Data Movement in Flash Memories

Anxiao (Andrew) Jiang
Computer Science Department
Texas A&M University
College Station, TX 77843
ajiang@cse.tamu.edu

Michael Langberg
Computer Science Division
Open University of Israel
Raanana 43107, Israel
mikel@openu.ac.il

Robert Mateescu
Electrical Engineering Dept.
Caltech
Pasadena, CA 91125
mateescu@paradise.caltech.edu

Jehoshua Bruck
EE & CNS Dept.
Caltech
Pasadena, CA 91125
bruck@caltech.edu

Abstract

NAND flash memories are the most widely used non-volatile memories, and data movement is common in flash storage systems. We study data movement solutions that minimize the number of block erasures, which are very important for the efficiency and longevity of flash memories. To move data among n blocks with the help of Δ auxiliary blocks, where every block contains m pages, we present algorithms that use $\Theta(n \cdot \min\{m, \log_\Delta n\})$ erasures without the tool of coding. We prove this is almost the best possible for non-coding solutions by presenting a nearly matching lower bound. Optimal data movement can be achieved using coding, where only $\Theta(n)$ erasures are needed. We present a coding-based algorithm, which has very low coding complexity, for optimal data movement. We further show the NP hardness of both coding-based and non-coding schemes when the objective is to optimize data movement on a *per instance* basis.

I. Introduction

NAND flash memories are the most widely used non-volatile memories due to their high data density and efficiency. In a NAND flash memory, cells are organized as blocks. A block has about 10^5 cells, and a cell can store one or more bits. Every block is partitioned into pages, where a page is the unit of a read or write operation. A prominent property of flash memories is *block erasure*. It means to change any stored data, the whole block must be erased first before rewriting. Block erasures significantly decrease the longevity and the speed of flash memories, so it is very important to reduce them [5].

Flash memories often store a large amount of data, and data movement is very useful for reassembling files, wear leveling, and in-place computation. We consider the basic form where the data in different pages need to be switched. This problem was first studied in [10], where coding-based data movement is shown to minimize the number of erasures. In this paper, we significantly extend the known results by rigorously proving the gain of coding, presenting efficient data movement algorithms, and showing the NP hardness of per-instance optimization.

In the data movement problem, there are n blocks, where each block has m pages of data. The mn pages of data need to be moved into each other's positions as required. Δ empty auxiliary blocks can be used to help data movement. The objective is to minimize the number of block erasures in the data movement process. We present efficient algorithms that use $\Theta(n \cdot \min\{m, \log_\Delta n\})$ erasures without the tool of coding. We prove it is nearly the best possible by proving a close lower bound. Since coding-based solutions require $\Theta(n)$ erasures, this result rigorously proves the benefit of coding.

We present a strictly optimal coding-based algorithm for $\Delta = 1$ with at most $2n - 1$ erasures. It has very low coding complexity. We further show that if the objective is to optimize data movement on a per instance basis, the problem is NP hard for both coding and non-coding schemes. Nevertheless, the coding technique in the above algorithm can be readily utilized in any per-instance-optimal solution.

A number of recent works have studied coding for rewriting [4], [6], [7], [9], [12] and error correction [8], [11] in flash memories at the cell level. There are also many works studying algorithms and data structures for flash data-storage systems [5]. This paper focuses on coding for data movement at the page level, and the results can be used to design more efficient flash storage systems.

The rest of the paper is organized as follows. Section II defines the data movement problem. Section III presents data movement algorithms without coding. Section IV derives a lower bound for data movement without coding. Section V presents an efficient coding scheme for optimal data movement. Section VI studies the complexity and approximation of per-instance optimization. Section VII presents the conclusions.

II. Notations and Concepts

There are n blocks containing data, denoted by $B_1, \ldots, B_n$. Every block consists of m pages. For $i = 1, \ldots, n$, the pages in B_i are called $p_{i,1}, \ldots, p_{i,m}$. For $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant m$, the data originally stored in $p_{i,j}$ is denoted by $d_{i,j}$. There is a function α:

$$\alpha \ : \ \{1, \ldots, n\} \times \{1, \ldots, m\} \to \{1, \ldots, n\}$$

such that $\forall\ 1 \leqslant i \leqslant n$, $|\{(a,b) | 1 \leqslant a \leqslant n, 1 \leqslant b \leqslant m, \alpha(a,b) = i\}| = m$. Our objective is to move (i.e., write) the data $d_{i,j}$ into a page in block $B_{\alpha(i,j)}$, for $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant m$. There are Δ empty extra blocks, called *auxiliary blocks*, that we can use to help move data. (Every auxiliary block also has m pages.) For reliability, it is required that during the data-movement process, the data in these $n + \Delta$ blocks must always be sufficient for recovering all the original data. In the end, the auxiliary blocks should return to the empty state. We measure the cost of data movement by the total number of block erasures. (Note

that changing any data in a block requires erasing the block first.) The solution that minimizes the number of erasures is called *optimal*.

There exist two types of solutions, namely, solutions with and without coding. In a solution without coding, data are simply copied from page to page. In a coding-based solution, the data written into a page can be any function of the existing data.

Example 1. *We show an example without coding where $m = 3$, $n = 4$ and $\Delta = 2$. Each page is indexed by its label (the destination block), and its content (the card suit). For example the pages indexed by $1_\heartsuit$, $1_\spadesuit$ and $1_\diamondsuit$ should be moved to the first block. In the first step, the content of the first block is copied to the auxiliary memory, and the first block is then erased. In the second step, the pages labeled with 1 are copied into the first block. In the third step, before erasing block 2, we copy only $3_\diamondsuit$ and $4_\diamondsuit$ to the auxiliary memory, because $1_\heartsuit$ already appears in block 1. After several steps, we realize the desired data movement.*

$2_\heartsuit$	$1_\heartsuit$	$3_\spadesuit$	$2_\diamondsuit$		
$1_\spadesuit$	$3_\diamondsuit$	$4_\heartsuit$	$1_\diamondsuit$		
$4_\spadesuit$	$4_\diamondsuit$	$2_\spadesuit$	$3_\heartsuit$		

$\rightarrow$

	$1_\heartsuit$	$3_\spadesuit$	$2_\diamondsuit$	$2_\heartsuit$	$4_\spadesuit$
	$3_\diamondsuit$	$4_\heartsuit$	$1_\diamondsuit$		$1_\spadesuit$
	$4_\diamondsuit$	$2_\spadesuit$	$3_\heartsuit$		

$\rightarrow$

$1_\heartsuit$	$1_\heartsuit$	$3_\spadesuit$	$2_\diamondsuit$	$2_\heartsuit$	$4_\spadesuit$
$1_\spadesuit$	$3_\diamondsuit$	$4_\heartsuit$	$1_\diamondsuit$		$1_\spadesuit$
$1_\diamondsuit$	$4_\diamondsuit$	$2_\spadesuit$	$3_\heartsuit$		

$\rightarrow$

$1_\heartsuit$		$3_\spadesuit$	$2_\diamondsuit$	$2_\heartsuit$	$4_\spadesuit$
$1_\spadesuit$		$4_\heartsuit$	$1_\diamondsuit$	$3_\diamondsuit$	$1_\spadesuit$
$1_\diamondsuit$		$2_\spadesuit$	$3_\heartsuit$		$4_\diamondsuit$

$\rightarrow \cdots \rightarrow$

$1_\heartsuit$	$2_\heartsuit$	$3_\heartsuit$	$4_\heartsuit$		
$1_\spadesuit$	$2_\spadesuit$	$3_\spadesuit$	$4_\spadesuit$		
$1_\diamondsuit$	$2_\diamondsuit$	$3_\diamondsuit$	$4_\diamondsuit$		

It is known that coding-based solutions can optimize data movement [10], which use at most $2n-1$ erasures. We now rigorously prove the gain of coding, and show efficient data-movement algorithms.

III. Data Movement without Coding

In this section, we present two algorithms for data movement without coding. They use $n\lceil\log_\Delta n\rceil + \frac{3n}{2}$ and $\frac{5nm}{2}$ erasures, respectively. This shows that without coding, $\Theta(n \cdot \min\{m, \log_\Delta n\})$ erasures are sufficient for data movement. Here we assume $\Delta \geqslant 2$, because when $\Delta = 1$ there are cases that cannot be solved without using coding [10].

A. Block-merging Algorithm

Given a positive integer i, let $[i]$ denote $\{1, 2, \ldots, i\}$. Call the Δ auxiliary blocks $B_{n+1}, \ldots, B_{n+\Delta}$. For $n+1 \leqslant i \leqslant n+\Delta$, denote the m pages in B_i by $p_{i,1}, \ldots, p_{i,m}$. In the data movement process, data are copied from page to page. At any given moment, if a page p_{i_1,j_1} stores the data d_{i_2,j_2}, the we use ϕ_{i_1,j_1} to denote $\alpha(i_2, j_2)$. That is, the data in page p_{i_1,j_1} need to be moved into block $B_{\phi_{i_1,j_1}}$ in the end.

Definition 2. *Let $S \subseteq [n+\Delta]$. The blocks $\{B_i | i \in S\}$ are called "semi-sorted" if there exists a bijection function $\pi : S \rightarrow S$ (that is, a permutation of the elements in S) such that $\forall\, i_1 \neq i_2 \in S$ and $j_1, j_2 \in [m]$, if $\pi(i_1) < \pi(i_2)$, then $\phi_{i_1,j_1} \leqslant \phi_{i_2,j_2}$.*

Let's use the help of semi-sorted blocks to move data. For simplicity, in the following we assume n is a power of Δ, namely, $n = \Delta^z$ for some integer z. We will extend the results for general n later. As the beginning step, we partition $[n]$ into n/Δ subsets $S_1, \cdots, S_{n/\Delta}$, where $|S_i| = \Delta$ for $1 \leqslant i \leqslant n/\Delta$. We create n/Δ sets of semi-sorted blocks as follows. We first copy the data in $\{B_i | i \in S_1\}$ into the Δ auxiliary blocks $\{B_i | n+1 \leqslant i \leqslant n+\Delta\}$ such that the blocks $\{B_i | n+1 \leqslant i \leqslant n+\Delta\}$ become semi-sorted blocks. Then for $i = 2, \ldots, n/\Delta$, we erase the blocks $\{B_j | j \in S_{i-1}\}$, and copy the data in $\{B_j | j \in S_i\}$ into $\{B_j | j \in S_{i-1}\}$ such that the blocks $\{B_j | j \in S_{i-1}\}$ become semi-sorted. Finally we erase the blocks $\{B_i | i \in S_{n/\Delta}\}$. This way, we have created n/Δ sets of semi-sorted blocks using n erasures.

