Data Movement and Aggregation in Flash Memories

Anxiao (Andrew) Jiang
CSE 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
Microsoft Research Cambridge
7 J J Thomson Ave.
Cambridge CB3 0FB, UK
romatees@microsoft.com

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

***Abstract*—NAND flash memories have become the most widely used type of non-volatile memories. In a NAND flash memory, every block of memory cells consists of numerous pages, and rewriting a single page requires the whole block to be erased. As block erasures significantly reduce the longevity, speed and power efficiency of flash memories, it is critical to minimize the number of erasures when data are reorganized. This leads to the *data movement problem*, where data need to be switched in blocks, and the objective is to minimize the number of block erasures. It has been shown that optimal solutions can be obtained by coding. However, coding-based algorithms with the minimum coding complexity still remain an important topic to study.**

In this paper, we present a very efficient data movement algorithm with coding over $GF(2)$ and with the minimum storage requirement. We also study data movement with more auxiliary blocks and present its corresponding solution. Furthermore, we extend the study to the *data aggregation problem*, where data can not only be moved but also aggregated. We present both non-coding and coding-based solutions, and rigorously prove the performance gain by using coding.

I. Introduction

NAND flash memories have become by far the most widely used type of non-volatile memories (NVMs). In a NAND flash memory, floating-gate memory cells are organized as *blocks*. Every block is further partitioned into *pages*. The page is the basic unit for read and write operations [1]. A typical page stores 2KB to 4KB of data, and a typical block consists of 64 or 128 pages [2]. Flash memories have a prominent *block erasure* property: once data are written into a page, to modify the data, the whole block has to be erased and then reprogrammed. A block can endure only $10^4 \sim 10^5$ erasures before it may break down, so the longevity of flash memories is measured by the block erasures [1]. Block erasures also significantly reduce the writing speed and the power efficiency of flash memories. Therefore, it is critical to minimize the number of erasures when data are reorganized [2]. This leads to the *data movement problem*, which has been studied in [3], [4]. Although data movement is very common in all storage systems, the unique block erasure property of flash memories calls for very special solutions.

In the data movement problem [3], [4], there are n blocks storing data, where every block has m pages. The nm pages of data need to be switched among the n blocks with specified destinations. There are δ empty blocks called *auxiliary blocks* that can help store intermediate results during the data movement process. The objective is to minimize the number of block erasures. It was proved in [4] that optimal solutions can be obtained by using coding. Furthermore, a coding-based algorithm using at most $2n-1$ erasures for $\delta = 1$ was presented [4], which is worst-case optimal. The algorithm in [4] requires coding over a large Galois field; to reduce the coding complexity, it was shown in [3] later that there exist solutions with coding over $GF(q)$ for $q \geq 3$. However, it remained an open question whether there exist optimal solutions that use coding over $GF(2)$. The answer is important for obtaining optimal data movement algorithms with the minimum coding complexity.

In this paper, we prove that the answer is positive by presenting a very efficient optimal algorithm over $GF(2)$ when $\delta = 1$ (i.e., minimum number of auxiliary blocks). When $\delta \geq 2$, we present a coding-based algorithm that uses at most $2n - \min\{\delta, \lfloor n/2 \rfloor\}$ erasures. Although it is NP hard to minimize the number of erasures for every instance (i.e., per-instance optimization), the above algorithms can achieve constant approximation ratios.

We further extend the study to the *data aggregation problem*, where data can not only be moved, but also aggregated. Specifically, data of similar attributes are required to be placed together, although the destination may not be specified; in other cases, the final data can be functions of the original data. Data aggregation has many applications in flash memories. For example, for *wear leveling* (i.e., balancing erasures across blocks), it is beneficial to store frequently modified data (i.e. *hot data*) together and store cold data together [2]. In flash-based databases, the temporarily stored raw data need to be organized as structured data [5]. The external memories of sensors often use flash memories [6], where aggregation is important for analyzing the collected data.

We present both non-coding and coding-based algorithms for data aggregation. We present a lower bound for the number of erasures needed by non-coding solutions, which is very close to the upper bound obtained from our algorithm. The lower bound also rigorously proves the performance gain by using coding because the coding-based algorithms use only a linear number of erasures, which is asymptotically optimal.

The rest of the paper is organized as follows. In Section II, we present the optimal data movement algorithm with coding over $GF(2)$ and with one auxiliary block. In Section III, we study the data movement problem with multiple auxiliary blocks. In Section IV, we define and study the data aggregation problem. In Section V, we present the concluding remarks.

