a composite number is given by $Pr\{L_1\} + Pr\{D_1 \cap L_2\} + \ldots + Pr\{D_1 \cap$

$\ldots \cap D_{r-1} \cap L_r\} + Pr\{D_1 \cap \ldots \cap D_r\}$, which is upper bounded by

$$\left(1 - \frac{1}{\vartheta \log n}\right)^r + \sum_{i=0}^{r-1} \left(1 - \frac{1}{\vartheta \log n}\right)^i \left(1 - \frac{1}{\vartheta \log n}\right) 4^{-t} <$$

$$\left(1 - \frac{1}{\vartheta \log n}\right)^r + 4^{-t} \sum_{i=0}^{r-1} 1^{i+1} = \left(1 - \frac{1}{\vartheta \log n}\right)^r + r4^{-t}$$

Since any prime in I has the same probability of being selected, the claim follows. $\square$

By choosing $r = \Theta(\log^2 n)$ and $t = \Theta(\log n)$, the error probability given in Claim 1 can be made smaller than $1/n^\gamma$ for any constant γ. This yields Lemma 1.