Clearly, our final objective is to make $\{B_1, \cdots, B_n\}$ one big set of semi-sorted blocks. How can we combine the n/Δ smaller sets of semi-sorted blocks we already have to achieve this objective? Let's use $T_1, \ldots, T_{n/\Delta}$ to denote our n/Δ sets of semi-sorted blocks, and use T_0 to denote the set of Δ empty blocks. We first show how to combine $T_1, \cdots, T_\Delta$ into one bigger set of semi-sorted blocks. We copy data from $T_1, \ldots, T_\Delta$ into the empty blocks (one empty block at a time) with the following rule: "for two pages p_{i_1,j_1} and p_{i_2,j_2} of $T_1, \ldots, T_\Delta$, if $\phi_{i_1,j_1} < \phi_{i_2,j_2}$, then the data in page p_{i_1,j_1} is copied before the data in p_{i_2,j_2}." (Note that for every T_i ($1 \leqslant i \leqslant \Delta$), the data in it are already sorted, so they just need to be copied sequentially block after block.) For a block in T_i ($1 \leqslant i \leqslant \Delta$), once its data are all copied, we erase it so that it becomes empty and can have data moved into it later. Can we keep moving data this way so that in the end, the data in $T_1, \cdots, T_\Delta$ are moved into Δ^2 blocks, which are semi-sorted? The answer is yes, because in the above procedure there is always place to move data into: *Every time we have completely filled a number of empty blocks, if we look at the blocks in $T_1, \ldots, T_\Delta$ whose data have been partially copied, their un-copied data together can fill at most $\Delta - 1$ empty blocks. Since we have Δ empty blocks to begin with, there is always an empty block to copy data into.*

Using the same method, we can combine the n/Δ sets of semi-sorted blocks into n/Δ^2 bigger sets of semi-sorted blocks. By repeatedly using this approach $\log_\Delta n$ times, we can get n blocks that are semi-sorted. All left to do is to move the data into their final positions, which takes at most $3n/2$ erasures. In total, this algorithm uses at most $n\log_\Delta n + \frac{3n}{2}$ erasures. For general values of n, this algorithm uses at most $n\lceil\log_\Delta n\rceil + \frac{3n}{2}$ erasures.

B. Algorithm based on Block-permutation Sets

We now present an algorithm that uses $O(nm)$ erasures.

Definition 3. *A set of n pages $p_{1,j_1}, p_{2,j_2}, \ldots, p_{n,j_n}$ is a "block-permutation set" if $\{\alpha(1,j_1), \alpha(2,j_2), \ldots, \alpha(n,j_n)\} = [n]$.*

It is known that the nm pages in $B_1, \ldots, B_n$ can be partitioned into exactly m block-permutation sets [10]. Without loss of generality (w.l.o.g.), let's assume that for $i = 1, \ldots, m$, the n pages $p_{1,i}, p_{2,i}, \ldots, p_{n,i}$ form a block-permutation set. (Since a block contributes exactly one page to every block-permutation set, this is just a matter of labelling.) By definition, the data of the n pages in a block-permutation set need to be permuted in the n blocks $B_1, \ldots, B_n$. In the following, we use this property to move data, using only two auxiliary blocks (which we will call B_{n+1} and B_{n+2}).

Consider a block-permutation set $p_{1,j}, p_{2,j} \ldots, p_{n,j}$. Since a permutation consists of "permutation cycles", let's consider such a cycle of length $z \leqslant n$: $p_{i_0,j}, p_{i_1,j}, \ldots, p_{i_{z-1},j}$. That is, for $k = 0, 1, \ldots, z-1$, $\alpha(i_k, j) = i_{(k+1) \mod z}$. With the two auxiliary blocks, we can move the data inside the cycle to their right places without moving any data outside the cycle. The basic idea is that with one auxiliary block, we can cyclically shift the data inside the cycle. With the other auxiliary block, we can use it to temporarily hold the data outside the cycle when the corresponding block is erased. Specifically, we can move the data of the cycle to their right places this way:

1) Copy the data $d_{i_{z-1},j}$ from $p_{i_{z-1},j}$ to $p_{n+1,j}$.
   For $j' \in [m] \setminus \{j\}$, copy the data $d_{i_{z-1},j'}$ from $p_{i_{z-1},j'}$ to $p_{n+2,j'}$.
   Erase $B_{i_{z-1}}$.
2) For $k = z-2, z-3, \ldots, 0$, do:
   a) Copy the data $d_{i_k,j}$ from $p_{i_k,j}$ to $p_{i_{k+1},j}$.
   b) For $j' \in [m] \setminus \{j\}$, copy the data $d_{i_{k+1},j'}$ from $p_{n+2,j'}$ to $p_{i_{k+1},j'}$.
   c) Erase B_{n+2}.
   d) For $j' \in [m] \setminus \{j\}$, copy the data $d_{i_k,j}$ from $p_{i_k,j}$ to $p_{n+2,j'}$.
   e) Erase B_{i_k};
3) Copy the data $d_{i_{z-1},j}$ from $p_{n+1,j}$ to $d_{i_0,j}$.
   For $j' \in [m] \setminus \{j\}$, copy the data $d_{i_0,j'}$ from $p_{n+2,j'}$ to $p_{i_0,j'}$.
   Erase B_{n+1} and B_{n+2}.

With the outlined procedure, we can move z pages of data in a cycle using $2z+1$ erasures. In the same way, we can move the n pages of data in a block-permutation set using at most $5n/2$ erasures. Furthermore, we can move all the nm pages using $5nm/2$ erasures.

We have designed another efficient data-movement algorithm, called the *bit-fixing algorithm*, that allows straightforward implementation in flash memories. We present it in the appendix. All in all, we conclude:

Theorem 4. *Let $\Delta \geqslant 2$. When coding is not used, the data movement problem can be solved using at most*

$$\min\{n\lceil \log_\Delta n \rceil + \frac{3n}{2}, \frac{5nm}{2}\} = \Theta(n \cdot \min\{m, \log_\Delta n\})$$

erasures.

IV. A Lower Bound

In this section, we prove that without coding, a data-movement algorithm needs $\Omega(n \cdot \min\{m, (\log_\Delta n)/(\log^*_\Delta n)\})$ erasures.[1] Since $\log^*_\Delta n$ is practically a very small number, this lower bound is very close to the upper bound shown in Theorem 4.

A. Model

In this section, to simplify our notation, we use an equivalent model defined as follows: Our data is modeled as an $m \times (n+\Delta)$ matrix $A = (a_{i,j})_{m \times (n+\Delta)}$. Initially, the data in the sub matrix consisting of the first n columns contains all the elements in the set $[mn]$; and the remaining (auxiliary) entries are empty. Namely, each of the mn entries in the first n columns have distinct values, and all $x \in [mn]$ appear in these columns. The value x of $a_{i,j}$ indicates that the data in page $p_{i,j}$ needs to be moved to location i', j' where $x = (i'-1) + (j'-1) * m$ (namely, $i'-1 = (x \mod m)$ and $j'-1 = \lfloor x/m \rfloor$). In other words, after the data movement, the final matrix A^* will include the elements of $[mn]$ in the first n columns in *increasing* order; the remaining Δ columns will be empty. We use B_i to denote the *set of values* that appear in the i-th column of A ($B_i \subset [mn]$). We assume throughout that $\Delta \leqslant n$ (otherwise the problems studied throughout this work become trivial).

In general, our objective is to perform operations to the initial matrix A, such that the data in each page is moved to its corresponding location. We allow two types of operations: (a) *Copying* data from a current page to an empty page. This corresponds to writing the value of an entry a_{ij} into a location that was previously empty. Copying data is thought of as a

[1] Here $\log^*_\Delta n$ is the *iterated logrithm* of n, which is defined as the number of times the logrithm function must be iteratively applied before the result is less than or equal to 1. Namely, $\log^*_\Delta n = 1 + \log^*_\Delta(\log_\Delta n)$ for $n > \Delta$. Notice that $\log^*_\Delta n$ grows very slowly with n.

costless operation. (b) *Erasing* a column of data. This corresponds to erasing the value of all entries in a given column. After the operation, all entries in the given column will be empty. Erasing data is thought of as a costly operation. Given an initial matrix, using the two operations above, our objective is to reach the *final* matrix A^* specified above. In this section, we present lower bounds on the number of erasures needed to reach this goal.

B. Proof outline

To prove our lower bounds, we consider the directed *configuration* graph $G = (S, E)$ of our process. The vertex set S of our graph will include all possible matrices A that can be reached by performing operations (a) and (b) above on the initial matrix A. To be precise, to allow a clean analysis, we will restrict ourselves to the set of matrices A', reachable from A, that have exactly Δm empty entries; and that the remaining mn entries include the set $[mn]$ (one entry per element). It is not hard to verify that this is without loss of generality. On one hand for every $x \in [mn]$ our intermediate matrix A' must include an entry of value x, otherwise we will not be able to reach the final matrix A^*. On the other, we may assume that there are exactly Δm empty entries as any value $x \in [mn]$ that appears in two positions is redundant. We define a slightly different operation on our matrix that preserves the above restrictions and combines operations (a) and (b) above. (ab) *Erase and copy*: Erase a column of data, and copy its content onto the empty locations in the matrix (these empty locations may include the currently erased column). Our objective is, starting from our initial matrix A, to perform a minimum number of (ab) operations to reach A^*.

We have yet to define the edge set E of our graph G. A pair (A', A'') is a directed edge in E if using a single (ab) operation one can transform A' into A''. The *distance* between a matrix A and A^* is the shortest path in G between A and A^*, and corresponds to the minimum number of (ab) operations needed to transform A into the desired matrix A^*. We will show the existence of an initial matrix A for which this distance is at least a certain lower bound lb.

Our proof has two steps. First, we show that the diameter of G is at least lb (namely there exist two matrices A' and A'' which are *far* apart.

Theorem 5. *The diameter of G is at least $lb = \Theta(n \cdot \min(m, \log_\Delta n / \log^*_\Delta n))$.*

We then show that this suffices to prove our assertion.