II. Optimal Data Movement over $GF(2)$

In this section, we present an optimal data movement algorithm with coding over $GF(2)$, which has very low coding complexity. First, let us define the data movement problem [4].

Definition 1. Data Movement Problem

Consider n blocks storing data in an NAND flash memory, where every block has m pages. They are denoted by $B_1, \ldots, B_n$, and the m pages in block B_i are denoted by $p_{i,1}, \ldots, p_{i,m}$, for $i = 1, \ldots, n$. Let $\alpha(i,j)$ and $\beta(i,j)$ be two functions:

$$\alpha(i,j) : \{1,\ldots,n\} \times \{1,\ldots,m\} \rightarrow \{1,\ldots,n\};$$

$$\beta(i,j) : \{1,\ldots,n\} \times \{1,\ldots,m\} \rightarrow \{1,\ldots,m\}.$$

The functions $\alpha(i,j)$ and $\beta(i,j)$ specify the desired data movement. Specifically, the data initially stored in the page $p_{i,j}$ are denoted by $D_{i,j}$, and need to be moved into page $p_{\alpha(i,j),\beta(i,j)}$, for all $(i,j) \in \{1,\ldots,n\} \times \{1,\ldots,m\}$.

There are δ empty blocks, called auxiliary blocks, that can be used in the data movement process, and they need to be erased in the end. To ensure data integrity, at any moment of the data movement process, the data stored in the flash memory blocks should be sufficient for recovering all the original data. The objective is to minimize the total number of block erasures in the data movement process.

We assume that each of $B_1, \ldots, B_n$ has at least one page of data that need to be moved to another block, because otherwise it can be excluded from the problem. Since a block has to be erased whenever any of its pages is to be modified, the data movement needs no less than n erasures.

n pages of data $\{D_{1,j_1}, D_{2,j_2}, \ldots, D_{n,j_n}\}$ are called a *block-permutation data set* if

$$\{\alpha(1,j_1), \alpha(2,j_2), \ldots, \alpha(n,j_n)\} = \{1,2,\ldots,n\}.$$

Clearly, the data in a block-permutation data set belong to n different blocks (i.e., $B_1, \ldots, B_n$) both before and after the data movement process. It is proved in [4] that the nm pages of data in $B_1, \ldots, B_n$ can be partitioned exactly into m block-permutation data sets.

Example 2. *Let $n = 21$ and $m = 3$. Let the nm values of $\alpha(i,j)$ be shown as the $m \times n$ matrix in Fig. 1. (For example, $\alpha(1,1) = 6, \alpha(1,2) = 15, \alpha(1,3) = 7$.) We can partition the $nm = 63$ pages of data into $m = 3$ block-permutation data sets, denoted by $\heartsuit$, $\spadesuit$ and $\diamondsuit$ in Fig. 1. In the figure, $\alpha(i,j)^{\heartsuit}$ (or $\alpha(i,j)^{\spadesuit}$, $\alpha(i,j)^{\diamondsuit}$, respectively) means that the data $D_{i,j}$ belong to the block permutation data set $\heartsuit$ (or $\spadesuit$, $\diamondsuit$, respectively).*

In the remaining of this section, we consider $\delta = 1$. Let y be the smallest integer in $\{1,2,\ldots,n-2\}$ with this property: *for any $i \in \{y+3, y+4, \ldots, n\}$ and $j \in \{1,\ldots,m\}$, either $\alpha(i,j) \leq y$ or $\alpha(i,j) \geq i-1$.* Our algorithm will use $n+y+1 \leq 2n-1$ erasures. This is *worst-case optimal*, because it is known that there exist instances where $2n-1$ erasures are necessary [4]. Note that we can label the n blocks storing data as $B_1, \ldots, B_n$ in $n!$ different ways and get different values of y. If we focus on per-instance optimization (i.e., optimization for every given instance), then it is know that there is a solution with $n+z+1$ erasures if and only if we can label the n blocks as $B_1, \ldots, B_n$ such that $y \leq z$ [4]. Therefore, our algorithm can also be readily utilized in per-instance optimal solutions. However, it is known NP hard to label $B_1, \ldots, B_n$ such that y is minimized [4].

The algorithm to be presented will work the same way for the m block-permutation data sets in parallel. Specifically, for every block and at any moment, the block's m pages are used by the m block-permutation data sets (one for each). So for convenience of presentation, in the following, we consider only one of the m block-permutation data sets. So let B_0 denote the auxiliary block, and for $i = 0,1,\ldots,n$, assume B_i has only one page. (Again, note that the block-permutation data set in consideration uses only one page in B_i.) For $i = 1,\ldots,n$, let D_i denote the data originally stored in B_i. For $i = 1,\ldots,n$, we use $\alpha(i) \in \{1,\ldots,n\}$ to mean that the data D_i need to be moved to block $B_{\alpha(i)}$. Let α^{-1} be the inverse function of α. (That is, $\forall\ i \in \{1,\ldots,n\}$, $\alpha(\alpha^{-1}(i)) = i$.)

We now introduce the data movement algorithm. In our algorithm, every block is erased at most twice. More specifically, the algorithm has three stages:

1) *Stage one*: For $i = 1,2,\ldots,y+1$, we write some coded data (which are the XOR of the original data) into B_{i-1}, then erase B_i.
2) *Stage two*: For $i = y+2, y+3, \ldots, n$, we write $D_{\alpha^{-1}(i-1)}$ into B_{i-1}, then erase B_i. Then, we write $D_{\alpha^{-1}(n)}$ into B_n and erase B_y.
3) *Stage three*: For $i = y-1, y-2, \ldots, 0$, we write $D_{\alpha^{-1}(i+1)}$ into B_{i+1}, then erase B_i.

We still need to specify what the *coded data* are in the algorithm, and prove that at all times, the data stored in the flash memory are sufficient for recovering all the original data. The data stored during the data movement process can be represented by a forest. For example, for the problem in Example 2, if we consider the block-permutation data set labelled by $\heartsuit$, then the forest is as shown in Fig. 2. Here every vertex represents a page of original data, and every edge (or hyper-edge) represents the XOR of its endpoint vertices. The forest shows all the data used by the algorithm. When the algorithm runs, there are always n linearly independent data symbols stored in the flash memory, which enables the recovery of all the original data $D_1, \ldots, D_n$. The forest structure will make it very efficient to analyze the linear independency.

Let us show how the forest is obtained. For $i = 1,2,\ldots,y+1$, we define $\tilde{S}_i \subseteq \{i, i+1, \ldots, n\}$ as a set that is recursively constructed as follows:

1) $i \in \tilde{S}_i$;
2) For any $j \in \tilde{S}_i$, if

$$\max\{j, y+1\} \leq \alpha(j) < n,$$

Fig. 1. The matrix of $\alpha(i,j)$ for $i = 1, \ldots, n$, $j = 1, \ldots, m$, and the partition of data into m block-permutation data sets.

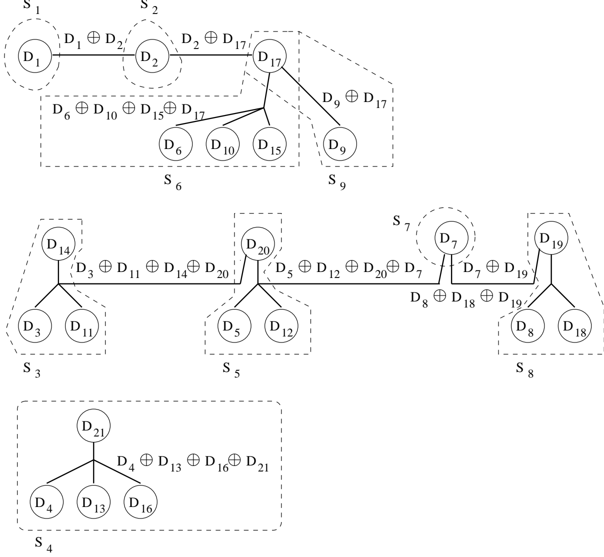

Fig. 2. Data movement with coding over $GF(2)$.

then $\alpha(j) + 1 \in \tilde{S}_i$.

Lemma 3. *The $y+1$ sets $\tilde{S}_1, \tilde{S}_2, \ldots, \tilde{S}_{y+1}$ form a partition of the set*

$$\{1, 2, \ldots, n\} \setminus \{i | y+2 \leq i \leq n, \alpha(i) = i-1\}.$$

Proof: We need to prove that

1) $\tilde{S}_i \cap \tilde{S}_j = \emptyset$ for any $i \neq j$;
2) For any $i \in \{1, \ldots, n\}$, $i \notin \cup_{i=1}^{y+1} \tilde{S}_i$ if and only if $i \geq y+2$ and $\alpha(i) = i-1$.