Corollary 6. *There exists a matrix A such that the distance between A and A^* is at least $\Theta(n \cdot \min(m, \log_\Delta n / \log^*_\Delta n))$.*

The proof of Corollary 6 is fairly simple and is given in full detail below. The proof of Theorem 5 is more involved and an outline followed by a detailed proof will be given after the proof of Corollary 6.

1) Proof of Corollary 6: *Proof:* Let A' and A'' be the matrices and lb be the bound from Theorem 5. Let $d(A', A'')$ denote the distance between two matrices A' and A'' in G. It holds (by our distance definition) that $d(A', A^*) + d(A^*, A'') \geqslant d(A', A'') \geqslant lb$. Thus either $d(A', A^*) \geqslant lb/2$ and we are done (set $A = A'$); or $d(A^*, A'') \geqslant lb/2$. Notice that in the latter case our proof is not done as we wish to find a matrix A such that $d(A, A^*)$ is large, while we are in the case that $d(A^*, A)$ is large (here we stress that the distance d is not symmetric).

To overcome this difficulty we will define two matrices A_1 and A_2 as follows. We start by noticing that there exists a matrix, which we denote by A_1, *close* to A'' for which the empty entries of A_1 are all located in the last Δ columns. More specifically, $d(A'', A_1) \leqslant \Delta$ and $d(A_1, A'') \leqslant n$. The matrix A_1 is obtained from A'' by *moving* all the non-empty entries in the last Δ columns to empty locations in the first n columns. Now it holds that $d(A^*, A_1) \geqslant lb/2 - n$ (otherwise $d(A^*, A'') \leqslant d(A^*, A_1) + d(A_1, A'') < lb/2$ which contradicts our assumption). Finally, we define a matrix A_2 corresponding to A_1 for which $d(A_2, A^*) = d(A^*, A_1) \geqslant lb/2 - n$. Setting A to be equal to A_2 we conclude our proof.

The matrix A_2 is obtained from the matrix A^* by a permutation on the values of $[mn]$; the permutation is defined by the matrix A_1. Specifically, let π be the permutation on $[mn]$ such that $\pi(A_1) = A^*$ (here for a matrix $A = (a_{ij})$ we define the (i, j)'th entry of $\pi(A)$ as $\pi(a_{i,j})$). Now setting $A_2 = \pi(A^*)$ and noticing that for any permutation π there is an edge in G between two matrices X and Y iff there is an edge in G between $\pi(X)$ and $\pi(Y)$, it holds that $d(A^*, A_1) = d(\pi(A^*), \pi(A_1)) = d(A_2, A^*) \geqslant lb/2 - n$. We note that it is not hard to show the existence of a (different) matrix A such that the distance between A and A^* is at least n (take any matrix A for which in each column some entry needs to be moved to obtain A^*). Thus, for the corollary, we obtain a bound of $\max(lb/2 - n, n) = \Theta(n \cdot \min(m, \log_\Delta n / \log^*_\Delta n))$. ■

2) Outline for proof of Theorem 5: To prove Theorem 5 (our lower bound on the diameter of G) we use a variation of the following naive idea. For a matrix A, let d_A be the *out-degree* of A in G. Assume one could prove that d_A is bounded by D for all matrices $A \in S$. This implies that a *walk* of length ℓ in G starting from A can reach at most D^ℓ different matrices of S. One can now deduce that the diameter of G is at least the smallest ℓ such that $D^\ell \geqslant |S|$; or in other words $lb \geqslant \log_D |S|$.

Applying this proof technique "as is" on our graph G will not yield a lower bound greater than n. We thus consider two modifications. First of all, we consider a slightly different graph $G' = (S', E')$, which is a homomorphic image of G. Namely, for a given matrix $A \in S$ let B_i be the set values that appear in the i'th column of A. We will identify A with the tuple $B_A = (B_1, \ldots, B_{n+\Delta})$; and $S' = \{B_A \mid A \in S\}$ will consist of the set of such tuples. Two tuples $B = (B_1, \ldots, B_{n+\Delta})$ and $B' = (B'_1, \ldots, B'_{n+\Delta})$ are connected by a directed edge in E' iff there exists matrices A and A' with corresponding tuples B

and B' that are connected in G. It is not hard to verify that the diameter of G is no less than the diameter of G'. (In fact, we can also show that the diameter of G is no more than the diameter of G' plus $n+\Delta$. So the two diameters are the same up to an additive factor of $n+\Delta$.) We conclude that a lower bound of $\Theta(n \cdot \min(m, \log_\Delta n / \log^*_\Delta n))$ for the diameter of G' will imply the same lower bound for G.

Up to this point we have discussed the first variation of the naive idea presented above: replacing the graph G by G'. However, bounding the degree D' of G' will not suffice to obtain a lower bound of $\log_{D'} |S'|$ larger than n. The main reason is that the maximum degree D' of G' is too large. But we have noticed that only rarely may one visit vertices of G' with large degree approaching this bound. Typically the degree of the vertices at hand will be fairly small. To utilize this observation, we consider the maximum number of vertices in S' reachable from a given vertex B by $\alpha > 1$ steps instead of a single step (for a single step, this value is exactly the vertex out-degree). This corresponds to the study of the α transitive closure of G' sometimes denoted as $(G')^\alpha$. Namely, $(G')^\alpha$ consists of the vertex set S', where two vertices are connected by an edge iff there is a path of length at most α between them in G'. Clearly, if $(G')^\alpha$ has diameter $diam$, then G' has diameter at least $diam \cdot \alpha$. Studying $(G')^\alpha$ instead of G' allows us to *average out* the differences between the degree of vertices in G' and to obtain the desired bound. In what follows we analyze the size of S' and the value Deg_α of the maximum out-degree in $(G')^\alpha$. We then deduce a bound on the diameter of G' of $lb \geqslant \Theta\left(\alpha \log_{Deg_\alpha} |S'|\right)$ which in turn implies the assertion of Theorem 5. In what follows, for two functions f and g, the notation $f \simeq g$ will represent the fact that $\log f = \Theta(\log g)$. We also assume that $\Delta < n$ (otherwise there is a trivial lower and upper bound of n).

C. The state space S'

We start by bounding (from below) the total number of configurations in the state space S' at hand.

$$|S'| \quad \geqslant \quad \binom{nm+\Delta m}{\Delta m} \frac{(nm)!}{(m!)^n} \geqslant n^{nm}$$

We explain our bound: We first choose the Δm location for the empty entries of our data matrix. This determines the size of the sets $B_1, \ldots, B_{n+\Delta}$, say $m_1, \ldots, m_{n+\Delta}$ where each m_i is at most m. We then decide on the content of each set B_i. If the sets were ordered, the number of configurations would be exactly $(nm)!$. As they are not ordered this number should be divided by $\Pi_{i=1}^{n+\Delta}(m_i)!$. It is not hard to verify that $\Pi_{i=1}^{n+\Delta}(m_i)! \leqslant (m!)^n$ as $\sum_{i=1}^{n+\Delta} m_i = nm$ and each m_i is at most m.

D. Deg_α: number of vertices reachable in α steps

We now bound (from above) the number of vertices Deg_α that can be reached using α steps from any given initial vertex in S'. We take $n > \alpha > \Delta$. We define $D(\alpha)$ to be the number of different configurations a certain set of α columns can take in α steps.

$$Deg_\alpha \quad \leqslant \quad \binom{n+\Delta}{\alpha}\binom{\alpha m}{\Delta m}(n+\Delta)^{\Delta m} D(\alpha) \simeq n^{\alpha+\Delta m} D(\alpha)$$

We explain our bound: We first pick α columns out of the $n+\Delta$ columns. There are now two types of changes that may have been made in the data matrix, changes inside the α columns we picked (referred to as internal columns) and changes outside these columns (referred to as external columns). We start with external changes. The external empty spaces may have been filled with elements from the internal columns. This can be bounded by $\binom{\alpha m}{\Delta m}$ to choose the internal elements, and $(n+\Delta)^{\Delta m}$ to distribute them among the external empty spaces (notice that there are at most Δm such empty spaces). Now we are left to consider the number of internal configurations one may obtain. We denote this value by $D(\alpha)$. Namely, $D(\alpha)$ equals the number of possible configurations obtainable in a given set of α columns when erasing these columns one after the other. It is clear that $D(\alpha) \leqslant \alpha^{m\alpha}$, Indeed, for a rough bound notice that internal entries must appear in one of the α columns. Plugging in this value of $D(\alpha)$ will already yield nice results. However, to tighten the results we compute $D(\alpha)$ recursively. Namely, for $\alpha > \beta > \Delta$ it holds that

$$D(\alpha) \quad \leqslant \quad \left[\binom{\alpha}{\beta}\binom{\beta m}{\Delta m}\alpha^{\Delta m} D(\beta)\right]^{\frac{\alpha}{\beta}} \simeq \left[\alpha^{\beta+\Delta m} D(\beta)\right]^{\frac{\alpha}{\beta}}$$

Again, we explain our bound. We would like to express the number of internal configuration obtainable in α steps $D(\alpha)$ by the number of internal configurations obtainable in β steps $D(\beta)$ for β smaller than α. The analysis is similar to the previous one for Deg_α. We first compute how many configurations can be obtained in β steps, and then raise this number by α/β. We start by picking β columns out of the α at hand. For the upcoming discussion, we refer to the α columns as external columns and to the β columns as internal columns. Again, there may be two types of changes in the configuration, external and internal. For external changes, we may fill some of the empty entries of the external columns with internal entries. This is counted for by $\binom{\alpha}{\beta}\binom{\beta m}{\Delta m}\alpha^{\Delta m}$. As before, the internal changes are attributed to $D(\beta)$.