For $i \in \{1, \ldots, y+1\}$, the integers in $\tilde{S}_i$ form a prefix of the sequence: $\{i,\ \alpha(i)+1,\ \alpha(\alpha(i)+1)+1,\ \alpha(\alpha(\alpha(i)+1)+1)+1,\ \ldots\}$. The set $\tilde{S}_i$ is the longest prefix that satisfies two conditions: (1) it monotonically increases; (2) the second number (if it exists) is at least $y+2$. Since α is a bijection, we can see that $\tilde{S}_i \cap \tilde{S}_j = \emptyset$ when $i \neq j$.

For $i = 1, \ldots, y+1$, since $i \in \tilde{S}_i$, we have $i \in \cup_{j=1}^{y+1} \tilde{S}_j$. So if $i \notin \cup_{j=1}^{y+1} \tilde{S}_j$, the $i \geq y+2$. Let $i \in \{y+2, y+3, \ldots, n\}$. By the definition of the parameter y, either $\alpha(i) \leq y$, or $\alpha(i) = i-1$, or $\alpha(i) \geq i$. When $\alpha(i) \leq y$ or $\alpha(i) \geq i$, by the definition of $\tilde{S}_1, \tilde{S}_2, \ldots, \tilde{S}_{y+1}$, we can see that i belongs to the $\tilde{S}_x$ (for some $x \in \{1, \ldots, y+1\}$) that $\alpha^{-1}(i-1)$ also belongs to. When $\alpha(i) = i-1$, i cannot belong to any $\tilde{S}_x$. So we get the conclusion. ■

Let $\eta \in \{1, 2, \ldots, y+1\}$ be the unique integer such that $\alpha^{-1}(n) \in \tilde{S}_\eta$. For any set of numbers C, let $\max(C)$ denote the greatest number in C. We have the following observation.

Lemma 4. *Throughout the stage one and stage two of the algorithm, for any $i \in \{1, \ldots, y+1\}$, among the $|\tilde{S}_i|$ pages of data in $\{D_j | j \in \tilde{S}_i\}$, at least $|\tilde{S}_i| - 1$ pages of data are stored in their original form in the flash memory.*

When the stage two of the algorithm ends, all the $|\tilde{S}_\eta|$ pages of data in $\{D_j | j \in \tilde{S}_\eta\}$ are stored in their original form. And for any $i \in \{1, \ldots, y+1\} \setminus \{\eta\}$, the only page of data in $\{D_j | j \in \tilde{S}_i\}$ that may not be stored in its original form is

$$D_{\max(\tilde{S}_i)}.$$

Proof: For any $i \in \{1, \ldots, y+1\}$, the integers in $\tilde{S}_i$ are of the form:

$$i,\ \alpha(i)+1,\ \alpha(\alpha(i)+1)+1,\ \ldots$$

In the stage one and stage two of the algorithm, after D_i is written into $B_{\alpha(i)}$, $D_{\alpha(i)+1}$ is erased from $B_{\alpha(i)+1}$; then $D_{\alpha(i)+1}$ is written into $B_{\alpha(\alpha(i)+1)}$, and $D_{\alpha(\alpha(i)+1)+1}$ is erased from $B_{\alpha(\alpha(i)+1)+1}$; and so on. So the conclusions hold. ■

We define $S_1, S_2, \ldots, S_{y+1}$ as follows. If $\eta = y+1$, then $S_i = \tilde{S}_i$ for $i = 1, \ldots, y+1$. If $\eta \neq y+1$, then $S_i = \tilde{S}_i$ for $i \in \{1, \ldots, y+1\} \setminus \{\eta\}$, and

$$S_\eta = \tilde{S}_\eta \cup \{\max(S_{y+1})\}.$$

We define $A_1, A_2, \ldots, A_y$ as follows. If $\eta = y+1$, then $A_i = \max(S_i)$ for $i = 1, \ldots, y$. If $\eta \neq y+1$, then $A_i = \max(S_i)$ for $i \in \{1, \ldots, y\} \setminus \{\eta\}$, and $A_\eta = \max(S_{y+1})$.