We now compute our lower bound. We have the freedom to fix the values of α and β. For the first level of recursion, we fix $\alpha = \Delta \log_\Delta n$ and $\beta = \Delta \log_\Delta \log_\Delta n = \Delta \log_\Delta^{(2)} n$. It holds that $n > \alpha > \beta$. Thus, $Deg_\alpha \simeq n^{\alpha+\Delta m} D(\alpha)$ is approximately

$$n^{\Delta \log_\Delta n + \Delta m} \left[(\Delta \log_\Delta n)^{\Delta \log_\Delta^{(2)} n + \Delta m} D(\Delta \log_\Delta^{(2)} n) \right]^{\frac{\log_\Delta n}{\log_\Delta^{(2)} n}}$$

For $\Delta < \log_\Delta n$ it holds that

$$n^{\frac{\log_\Delta^{(2)} n}{\log_\Delta n}} \simeq \Delta \log_\Delta n$$

Thus we have that $Deg_\alpha \simeq$

$$n^{\Delta \log_\Delta n + \Delta \log_\Delta^{(2)} n} n^{2\Delta m} D(\Delta \log_\Delta^{(2)} n)^{\frac{\log_\Delta n}{\log_\Delta^{(2)} n}}$$

For the second step of our recursion, we need to compute $D(\alpha') = D(\Delta \log_\Delta^{(2)} n)$. We do this by fixing $\alpha' = \Delta \log_\Delta^{(2)} n$ and $\beta' = \Delta \log_\Delta^{(3)} n$. For $\Delta < \log_\Delta^{(2)} n$ it holds that

$$n^{\frac{\log_\Delta^{(3)} n}{\log_\Delta^{n}}} \simeq \Delta \log_\Delta^{(2)} n$$

Thus we have that $Deg_\alpha \simeq$

$$n^{\sum_{i=1}^{3} (\Delta \log_\Delta^{(i)} n + \Delta m)} D(\Delta \log_\Delta^{(3)} n)^{\frac{\log_\Delta n}{\log_\Delta^{(3)} n}}$$

In general, we can continue the recursion as long as $\Delta < \log_\Delta^{(i)} n$, while for the base we take $D(\Delta) = \Delta^{m\Delta}$. So all in all we get

$$Deg_\alpha \quad \leqslant \quad n^{\sum_{i=1}^{\log_\Delta^* n} (\Delta \log_\Delta^{(i)} n + \Delta m)} \Delta^{m\Delta \frac{\log_\Delta n}{\Delta}} \simeq n^{\Delta m \log_\Delta^* n} n^{\Delta \log_\Delta n}$$

Finally, we evaluate our lower bound

$$lb \quad \geqslant \quad \alpha \frac{\log_\Delta |S'|}{\log_\Delta Deg_\alpha} \simeq \frac{\Delta m n \log_\Delta n}{\Delta m \log_\Delta^* n + \Delta \log_\Delta n} = \frac{n}{\log_\Delta^* n / \log_\Delta n + 1/m} \tag{1}$$

Which implies a lower bound of

$$lb = \Theta(n \cdot \min(m, \log_\Delta n / \log_\Delta^* n))$$

Almost matching our upper bound of $\Theta(n \min(m, \log_\Delta n))$.

V. Efficient Coding-based Data Movement

We now focus our attention on data movement using coding, where $\Delta = 1$. It is known that with coding, $2n-1$ erasures are sufficient and necessary in the worst case. What about minimizing the number of erasures for each given instance, instead of just the worst case? (Certainly, for some instances, fewer than $2n-1$ erasures are needed.) Our coding-based solution will use the concept of "canonical labelling."

Definition 7. *[10] Let $y \in [n-2]$. When we relabel the n blocks $B_1, \ldots, B_n$ as $B'_1, \ldots, B'_n$, it is called a "canonical labelling with parameter y" if for any $y+1 \leqslant i \leqslant n-2$ and $i+2 \leqslant j \leqslant n$, no data in B'_j need to be moved into B'_i.*

The following observation has been made in [10]: "Given an instance of the data-movement problem with $\Delta = 1$, there exists a coding-based solution using $n+y+1$ erasures if and only if there exists a canonical labelling with parameters y." It is NP hard to find a canonical labelling with y minimized [10]. However, we should notice that when $y = n-2$, any labelling is a canonical labelling, and that would give us the $n+y+1 = 2n-1$ erasures, which is worst-case optimal.

We present a very efficient coding-based algorithm that uses $n+y+1$ erasures once a canonical labelling with parameter y is given. In the algorithm, a very small Galois field is used for computation ($GF(q)$ with $q \geqslant 3$), and it generates parity-check symbols with a small number of additions on average. Before this work, it was unknown how to find such strictly-optimal and efficient solutions over small Galois fields [10]. Our algorithm achieves low coding complexity while minimizing the number of erasures.

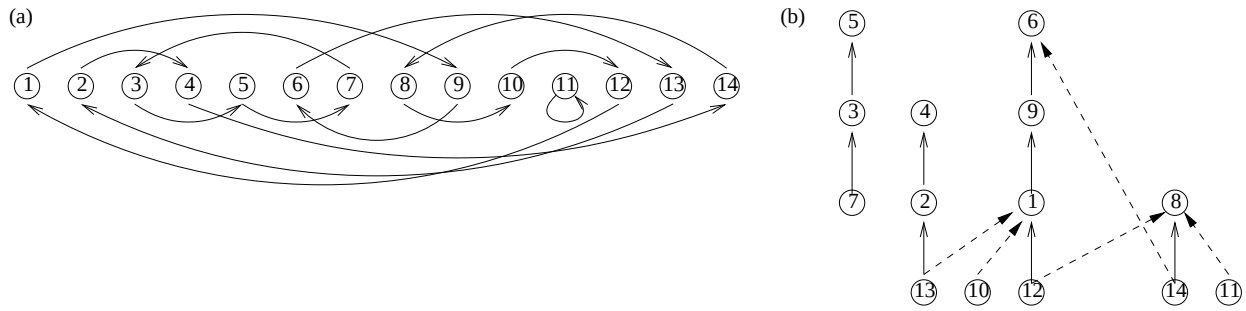

Figure 1. Example of the data-movement problem with $n = 14$. We let $y = 8$. (a) The data-movement graph G_d. If the data d_i needs to be moved to page p_j, there is an edge from vertex i to vertex j. (b) The symbol graph G_s. The solid edges are the *black edges*, and the dashed edges are the *red edges*.

A. Coding-based Algorithm

The algorithm takes a canonical labelling with parameter y as input. For simplicity, let us say that the original labelling – $B_1, B_2, \ldots, B_n$ – is the canonical labelling. (It is just a matter of naming.) It has been shown in Section III that the nm pages of $B_1, \ldots, B_n$ can be partitioned into m block-permutation sets. Our algorithm will work the same way for the m block-permutation sets. So for simplicity, in the following, we will consider only one block-permutation set, and call its pages $p_1, p_2, \ldots, p_n$. For $i \in [n]$, p_i is the page in block B_i and originally holds the data d_i. For $i \in [n]$, the function $\alpha(i) = j$ means that the data d_i need to be moved into the page p_j. (Here $j \in [n]$.) By the definition of block-permutation set, we know that $\{\alpha(1), \ldots, \alpha(n)\} = \{1, \ldots, n\}$. If $\alpha(i) = j$, we say $\alpha^{-1}(j) = i$. By the definition of canonical labelling with parameter y, we know that for $i \in \{y+1, y+2, \ldots, n-2\}$, $\alpha^{-1}(i) \leqslant i+1$.

We will use only one auxiliary block. We call it B_0, and call its page p_0. In our algorithm, $B_1, \ldots, B_y$ will be erased twice, while B_0 and $B_{y+1}, \ldots, B_n$ will be erased only once.

We define a function $\beta(i)$ for $i \in \{y+2, y+3, \ldots, n\}$:

- If $\alpha^{-1}(i-1) \leqslant y+1$ and $\alpha^{-1}(i-1) \neq \alpha(i)$, then $\beta(i) = \alpha^{-1}(i-1)$.
- If $y+2 \leqslant \alpha^{-1}(i-1) \leqslant i-1$, then $\beta(i) = \beta(\alpha^{-1}(i-1))$.
- If $\alpha^{-1}(i-1) = i$, or if $\alpha^{-1}(i-1) \leqslant y+1$ and $\alpha^{-1}(i-1) = \alpha(i)$, then $\beta(i) = NULL$.

We define the set γ_i, for $i \in [y+1]$, as follows:

$$\gamma_i = \{j \mid \beta(j) = i, y+2 \leqslant j \leqslant n\}.$$

Example 8. *An example is shown in Fig.* 1 *(a). Here $n = 14$ and $y = 8$. For $i \in [n]$, if $\alpha(i) = j$ (that is, the data d_i need to be moved into page p_j), we draw an edge from vertex i to vertex j. We call the graph the* data-movement graph G_d.

Here we have $\alpha(1) = 9$, $\alpha(2) = 4$, $\ldots$, $\alpha(14) = 8$, and $\alpha^{-1}(1) = 12$, $\alpha^{-1}(2) = 13$, $\ldots$, $\alpha^{-1}(14) = 4$. We also have $\beta(10) = 1$, $\beta(11) = 8$, $\beta(12) = 8$, $\beta(13) = 1$, $\beta(14) = 6$. Correspondingly, $\gamma_1 = \{10, 13\}$, $\gamma_6 = \{14\}$, $\gamma_8 = \{11, 12\}$. When $i \in [y+1]$ and $i \neq 1, 6, 8$, $\gamma_i = \emptyset$.

It is well known that a permutation consists of cycles.

Definition 9. *A set of pages $p_{i_0}, p_{i_1}, \ldots, p_{i_{x-1}}$ is called a* permutation cycle *if for $j = 0, 1, \ldots, x-1$, $\alpha(i_j) = i_{j+1 \bmod x}$. Here $i_j \in [n]$ for all j. Among the x pages, $p_{\max\{i_j \mid 0 \leqslant j \leqslant x-1\}}$ is called the* tail *of the permutation cycle.*

Example 10. *The data-movement problem shown in Fig.* 1 *(a) has three permutation cycles: (1) p_1, p_9, p_6, p_{13}, p_2, p_4, p_{14}, p_8, p_{10}, p_{12}; (2) p_3, p_5, p_7; (3) p_{11}. Their tails are p_{14}, p_7 and p_{11}, respectively.*

Let us build a directed graph G_s, called the *symbol graph*, as follows. G_s has n vertices labelled by $1, \ldots, n$, which correspond to the n pages $p_1, \ldots, p_n$. The edges of G_s have two colors: black and red. There is a black edge from vertex i to vertex j if $\alpha(i) = j$, $j \in [y+1]$ and p_j is not the tail of its permutation cycle. There is a red edge from vertex i to vertex j if $y+2 \leqslant i \leqslant n$, $j \in [y+1]$ and $i \in \gamma_j$.

Example 11. *An example of the symbol graph G_s is shown in Fig.* 1 *(b). It corresponds to the data movement problem in Fig.* 1 *(a). The black edges of G_s always form disjoint paths. In this example, the paths are: (1)* $7 \to 3 \to 5$*; (2)* $13 \to 2 \to 4$*; (3)* 10*; (4)* $12 \to 1 \to 9 \to 6$*; (5)* $14 \to 8$*; (6)* 11*. We call them* black paths*. The "sources" of the above six black paths are vertices* $7, 13, 10, 12, 14, 11$*, respectively, and their "sinks" are vertices* $5, 4, 10, 6, 8, 11$*, respectively.*

It is easy to see that a black path can have at most one vertex from $\{y+2, y+3, \ldots, n\}$; and if it does, that vertex must be its source. For a red edge, its beginning point must be the source of a black path. The end point of a red edge is either the end of a black path or the vertex $\alpha^{-1}(y+1)$.

We show that the symbol graph G_s has a simple structure. Let $\overline{G}_s$ be the undirected version of G_s. That is, if we covert all the edges of G_s to be undirected edges, we get $\overline{G}_s$.

Lemma 12. *Every connected component of $\overline{G}_s$ has at most one cycle.*

Proof: Let us remove all the vertices of $\overline{G}_s$ that are not in any cycle, and call the remaining graph G_0. Every cycle in G_0 must have both black and red edges. Note that the black edges belong to the disjoint black paths, and the red edges have the properties described in Example 11. Let x denote the number of red edges in G_0. The beginning point of the x red edges (if we view them as directed edges) belong to x different black paths. Since every vertex in G_0 is in a cycle, and every beginning point of a red edge (if we view it as a directed edge) of G_0 is incident to exactly one red edge and one black edge, each of those x black paths must also contain exactly one end point of a red edge (if we view it as a directed edge). So every vertex in G_0 has degree two. So G_0 consists of vertex-disjoint cycles. So all the cycles in $\overline{G}_s$ are vertex-disjoint. If a path in $\overline{G}_s$ connects two cycles, the path would start with a red edge; but that red edge would have to share the starting point of another red edge – the latter edge is in a cycle – which would be impossible. So every connected component of $\overline{G}_s$ has at most one cycle. ■

We define a function $w(j)$ for $j \in \{y+2, y+3, \ldots, n\}$. Given a cycle C in $\overline{G}_s$, let us call its vertices $i_1, i_2, \ldots, i_x$. Then for $j \in \{i_1, i_2, \ldots, i_x\} \cap \{y+2, y+3, \ldots, n\}$, let $w(j) = -1$ if $j = \max\{\{i_1, i_2, \ldots, i_x\} \cap \{y+2, y+3, \ldots, n\}\}$, and $w(j) = 1$ otherwise. For a vertex $j \in \{y+2, y+3, \ldots, n\}$ not in any cycle of $\overline{G}_s$, let $w(j) = 1$.

We now present the coding-based algorithm. It uses $n+y+1$ erasures to move data. For simplicity, we choose the computation to be over $GF(3)$, whose elements are $\{0, 1, -1\}$. It is also feasible to use $GF(q)$ with $q > 3$.

Algorithm 13 CODING-BASED DATA MOVEMENT

Step 1: For $i = 1, 2, \ldots, y+1$, do:

- *If p_i is not the* tail *of its permutation cycle, write the data $d_i - d_{\alpha^{-1}(i)} - \sum_{j \in \gamma(i)} w(j) d_j$ into the page p_{i-1}; otherwise, write the data $d_i - \sum_{j \in \gamma(i)} w(j) d_j$ into the page p_{i-1}.*
- *Erase the block B_i.*

Step 2: For $i = y+1, y+2, \ldots, n-1$, write the data $d_{\alpha^{-1}(i)}$ into the page p_i, then erase the block B_{i+1}.

Step 3: Write the data $d_{\alpha^{-1}(n)}$ into the page p_n, then erase the block B_y.

Step 4: For $i = y, y-1, \ldots, 1$, write the data $d_{\alpha^{-1}(i)}$ into the page p_i, then erase the block B_{i-1}.

Example 14. *Let the data movement problem be as shown in Fig.* 1 *(a), and we use Algorithm* 13 *to move data. The data stored in the pages $p_0, \ldots, p_{14}$ during the data movement process are shown in Fig.* 2*. Note that $p_1, \ldots, p_y$ are erased twice each, while $p_{y+1}, \ldots, p_n$ and p_0 are erased once each. The total number of erasures is $n+y+1$.*

pages	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}	p_{14}
data		d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}	d_{13}	d_{14}
	$d_1 - d_{12}$ $-d_{10} - d_{13}$	$d_2 - d_{13}$	$d_3 - d_7$	$d_4 - d_2$	$d_5 - d_3$	$d_6 - d_9$ $+d_{14}$	d_7	$d_8 - d_{14}$ $-d_{12} - d_{11}$	$d_9 - d_1$	d_1	d_8	d_{11}	d_{10}	d_6	d_4
		d_{12}	d_{13}	d_7	d_2	d_3	d_9	d_5	d_{14}						

Figure 2. Example of the coding-based data movement algorithm.

B. Analysis

To prove that the algorithm is correct, the key is to prove that at any moment of the data-movement process, all the data can be recovered by decoding the data currently stored in the pages. For that, it is helpful to observe the one-to-one correspondence between the data written into the pages and the vertices/edges of the symbol graph G_s: *Every vertex $i \in [n]$ of G_s corresponds to the data d_i, and the edges entering a vertex i correspond to the data written into p_{i-1} before p_i is erased.* (That is why G_s is called the *symbol graph*.) For example, let us look at vertex 1 in Fig. 1 (b). It has three incoming edges, respectively from vertex 12, 10 and 13. Those incoming edges correspond to the symbol $d_1 - d_{12} - d_{10} - d_{13}$. From Fig. 2, we see that was the symbol written into p_0 (before p_1 is erased by the algorithm for the first time). The vertex 1 itself corresponds to the symbol d_1, which was initially stored in p_1 and later written into p_9 (which are also shown in Fig. 2). Algorithm 13 always sequentially writes and erases symbol pairs sharing a common variable (i.e., d_i for some i), which correspond to a vertex and its adjacent edge(s) in G_s. This makes the stored symbols always linearly independent.

Theorem 15 *When Algorithm* 13 *is used to move data, at any moment, all the data $d_1, d_2, \ldots, d_n$ can be recovered by decoding the data currently stored in the pages $p_0, p_1, \ldots, p_n$.*

Proof: We analyze the four steps of the algorithm, and use the symbol graph G_s for explanation. Note the one-to-one correspondence between the data written into the pages and the vertices/edges of G_s, which is shown above.

In step 1, every time a data symbol d_i is erased (with $1 \leqslant i \leqslant y+1$), the symbol corresponding to the edges entering vertex i of G_s has been stored. So for any symbol d_j with $1 \leqslant j \leqslant y+1$, there is a set of stored symbols $d_{i_0}, d_{i_1} - d_{i_0} - f_0, d_{i_2} - d_{i_1} - f_1, \ldots, d_{i_x} - d_{i_{x-1}} - f_{x-1}$, such that $d_{i_k} = \alpha(d_{i_{k-1}})$ for $1 \leqslant k \leqslant x$, $d_{i_x} = d_j$, and $f_0, \ldots, f_{x-1}$ are linear functions of $d_{y+2}, d_{y+3}, \ldots, d_n$. The symbols $d_{y+2}, d_{y+3}, \ldots, d_n$ are all stored. So d_j can be decoded.

In step 2, consider a vertex i of G_s (with $1 \leqslant i \leqslant y+1$) and the vertices in $\gamma_i = \{i_1, i_2, \ldots, i_x\}$. Let $i_0 = i$, and let $i_1 < i_2 < \cdots < i_x$. Note that $(i_1, i_0), (i_2, i_0), \ldots, (i_x, i_0)$ are the red edges that share the same end point i_0 in G_s. For $j = 1, 2, \ldots, x$, right before the symbol d_{i_j} is erased (from page p_{i_j}), the symbol $d_{i_{j-1}}$ has been stored (in page $p_{i_j - 1}$). By the way we set the linear functions of the stored symbols and the fact that every connected component of G_s has at most one cycle, no subset of stored symbols can be linearly dependent. So all data can be recovered.

In steps 3 and 4, for any symbol d_j, either there is a set of stored symbols $d_{i_0}, d_{i_0} - d_{i_1} - f_0, d_{i_1} - d_{i_2} - f_1, \ldots, d_{i_{x-1}} - d_{i_x} - f_{x-1}$ with $d_{i_k} = \alpha(d_{i_{k+1}})$ for all k, or there is a set of stored symbols $d_{i_0}, d_{i_1} - d_{i_0} - f_0, d_{i_2} - d_{i_1} - f_1, \ldots, d_{i_x} - d_{i_{x-1}} - f_{x-1}$, with $d_{i_k} = \alpha(d_{i_{k-1}})$ for all k, such that $d_{i_x} = d_j$ and $f_0, \ldots, f_{x-1}$ are linear functions of $d_{y+2}, d_{y+3}, \ldots, d_n$. By the way the linear functions of the symbols are set and the fact that every connected component of G_s has at most one cycle, we see that d_j can be decoded. Therefore, Algorithm 13 can successfully move data using $n + y + 1 \leqslant 2n - 1$ erasures. ■

VI. Hardness of Optimizing Data Movement for Each Instance

In the last section, we have presented a coding-based algorithm that moves data using $n + y + 1 \leqslant 2n - 1$ erasures, once a canonical labelling of $B_1, \ldots, B_n$ with parameter $y \leqslant n - 2$ is given. On the other hand, there exists a data-movement solution using $n + y + 1$ erasures if and only if there exists a canonical labelling with parameter y. So the presented algorithm is strictly optimal. However, it is NP hard to find an optimal coding-based solution if the the canonical labelling with minimized parameter y is not given first [10]. A natural question is: What is the complexity of finding the best solution without coding for each specific instance of the data-movement problem? (Again, this optimization is per-instance instead of for the worst case.) We study this topic in this section.