Example 5. *Consider the data movement problem in Example 2, where $n = 21$. We can easily verify that $y = 8$ in this case. Consider the block-permutation data set labelled by ♡ in Example 2, for which we get*

$$(\alpha(1), \ldots, \alpha(21)) = (6, 1, 10, 12, 11, 9, 5, 17, 16, 14, 13, 19, 15, 8, 21, 20, 2, 18, 7, 3, 4).$$

Then, we get $\tilde{S}_1 = \{1\}$, $\tilde{S}_2 = \{2\}$, $\tilde{S}_3 = \{3, 11, 14\}$, $\tilde{S}_4 = \{4, 13, 16, 21\}$, $\tilde{S}_5 = \{5, 12, 20\}$, $\tilde{S}_6 = \{6, 10, 15\}$, $\tilde{S}_7 = \{7\}$, $\tilde{S}_8 = \{8, 18, 19\}$, $\tilde{S}_9 = \{9, 17\}$. And $\eta = 6$.

Furthermore, we get $S_1 = \{1\}$, $S_2 = \{2\}$, $S_3 = \{3, 11, 14\}$, $S_4 = \{4, 13, 16, 21\}$, $S_5 = \{5, 12, 20\}$, $S_6 = \{6, 10, 15, 17\}$, $S_7 = \{7\}$, $S_8 = \{8, 18, 19\}$, $S_9 = \{9, 17\}$.

We have $(A_1, \ldots, A_8) = (1, 2, 14, 21, 20, 17, 7, 19)$. (Note how $S_1, \ldots, S_{y+1}$ and $D_{A_1}, \ldots, D_{A_y}$ appear in Fig. 2.)

Lemma 6. *$(\alpha(A_1), \alpha(A_2), \ldots, \alpha(A_y))$ is a permutation of $(1, 2, \ldots, y)$.*

Proof: By the definition of $\tilde{S}_i$ and S_i, we can see $\alpha(A_i) \in \{1, \ldots, y\}$ for $i \in \{1, \ldots, y\}$. Since α is a bijection, $\alpha(A_i) \neq \alpha(A_j)$ when $i \neq j$. ■

Let γ be the permutation over $(1, 2, \ldots, y)$ such that for $i \in \{1, \ldots, y\}, \gamma(i) = \alpha(A_i)$. Let γ^{-1} be the inverse function of γ. Since γ is a permutation, it can be decomposed into disjoint permutation cycles. (A permutation cycle in γ is an ordered set of distinctive integers $(x_0, x_1, \ldots, x_{z-1})$, where $x_i \in \{1, 2, \ldots, y\}$ for $i \in \{0, 1, \ldots, z-1\}$, such that for $i = 0, 1, \ldots, z-1$, $\gamma(x_i) = x_{i+1 \bmod z}$.)

For $i = 1, \ldots, y$, we define the data b_i as follows. If i is not the greatest number in its corresponding permutation cycle in γ, then

$$b_i = D_{A_{\gamma^{-1}(i)}}.$$

Otherwise,

$$b_i = \mathbf{0}.$$

(Here $\mathbf{0}$ denote a page of data where all the bits are 0.)

Example 7. *We follow Example 5, where $y = 8$. We have $(\gamma(1), \gamma(2), \ldots, \gamma(8)) = (6, 1, 8, 4, 3, 2, 5, 7)$, and $(\gamma^{-1}(1), \gamma^{-1}(2), \ldots, \gamma^{-1}(8)) = (2, 6, 5, 4, 7, 1, 8, 3)$. The permutation γ consists of three permutation cycles: $(6, 2, 1)$, $(8, 7, 5, 3)$, and (4). So we have $(b_1, b_2, \ldots, b_8) = (D_{A_2}, D_{A_6}, D_{A_5}, \mathbf{0}, D_{A_7}, \mathbf{0}, D_{A_8}, \mathbf{0}) = (D_2, D_{17}, D_{20}, \mathbf{0}, D_7, \mathbf{0}, D_{19}, \mathbf{0})$.*

Let $\oplus$ denote the bit-wise XOR operation. In the following, the summation sign $\sum$ also denotes the $\oplus$ operation. We can now specify the *coded data* written into B_{i-1} in the *stage one* of the algorithm. For $i = 1, \ldots, y$, the coded data written into B_{i-1} is

$$b_i \oplus \sum_{j \in S_i} D_j;$$

and the coded data written into B_y is

$$\sum_{j \in S_{y+1}} D_j.$$

Example 8. *We follow Example 5. When we use our algorithm to move data, the data stored in the $n + \delta = 22$ blocks at different times are shown in Fig. 3. For $i = 0, 1, \ldots, 21$, the data in the column labelled by i are the data stored in the block B_i.*

We can see that in stage one, the algorithm writes linear functions of the data into $B_0, B_1, \ldots, B_y$. In stage two, the algorithm writes the final data into $B_{y+1}, B_{y+2}, \ldots, B_n$. In step three, the algorithm writes the final data into $B_y, B_{y-1}, \ldots, B_1$.