We will prove the NP hardness of non-coding solutions for a slightly generalized version of the data-movement problem. Let's allow some original data to be just erased, instead of moved. More specifically, we change the function α to $\alpha : \{1, \ldots, n\} \times \{1, \ldots, m\} \to \{1, \ldots, n\} \cup \{\perp\}$, where $\alpha(i, j) = \perp$ means that the data $d_{i,j}$ just need to be erased, instead of moved. This is a very practical generalization, because in flash memories, there are usually pages whose data are no longer useful, and such pages are labelled as "invalid" in flash memories to be erased later [5]. Let us call this version the *generalized data-movement problem*.

For $i, j \in [n]$, let $d(i \to j)$ denote the number of pages of data that need to move from block B_i to block B_j. That is, $d(i \to j) = |\{k \mid k \in [m], \alpha(i, k) = j\}|$. We now define a concept called the *movement graph* G_m.

Definition 16. (Movement graph) *Corresponding to a generalized data-movement problem, we build a directed graph $G_m = (V, E)$ as follows. We let $V = \{v_1, v_2, \cdots, v_n\}$, where v_i represents the block B_i for $i \in [n]$. For any $i, j \in [n]$ and $i \neq j$, there are $d(i \to j)$ directed edges from vertex v_i to v_j. This graph $G_m = (V, E)$ is called the "*movement graph*."*

Definition 17. (Permuted Labelling, Ascending edges, and Descending edges) *Let π be a permutation of $\{1, 2, \cdots, n\}$. That is, $\pi(i) \in [n]$ for any $i \in [n]$, and $\pi(i) \neq \pi(j)$ for any $i \neq j \in [n]$. Let Π denote the set of all the $n!$ such permutations. Let π^{-1} be the inverse function of π.*

*Given a permutation π, in the movement graph $G_m = (V, E)$, we call an edge from v_i to v_j an "*ascending edge*" if $\pi^{-1}(i) < \pi^{-1}(j)$; we call it a "*descending edge*" if $\pi^{-1}(i) > \pi^{-1}(j)$. Let A_π denote the set of ascending edges and D_π denote the set of descending edges, given the permutation π. Note that $E = A_\pi \cup D_\pi$. The permutation π is also called a "*permuted labelling*" of the graph G_m.*

We first present a (tight) lower bound for the number of erasures. The concept of "*canonical data-movement solution*" used in the following proof will also be used later for proving NP hardness and approximation results.

Theorem 18. *For data-movement solutions without coding, the number of erasures needed for moving data is at least*

$$n + \lceil \frac{\min_{\pi \in \Pi} |A_\pi| + \sum_{i \in [n]} d(i \to i)}{m} \rceil.$$

And when $\Delta \geqslant \lceil \frac{\min_{\pi \in \Pi} |A_\pi| + \sum_{i \in [n]} d(i \to i)}{m} \rceil$, this bound is tight.

Proof: Let k denote a non-negative integer, and let π denote a permutation of $\{1, 2, \cdots, n\}$. We call a data-movement solution "*canonical*" if – for some k and π – it consists of the following three steps:

- Step 1: For $i = n+1, n+2, \cdots, n+k$, write data into the auxiliary block B_i.
- Step 2: For $i = \pi(1), \pi(2), \cdots, \pi(n)$, erase B_i and then write data into B_i.
- Step 3: For $i = n+1, n+2, \cdots, n+k$, erase the auxiliary block B_i.

Let $\mathcal{P}$ denote the generalized data movement problem. Let $\mathcal{P}_\infty$ denote the same data movement problem except that we change the number of auxiliary blocks from Δ to ∞. Clearly, the number of erasures needed for problem $\mathcal{P}$ is no less than that for problem $\mathcal{P}_\infty$. The idea of the proof is as follows:

- First, we show that for any solution to problem $\mathcal{P}$ that uses x erasures, there is a corresponding "*canonical*" solution to problem $\mathcal{P}_\infty$ that also uses x erasures;

- Next, we show that every "*canonical*" solution to problem $\mathcal{P}_\infty$ uses at least $n + \lceil \frac{\min_{\pi\in\Pi} |A_\pi| + \sum_{i\in[n]} d(i\to i)}{m} \rceil$ erasures.

Let us now prove the first step.

Let s denote a solution to problem $\mathcal{P}$ that uses x erasures. For $i = 1, 2, \cdots, n+\Delta$, let y_i denote the number of times that block B_i is erased in solution s. (Clearly, $x = \sum_{i=1}^{n+\Delta} y_i$.) We construct a "*canonical*" solution to problem $\mathcal{P}_\infty$ that uses x erasures in the following way:

- Let $z = n$.
- For $i = n+1, n+2, \cdots, n+\Delta$ and $j = 1, 2, \cdots, y_i$, do: Increase z by one, then write into the auxiliary block B_z the data that were written into B_i in the solution s between its $(j-1)$-th erasure and its j-th erasure.
- For $i = 1, 2, \cdots, n$ and $j = 1, 2, \cdots, y_i - 1$, do: Increase z by one, then write into the auxiliary block B_z the data that were written into B_i in the solution s between its i-th erasure and its $(i+1)$-th erasure.
- Let π be such a permutation of $\{1, 2, \cdots, n\}$: in the solution s, $\forall\ 1 \leqslant i < j \leqslant n$, the block $B_{\pi(i)}$ was erased for the last time (i.e., for its $y_{\pi(i)}$-th time) before the block $B_{\pi(j)}$ was erased for the last time (i.e., for its $y_{\pi(j)}$-th time).
- For $i = \pi(1), \pi(2), \cdots, \pi(n)$, erase block B_i, then write into B_i the data that were written into B_i in the solution s after its y_i-th erasure.
- For $i = n+1, n+2, \cdots, z$, erase the auxiliary B_i.

The above solution is indeed a feasible solution to problem $\mathcal{P}_\infty$ because of its correspondence to the solution s. What differs the new solution from the solution s is that instead of erasing data from a block, the new solution keeps the data in some auxiliary block. (And that makes the new solution feasible since no less data is preserved during the data movement process.) The new solution is a "*canonical*" solution, where the integer parameter $k = \sum_{i=n+1}^{n+\Delta} y_i + \sum_{i=1}^{n}(y_i - 1) = x - n$ and the permutation parameter π is as specified above. The new solution uses $k + n = x$ erasures. So the first step in our proof is shown to be correct.

Let us now prove the second step.

A "*canonical*" solution to problem $\mathcal{P}_\infty$ first writes data into auxiliary blocks, then erases the blocks $B_{\pi(1)}, B_{\pi(2)}, \cdots, B_{\pi(n)}$ sequentially based on some permutation π (and writes data into them after the erasures), and finally erases all the auxiliary blocks. Let π^{-1} denote the inverse function of π. What kind of data needs to be written into the auxiliary blocks? For the data originally in the page $p_{i,j}$ (where $i \in [n]$ and $j \in [m]$), namely $d_{i,j}$, we have

- If $\pi^{-1}(\alpha(i,j)) > \pi^{-1}(i)$, then $d_{i,j}$ needs to be written into the auxiliary blocks, because otherwise, when B_i is erased, the data $d_{i,j}$ would be lost. (Note that $d_{i,j}$ needs to be moved into $B_{\alpha(i,j)}$, and in the solution, B_i is erased before $B_{\alpha(i,j)}$.)
- If $\pi^{-1}(\alpha(i,j)) = \pi^{-1}(i)$ (i.e., $\alpha(i,j) = i$), then $d_{i,j}$ needs to be written into the auxiliary blocks, because otherwise, when B_i is erased, the data would be lost.
- If $\pi^{-1}(\alpha(i,j)) < \pi^{-1}(i)$, then $d_{i,j}$ does not need to be written into the auxiliary blocks, because when B_i is erased, it has already been moved into $B_{\alpha(i,j)}$.
- If $\pi^{-1}(\alpha(i,j)) = \phi$, then $d_{i,j}$ does not need to be written into the auxiliary blocks, because the data just need to be erased.

In the above four cases, there are $|A_\pi|$ pages of data in the first case because every such page of data corresponds to an *ascending edge* in the *movement graph* whose permuted labelling is π. There are $\sum_{i\in[n]} d(i \to i)$ pages of data in the second case. So the canonical solution needs at least $\lceil \frac{\min_{\pi\in\Pi} |A_\pi| + \sum_{i\in[n]} d(i\to i)}{m} \rceil$ auxiliary blocks to store the data in the first and second cases. So the solution needs at least $n + \lceil \frac{\min_{\pi\in\Pi} |A_\pi| + \sum_{i\in[n]} d(i\to i)}{m} \rceil$ erasures. So the second step of the proof is also shown to be correct. On the other side, when $\Delta \geqslant \lceil \frac{\min_{\pi\in\Pi} |A_\pi| + \sum_{i\in[n]} d(i\to i)}{m} \rceil$, the solution described above is also feasible, so the lower bound becomes tight. ■

Theorem 19. *For the generalized data-movement problem, it is NP hard to find per-instance optimal solutions without coding.*

Proof: It is sufficient to prove that the problem is NP hard when $\Delta \geqslant nm$. When $\Delta \geqslant nm$, an optimal solution can be transformed into a canonical optimal solution in polynomial time. Also, if a canonical optimal solution is known, a *permuted labelling* (i.e., permutation) π for the movement graph that minimizes the value $\lceil \frac{|A_\pi| + \sum_{i\in[n]} d(i\to i)}{m} \rceil$ can be found in polynomial time, and vice versa. (Both are shown in the proof of Theorem 18.) Let $F = ((\sum_{i\in[n]} d(i \to i)) \bmod m)$. We see that $0 \leqslant F \leqslant m-1$ and F is fixed given the instance of the problem. The above observations show that it is sufficient to prove that it is NP hard to find a permutation π that minimizes $\lceil \frac{|A_\pi| + F}{m} \rceil$.

Let π_0 be a permutation such that $|A_{\pi_0}| = \min_{\pi\in\Pi} |A_\pi|$. Let π^* be a permutation such that $|A_{\pi^*}| = \lceil \frac{\min_{\pi\in\Pi} |A_\pi| + F}{m} \rceil$. (So $|A_{\pi^*}| \geqslant |A_{\pi_0}|$, but $\lceil \frac{|A^*_\pi| + F}{m} \rceil = \lceil \frac{|A_{\pi_0}| + F}{m} \rceil$.) We will prove two claims:

- CLAIM ONE: It is APX-hard to find π_0.
- CLAIM TWO: It is NP hard to find π^*.

First, we prove CLAIM ONE using a reduction from the APX-hard *minimum feedback arc set problem*. Given a permutation π, we can easily find the *ascending edges* A_π. Since every cycle in the movement graph $G_m = (V, E)$ must contain an ascending edge, A_π is also a *feedback arc set* of G. (A feedback arc set of a direct graph is a subset of edges such that every

cycle contains at least one edge from the subset.) On the other side, given a feedback arc set $S \subseteq E$ of G, we can easily find a topological ordering of the vertices of the directed acyclic graph $G_S = (V, E - S)$ such that all the edges of G_S are descending edges. (Note that a topological ordering is also a permutation of the vertices.) So if we use π_S to denote this topological ordering, then $A_{\pi_S} \subseteq S$. So it is not difficult to see that a minimum feedback arc set is also a minimum set of ascending edges, and vice versa. (Also note that every directed graph can be the movement graph of a data movement problem.) So finding π_0 is equivalent to the minimum feedback arc set problem, which is APX hard. So CLAIM ONE is true.

We now prove CLAIM TWO by contradiction. Suppose that there is a polynomial-time algorithm to find π^*. We have shown that finding π_0 is APX-hard, which means that there exists a constant $c > 1$ such that no polynomial-time algorithm can guarantee to find a permutation π with the property $|A_\pi| < c|A_{\pi_0}|$, assuming $P \neq NP$. Let $G_z = (V_z, E_z)$ be a graph that consists of z copies of the data movement graph $G = (V, E)$, where $z = \lceil \frac{m}{c-1} \rceil$. Clearly, G_z is also the data movement graph of a data movement problem, and its optimal data movement solution uses $zn + \lceil \frac{z(|A_{\pi_0}| + \sum_{i\in[n]} d(i\to i))}{m} \rceil$ erasures. (W.l.o.g., we can assume that $|A_{\pi_0}| \geqslant 1$.) We can use the polynomial-time algorithm to find an optimal solution for G_z that applies the same permutation π' to the z subgraphs of G_z, which uses $zn + \lceil \frac{z(|A_{\pi'}| + \sum_{i\in[n]} d(i\to i))}{m} \rceil$ erasures. So $zn + \lceil \frac{z(|A_{\pi_0}| + \sum_{i\in[n]} d(i\to i))}{m} \rceil = zn + \lceil \frac{z(|A_{\pi'}| + \sum_{i\in[n]} d(i\to i))}{m} \rceil$. So $\lceil \frac{z(|A_{\pi_0}| + F)}{m} \rceil = \lceil \frac{z(|A_{\pi'}| + F)}{m} \rceil$. So $z(|A_{\pi'}| + F) \leqslant z(|A_{\pi_0}| + F) + (m-1)$. So $|A_{\pi'}| \leqslant |A_{\pi_0}| + \frac{m-1}{z} = |A_{\pi_0}| + \frac{m-1}{\lceil m/(c-1) \rceil} < |A_{\pi_0}| + (c-1) \leqslant |A_{\pi_0}|(1 + \frac{c-1}{|A_{\pi_0}|}) \leqslant |A_{\pi_0}|(1 + c - 1) = c|A_{\pi_0}|$. That contradicts the fact that no polynomial-time algorithm can guarantee to find a permutation π with the property $|A_\pi| < c|A_{\pi_0}|$, assuming $P \neq NP$. So it is NP hard to find π^*. So CLAIM TWO is true, and the theorem is proved. ■

Let us now study approximation algorithms for the data movement problem, for the case where $\Delta \geqslant nm$. In this case, canonical solutions exist; and to minimize the number of erasures, we just need to find a permuted labelling π that minimizes the number of ascending edges. As shown in the proof of Theorem 19, this is the same as finding the minimum feedback arc set. There are known approximation algorithms for the latter problem [3]. We now present the corresponding approximation algorithm for the data movement problem.

Algorithm 20. (*Canonical data-movement algorithm without coding)*

Let $G_m = (V, E)$ be the movement graph. Let F be a known approximation algorithm for the minimum feedback arc set problem.

1) *Use the algorithm F to find a feedback arc set S of the movement graph G_m.*
2) *Find a topological ordering π of the acyclic directed graph $G_S = (V, E - S)$, such that all the edges of G_S are descending edges. (Note that a topological ordering is also a permutation of the vertices.)*
3) *Let π be the permuted ordering for G_m. Let $S' \subseteq S$ be the set of ascending edges in S.*[2] *Let $z = \lceil \frac{|S'| + \sum_{i\in[n]} d(i\to i)}{m} \rceil$.*
4) *Write into the auxiliary blocks $B_{n+1}, B_{n+2}, \cdots, B_{n+z}$ the data that either correspond to the "ascending edges" or are required by this problem to stay in their original blocks.*
5) *For $i = \pi(1), \pi(2), \cdots, \pi(n)$, erase block B_i, then write into it the data that are required to be moved into it by this problem.*
6) *Erase the auxiliary blocks $B_{n+1}, B_{n+2}, \cdots, B_{n+z}$.*

The following result analyzes the approximation ratio of Algorithm 24.

Theorem 21. *Let c denote the approximation ratio of the known algorithm F for the size of the feedback arc set. Then the approximation ratio of Algorithm 24 for the number of erasures is at most*

$$\frac{3c}{2c+1} + \frac{1}{n} \approx \frac{3c}{2c+1} < \min\{\frac{3}{2}, c\}.$$

Proof: Let S^* be the minimum feedback arc set of the movement graph G_m. Then the algorithm F finds a feedback arc set S of size $|S| \leqslant c|S^*|$. The optimal solution without coding to the data movement problem uses $n + \lceil \frac{|S^*| + \sum_{i\in[n]} d(i\to i)}{m} \rceil$ erasures. Algorithm 24 uses $n + \lceil \frac{|S| + \sum_{i\in[n]} d(i\to i)}{m} \rceil$ erasures. We have $|S| \leqslant \frac{nm - \sum_{i\in[n]} d(i\to i)}{2}$ because it is trivial to obtain a

[2] In fact, since $S' \subseteq S$ is also a feedback arc set, a carefully designed algorithm F will have $S' = S$.

feedback arc set in G that consists of half of its edges. So the approximation ratio of Algorithm 24 is

$$\begin{aligned}
& \frac{n+\lceil \frac{|S|+\sum_{i\in[n]} d(i\to i)}{m} \rceil}{n+\lceil \frac{|S^*|+\sum_{i\in[n]} d(i\to i)}{m} \rceil} \\
< \quad & \frac{n+\frac{|S|+\sum_{i\in[n]} d(i\to i)}{m}+1}{n+\frac{|S^*|+\sum_{i\in[n]} d(i\to i)}{m}} \\
\leqslant \quad & \frac{n+\frac{|S|+\sum_{i\in[n]} d(i\to i)}{m}}{n+\frac{|S^*|+\sum_{i\in[n]} d(i\to i)}{m}} + \frac{1}{n} \\
\leqslant \quad & \frac{n+\frac{nm/2}{m}}{n+\frac{nm/2c}{m}} + \frac{1}{n} \\
= \quad & \frac{n+\frac{n}{2}}{n+\frac{n}{2c}} + \frac{1}{n} \\
= \quad & \frac{3c}{2c+1} + \frac{1}{n}
\end{aligned}$$

■

VII. Conclusions

In this paper, we present both coding and non-coding based algorithms for efficient data movement in flash memories. By proving a lower bound for the number of erasures used by algorithms without coding, we rigorously show the advantage of coding. The hardness and the approximation of per-instance optimization are also studied.

Appendix

We describe here an algorithm for the non-coding data movement problem that is efficient and has a straightforward implementation. The model we use is slightly different from the one in previous sections, but only for the sake of a clear description. More specifically, we model the data by an $m \times n$ matrix. Each entry is labeled with a number from 0 to $n-1$, specifying the destination block. For each label i, there are exactly m entries labeled with i. In the construction below we use $n = m = 2^p$ and $\Delta = 2$. It is not hard to verify that the construction extends to genearal Δ and arbitrary $m \geqslant n$ (by changing the underlying alphabet used in the construction from 2 to Δ symbols).

A. Decomposition by Hall's Theorem

We can apply the well known Hall's theorem from combinatorics to decompose the matrix into n sets $S_0, \ldots, S_{n-1}$, such that each set contains all the numbers from 0 to $n-1$ exactly once, and also each set contains exactly one number from each column. For completeness, we include Hall's theorem below [2]:

Theorem 22. *[Hall] There exists a system of distinct representatives for a family of n sets iff the union of any k of these sets contains at least k elements for all k from 1 to n.*

In our case a set is a column, and the family of sets is the entire matrix. It is easy to verify that the union of any k columns, for all k from 1 to n contains at least k different numbers. Therefore the theorem applies, and the decomposition can be found in $O(n^3 \log n)$, or $O(mn^2 \log n)$ in general [1], while this does not involve any erasure.

B. Data Movement by Twice Transpose

The data movement problem can be solved by two applications of an algorithm that realizes the transpose of the matrix. First, we decompose the matrix by Hall's theorem, to obtain the sets $S_0, \ldots, S_{n-1}$ as described before. With n erasures, by changing the position of pages within each column, each set S_i can be moved to occupy the i-th row of the matrix. After the first transpose, each set S_i will ocuppy the i-th column. With n more erasures, again rearrangements within each column, every set S_i can be ordered from 0 to $n-1$. Finally, the second transpose gives the desired configuration. We show an example for a 4×4 matrix.