To show that the algorithm is correct, we just need to show that at all time, there are n linearly independent pages of data stored in the flash memory. This is proved in the following theorem. The proof also shows that when the algorithm runs, both encoding and decoding of the data are easily computable.

Theorem 9. *When the data-movement algorithm runs, there are always n linearly independent pages of data stored in the flash memory, which can be used to recover all the original data.*

Proof: We first show how to build a forest as the one in Fig. 2, which contains all the data stored during the data movement process. The forest has n vertices, representing the data

$$D_1, \ldots, D_n.$$

(So we will let D_i also denote its corresponding vertex.) For every edge in the forest (which can be a hyper-edge), the data it represents are the XOR of the edge's incident vertices. Let's explain how the edges are built:

- First, for $i = 1, \ldots, y+1$, if $|S_i| > 1$ (which means $\{i\} \subset S_i$), then the vertices in $\{D_j | j \in S_i\}$ are incident to the same edge.
- Second, for every permutation cycle $(x_0, x_1, \ldots, x_{z-1})$ in γ, for $i = 0, 1, \ldots, z-1$, if $x_i \neq \max\{x_0, x_1, \ldots, x_{z-1}\}$, then vertex $D_{A_{\gamma^{-1}(i)}}$ is incident to the edge with vertices in $\{D_j | j \in S_i\}$ as endpoints. (If $S_i = \{i\}$, then connect vertex $D_{A_{\gamma^{-1}(i)}}$ to vertex D_i.)

It is simple to verify that the data in the forest are exactly the same data used by the algorithm.

We now prove that there are always n linearly independent pages of data stored in the flash memory. First, for those data in $\{D_i | y+2 \leq i \leq n, \alpha(i) = i-1\}$, they are always stored in their original form, and the only thing the algorithm does to them is to move them – say it is data D_i – from block B_i to B_{i-1} (in *stage two* of the algorithm). So in the following, we consider the rest of the data, namely, the data in $\{D_i | i \in S_1 \cup S_2 \cup \cdots \cup S_{y+1}\} = \{D_i | i \in \tilde{S}_1 \cup \tilde{S}_2 \cup \cdots \cup \tilde{S}_{y+1}\}$.

Throughout the stage one and stage two of the algorithm, by Lemma 4, for the data in $\{D_j | j \in \tilde{S}_i\}$ (for $i = 1, \ldots, y+1$), at most one page of data may not be stored in its original form. In that case, the data represented by the edge that has $\{D_j | j \in S_i\}$ as endpoints has been stored in block B_{i-1}, whose representation contains the form $\sum_{j \in \tilde{S}_i} D_j$. (Note the close relation between $\tilde{S}_i$ and S_i.) They together are sufficient for recovering all the data in $\tilde{S}_i$.

When the stage two of the algorithm ends, the only data that may not be stored in their original form in the flash memory are

$$D_{A_1}, D_{A_2}, \ldots, D_{A_y}.$$

And by the requirement of the data movement problem, they need to be moved to the blocks $B_1, B_2, \ldots, B_y$ (in some permuted order). (All the other $n - y$ pages of data have been stored in their original form in their respective destination blocks, which are $B_{y+1}, B_{y+2}, \ldots, B_n$.) We can partition $\{D_{A_1}, D_{A_2}, \ldots, D_{A_y}\}$ into subsets according to the permutation cycles in the permutation γ. We see that for every permutation cycle $(x_0, x_1, \ldots, x_{z-1})$ in γ, the data $D_{A_{x_0}}, D_{A_{x_1}}, \ldots, D_{A_{x_{z-1}}}$ are connected by a path in the forest. (For example, in Fig. 2, corresponding to the three permutation cycles $(6, 2, 1)$, $(8, 7, 5, 3)$ and (4), we have the three paths $(D_{17} \to D_2 \to D_1)$, $(D_{19} \to D_7 \to D_{20} \to D_{14})$, and (D_{21}).)

Consider such a path. Say it is

$$(D_{A_{x_0}} \to D_{A_{x_1}} \to \cdots \to D_{A_{x_{z-1}}}).$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Originally		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_{15}	D_{16}	D_{17}	D_{18}	D_{19}	D_{20}	D_{21}
After stage one	$D_1 \oplus D_2$	$D_2 \oplus D_{17}$	$D_3 \oplus D_{11} \oplus D_{14} \oplus D_{20}$	$D_4 \oplus D_{13} \oplus D_{16} \oplus D_{21}$	$D_5 \oplus D_{12} \oplus D_{20} \oplus D_7$	$D_6 \oplus D_{10} \oplus D_{15} \oplus D_{17}$	$D_7 \oplus D_{19}$	$D_8 \oplus D_{18} \oplus D_{19}$	$D_9 \oplus D_{17}$		D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}	D_{16}	D_{17}	D_{18}	D_{19}	D_{20}	D_{21}
After stage two	$D_1 \oplus D_2$	$D_2 \oplus D_{17}$	$D_3 \oplus D_{11} \oplus D_{14} \oplus D_{20}$	$D_4 \oplus D_{13} \oplus D_{16} \oplus D_{21}$	$D_5 \oplus D_{12} \oplus D_{20} \oplus D_7$	$D_6 \oplus D_{10} \oplus D_{15} \oplus D_{17}$	$D_7 \oplus D_{19}$	$D_8 \oplus D_{18} \oplus D_{19}$		D_6	D_3	D_5	D_4	D_{11}	D_{10}	D_{13}	D_9	D_8	D_{18}	D_{12}	D_{16}	D_{15}
After stage three		D_2	D_{17}	D_{20}	D_{21}	D_7	D_1	D_{19}	D_{14}	D_6	D_3	D_5	D_4	D_{11}	D_{10}	D_{13}	D_9	D_8	D_{18}	D_{12}	D_{16}	D_{15}

Fig. 3. The data in the $n + \delta = 22$ blocks at different times, using the data-movement algorithm with coding over $GF(2)$.

At the end of stage two of the algorithm, all the data represented by the $z - 1$ edges in the path are stored in the flash memory, as well as the data represented by the edge that has $\{D_i | i \in S_{x_0}\}$ as endpoints. We can use them (along with the other original data stored in $B_{y+1}, \ldots, B_n$) to recover $\{D_{A_{x_0}}, D_{A_{x_1}}, \ldots, D_{A_{x_{z-1}}}\}$ by simply following the path. Then, in stage three of the algorithm, first, the data

$$D_{A_{x_{z-1}}}$$

are written into block B_{x_0} and the data represented by the edge that has $\{D_i | i \in S_{x_0}\}$ as endpoints are erased. (By following the path, we can still recover $\{D_{A_{x_0}}, D_{A_{x_1}}, \ldots, D_{A_{x_{z-1}}}\}$.) Then, every time after we write the data $D_{A_{x_i}}$ (for $i \in \{0, 1, \ldots, z-2\}$ into block

$$B_{\gamma(x_i)} = B_{x_{i+1}},$$

we erase from block

$$B_{x_{i+1}-1}$$

the (now redundant) data represented by the edge between $D_{A_{x_i}}$ and $D_{A_{x_{i+1}}} = D_{A_{\gamma(x_i)}}$. So we can still recover all the data by following the path. So the final conclusion holds. ■

III. Data Movement with Multiple Auxiliary Blocks

In this section, we study data movement with $\delta \geq 1$ auxiliary blocks, which we denote by $B_{n+1}, B_{n+2}, \ldots, B_{n+\delta}$. Note that the existing coding-based data-movement algorithms are for $\delta = 1$ [3], [4], where $2n - 1$ erasures are necessary in the worst case. We will derive bounds for the number of erasures for $\delta \geq 1$, and present a coding-based algorithm for optimal performance. We first define some terms.