Example 23. *The Hall decomposition is shown by using card suits, indexed by the original number of the entry. Notice that each suit has all the indexes from 0 to 3. Then we rearrange each column such that at the end every suit appears in one line. After the first transpose, every suit appears in one column. After another vertical rearrangement of each column followed by a transpose, we reach the final state where all indexes i belong to column i, for $i \in \{0, \ldots, n-1\}$.*

$$\begin{pmatrix} 3 & 2 & 1 & 0 \\ 1 & 3 & 3 & 2 \\ 2 & 0 & 0 & 1 \\ 0 & 2 & 1 & 3 \end{pmatrix} \to \overbrace{\begin{pmatrix} \spadesuit_3 & \clubsuit_2 & \spadesuit_1 & \diamondsuit_0 \\ \diamondsuit_1 & \heartsuit_3 & \diamondsuit_3 & \spadesuit_2 \\ \heartsuit_2 & \spadesuit_0 & \heartsuit_0 & \heartsuit_1 \\ \clubsuit_0 & \diamondsuit_2 & \clubsuit_1 & \clubsuit_3 \end{pmatrix}}^{\textit{Hall decomposition}} \to \overbrace{\begin{pmatrix} \spadesuit_3 & \spadesuit_0 & \spadesuit_1 & \spadesuit_2 \\ \heartsuit_2 & \heartsuit_3 & \heartsuit_0 & \heartsuit_1 \\ \diamondsuit_1 & \diamondsuit_2 & \diamondsuit_3 & \diamondsuit_0 \\ \clubsuit_0 & \clubsuit_2 & \clubsuit_1 & \clubsuit_3 \end{pmatrix} \to \begin{pmatrix} \spadesuit_3 & \heartsuit_2 & \diamondsuit_1 & \clubsuit_0 \\ \spadesuit_0 & \heartsuit_3 & \diamondsuit_2 & \clubsuit_2 \\ \spadesuit_1 & \heartsuit_0 & \diamondsuit_3 & \clubsuit_1 \\ \spadesuit_2 & \heartsuit_1 & \diamondsuit_0 & \clubsuit_3 \end{pmatrix}}^{\textit{First transpose}} \to \overbrace{\begin{pmatrix} \spadesuit_0 & \heartsuit_0 & \diamondsuit_0 & \clubsuit_0 \\ \spadesuit_1 & \heartsuit_1 & \diamondsuit_1 & \clubsuit_1 \\ \spadesuit_2 & \heartsuit_2 & \diamondsuit_2 & \clubsuit_2 \\ \spadesuit_3 & \heartsuit_3 & \diamondsuit_3 & \clubsuit_3 \end{pmatrix} \to \begin{pmatrix} \spadesuit_0 & \spadesuit_1 & \spadesuit_2 & \spadesuit_3 \\ \heartsuit_0 & \heartsuit_1 & \heartsuit_2 & \heartsuit_3 \\ \diamondsuit_0 & \diamondsuit_1 & \diamondsuit_2 & \diamondsuit_3 \\ \clubsuit_0 & \clubsuit_1 & \clubsuit_2 & \clubsuit_3 \end{pmatrix}}^{\textit{Second transpose}}$$

The basic step of the algorithm is an exchange of entries between two columns. Therefore, the vertical position of pages in a column is not important at the beginning of the algorithm, and we can exclude the n erasures that were mentioned before each transpose.

C. Bit-Fixing Algorithm for Transpose

We now describe the algorithm that realizes the transpose of a square matrix with $n = 2^p$ columns. It is shown as Algorithm 24 below. The guiding principle of the algorithm is to use the binary representation of the column indexes (and of the matrix entries), and move the data between columns such that corresponding bits of the entries are in agreement with those of the columns, therefore by "fixing the bits". The outer loop has p steps, corresponding to each of the p bits in the binary representation. They can be fixed in any order, but for the sake of a definition we chose the one from least to most significant. For each bit that is fixed, we make $n/2 = 2^{p-1}$ pairs of columns. The condition is that for each pair, the binary representation of their indexes has to agree on the bits that have already been fixed. Again, for the simplicity of defining the algorithm, we choose the pairs such that the binary representations of their indexes agree on all the bits, except the one being fixed in the current round. A pair is defined by the columns B_0 and B_1 in the algorithm. For any such pair, we rearrange their entries such that they agree with the column index on the bit i that is currently being fixed. The correctness of the algorithm is shown by the following theorem.

Algorithm 24. BIT-FIXING ALGORITHM FOR TRANSPOSE

INPUT: *Square matrix* $\mathcal{A} = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \cdots & \vdots \\ n-1 & n-1 & \cdots & n-1 \end{pmatrix}$, *where n is a power of 2, $n = 2^p$.*

OUTPUT: *The transpose of $\mathcal{A}$.*

ALGORITHM: *For $i = 0$ to $p-1$, and for $k = 0$ to $2^{p-1}-1$, do:*

1) *Let $(b_{p-2} \dots b_1 b_0)$ be the binary representation of k.*
2) *Let B_0 be the column with binary index $(b_{p-2} \dots b_i \, \mathbf{0} \, b_{i-1} \dots b_1 b_0)$.*
3) *Let B_1 be the column with binary index $(b_{p-2} \dots b_i \, \mathbf{1} \, b_{i-1} \dots b_1 b_0)$.*
4) *Between B_0 and B_1, move entries whose i-th bit is 0 to B_0 and those whose i-th bit is 1 to B_1.*

Theorem 25. *The* BIT-FIXING TRANSPOSE *Algorithm is correct, namely it realizes the transpose of the input matrix. The number of block erasures is $n \log n$, using only two extra blocks of memory.*

Proof: We give a proof by induction over the number of bits that are fixed. We want to prove the following property: "after fixing k bits, $k \in \{1, \dots, (\log n) - 1\}$, each column contains exactly 2^{p-k} numbers, each one of them appearing 2^k times."

Basis case: it is easy to verify that the property is true for $k = 1$. After fixing the first bit, any odd column will contain all the odd numbers, each appearing twice, and the same is true for even columns and numbers.

Suppose the property is true for any number of bits up to k, $k \in \{1, \dots (\log n) - 1\}$. We prove that after fixing the $(k+1)$-th bit the property is still preserved. Consider an assignment $(b_{k-1} \dots b_0)$ of the first k bits. There are 2^{p-k} numbers whose binary representation agrees with the assignment $(b_{k-1} \dots b_0)$, and from the induction hypothesis it follows that any column whose index agrees with $(b_{k-1} \dots b_0)$ must contain all these 2^{p-k} numbers. Therefore the set of columns that agree with $(b_{k-1} \dots b_0)$ has cardinality 2^{p-k}, and any such column contains all the possible numbers than can agree with $(b_{k-1} \dots b_0)$, for a total of 2^{p-k} numbers, each appearing 2^k times in any of the columns. The algorithm exchanges data between columns that agree on the first k bits, and do not agree on the $(k+1)$-th. Therefore, for each pair, $n/2$ of the numbers in a column agree with the $(k+1)$-th bit of the column index, and $n/2$ do not. Therefore, after fixing the $(k+1)$-th bit the property is still preserved.

After fixing the last bit, every column will contain just one number, that is equal to the column index, thereby realizing the transpose.

To count the number of erasures, there are $\log n$ steps in the outer loop. Each inner loop can be done with $2n$ erasures, assuming two extra columns because each pair of columns can exchange their data with 4 erasures by simply copying them to the extra columns and back in the desired order. In fact, by rotating the extra space, namely by copying the pair to the extra space in the order dictated by fixing the bit and using the space of the pair as the new empty space, the inner loop can be realized by n erasures. Therefore, the algorithm makes $n \log n$ when rotating the empty space. ■

Example 26. *We show the execution of the* BIT-FIXING TRANSPOSE *Algorithm for $n = 8$. The top row is the binary column index, and the bottom row indicates the pairs of columns p_i, $i \in \{1, \dots, n/2\}$, used to move the data.*

000	001	010	011	100	101	110	111
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7
p_1	p_1	p_2	p_2	p_3	p_3	p_4	p_4

Pairs used to fix b_0

$\xrightarrow{b_0}$

000	001	010	011	100	101	110	111
0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1
2	3	2	3	2	3	2	3
2	3	2	3	2	3	2	3
4	5	4	5	4	5	4	5
4	5	4	5	4	5	4	5
6	7	6	7	6	7	6	7
6	7	6	7	6	7	6	7
p_1	p_2	p_1	p_2	p_3	p_4	p_3	p_4

Pairs used to fix b_1

$\xrightarrow{b_1}$

000	001	010	011	100	101	110	111
0	1	2	3	0	1	2	3
0	1	2	3	0	1	2	3
0	1	2	3	0	1	2	3
0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
4	5	6	7	4	5	6	7
4	5	6	7	4	5	6	7
4	5	6	7	4	5	6	7
p_1	p_2	p_3	p_4	p_1	p_2	p_3	p_4

Pairs used to fix b_2

$\xrightarrow{b_2}$

000	001	010	011	100	101	110	111
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

REFERENCES

[1] N. Alon, "A simple algorithm for edge-coloring bipartite multigraphs," in *Inf. Process. Lett.*, vol. 85, no. 6, pp. 301–302, 2003.

[2] B. Bollobás, *Random Graphs (2nd Edition)*. Cambridge University Press, 2001.

[3] G. Even, J. Naor, B. Schieber and M. Sudan, "Approximating minimum feedback sets and multi-cuts in directed graphs," in *Proc. 4th Int. Conf. on Integer Prog. and Combinatorial Optimization*, Lecture Notes in Comput. Sci. 920, Springer-Verlag, pp. 14-28, 1995.

[4] H. Finucane, Z. Liu, and M. Mitzenmacher, "Designing floating codes for expected performance," in *Proc. of the Annual Allerton Conference*, 2008.

[5] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," in *ACM Computing Surveys*, vol. 37, no. 2, pp. 138-163, June 2005.

[6] A. Jiang, V. Bohossian, and J. Bruck, "Floating codes for joint information storage in write asymmetric memories," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, Nice, France, Jun. 2007, pp. 1166–1170.

[7] A. Jiang, M. Langberg, M. Schwartz and J. Bruck, "Universal rewriting in constrained memories," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, Seoul, Korea, 2009, pp. 1219-1223.

[8] A. Jiang, H. Li and Y. Wang, "Error scrubbing codes for flash memories," in *Proc. Canadian Workshop on Information Theory (CWIT)*, Ottawa, Canada, May 2009, pp. 32-35.

[9] A. Jiang, R. Mateescu, M. Schwartz and J. Bruck, "Rank modulation for flash memories," in *IEEE Transactions on Information Theory*, vol. 55, no. 6, pp. 2659-2673, June 2009.

[10] A. Jiang, R. Mateescu, E. Yaakobi, J. Bruck, P. Siegel, A. Vardy and J. Wolf, "Storage coding for wear leveling in flash memories," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, Seoul, Korea, 2009, pp. 1229-1233.

[11] A. Jiang, M. Schwartz and J. Bruck, "Error-correcting codes for rank modulation," *Proc. IEEE International Symposium on Information Theory (ISIT)*, Toronto, Canada, July 2008, pp. 1736-1740.

[12] E. Yaakobi, A. Vardy, P. H. Siegel, and J. K. Wolf, "Multidimensional flash codes," in *Proc. of the Annual Allerton Conference*, 2008.