For $y = 1, 2, \ldots, n-2$ and $k = y+1, y+2, \ldots, n$, we define $\mathcal{R}(y, k)$ as

$$\mathcal{R}(y, k) = \{(i, j) | k < i \leq n, 1 \leq j \leq m, y < \alpha(i, j) < k\}.$$

That is, $\{D_{i,j} | (i, j) \in \mathcal{R}(y, k)\}$ are those data that need to be moved from $B_{k+1}, \ldots, B_n$ to $B_{y+1}, \ldots, B_{k-1}$. We define $r(y)$ as

$$r(y) = \max_{k \in \{y+1, y+2, \ldots, n\}} |\mathcal{R}(y, k)|.$$

For $\Delta = 1, 2, \ldots, \delta$, define $\eta(\Delta)$ as

$$\eta(\Delta) = \min\{y \in \{1, 2, \ldots, n-2\} \mid r(y) \leq (\Delta - 1)m\}.$$

We define E_{min} as

$$E_{min} = \min_{\Delta \in \{1, 2, \ldots, \delta\}} \Delta + \eta(\Delta) + n.$$

We present a data-movement algorithm that uses E_{min} erasures. Let $\Delta_{min} \in \{1, \ldots, \delta\}$ be an integer such that $\Delta_{min} + \eta(\Delta_{min}) + n = E_{min}$. Let $\mathbb{C}$ be an $((\Delta_{min} + \eta(\Delta_{min}) + n)m, nm)$ MDS code, whose codeword is

$$(I_1 \ I_2 \ \ldots \ I_{nm} \ P_1 \ P_2 \ \ldots \ P_{(\Delta_{min} + \eta(\Delta_{min}))m}).$$

Here the codeword symbols $I_1, I_2, \ldots, I_{nm}$ are the nm pages of data originally stored in $B_1, \ldots, B_n$ (namely, the data $D_{1,1}, D_{1,2}, \ldots, D_{n,m}$), and the codeword symbols $P_1, P_2, \ldots, P_{(\Delta_{min} + \eta(\Delta_{min}))m}$ are their parity-check symbols.[1] The MDS code has the property that any nm symbols out of the $(\Delta_{\min} + \eta(\Delta_{min}) + n)m$ symbols in the codeword can be used to generate the original data $I_1, I_2, \ldots, I_{nm}$. We can use the generalized Reed-Solomon code as $\mathbb{C}$. The algorithm has three steps:

1) *Step one*: For $i = 1, 2, \ldots, \Delta_{min}$, we write the data $P_{(i-1)m+1}, P_{(i-1)m+2}, \ldots, P_{(i-1)m+m}$ into the block B_{n+i}.
2) *Step two*: For $i = 1, 2, \ldots, y$, we erase the block B_i, and write the data $P_{\Delta_{min}m+(i-1)m+1}$, $P_{\Delta_{min}m+(i-1)m+2}$, $\ldots$, $P_{\Delta_{min}m+(i-1)m+m}$ into the block B_i.
3) *Step three*: For $i = y+1, y+2, \ldots, n$ and then for $i = 1, 2, \ldots, y$, we erase the block B_i, then write into B_i the m pages of data that the data movement problem requires to move into B_i. Then, erase $B_{n+1}, B_{n+2}, \ldots, B_{n+\Delta_{min}}$.

To prove the correctness of the algorithm, we need to show that during the data movement process, there are always at least nm distinct codeword symbols from the MDS code $\mathbb{C}$ stored in the flash memory, with which we can recover all the original data.

Theorem 10. *When the data-movement algorithm runs, at any time, there are at least nm distinct codeword symbols from the MDS code $\mathbb{C}$ stored in the flash memory.*

[1] We assume that a page is large enough so that the MDS code exists. In practice, a page has 2KB or 4KB, so this is essentially always true.

Proof: When the algorithm runs, each of the parity-check symbols of the codeword is written only once. For $i = 1, 2, \ldots, nm$, the original data I_i may have two copies in the flash memory at some point, and that is when it is copied from one block B_j (for some $j \in \{\eta(\Delta_{min}) + 2, \eta(\Delta_{min}) + 3, \ldots, n\}$ to another block $B_{j'}$ (for some $j' \in \{\eta(\Delta_{min}) + 1, \eta(\Delta_{min}) + 2, \ldots, j - 1\}$). If we consider those moments in *step three* when a block among $\{B_{\eta(\Delta_{min})+1}, B_{\eta(\Delta_{min})+2}, \ldots, B_n\}$ is erased, there can be at most

$$r(\eta(\Delta_{min})) \leq (\Delta_{min} - 1)m$$

duplicated copies of the original data. So the number of distinct codeword symbols in the flash memory is at least $(n + \Delta_{min})m - m - (\Delta_{min} - 1)mm$. So the conclusion holds. ∎

The performance of the algorithm (i.e., E_{min} erasures) depends on how we label $B_1, B_2, \ldots, B_n$. There are $n!$ ways to label the n blocks storing data as $B_1, \ldots, B_n$, and every labelling can give a different value of E_{min}. If we use π to denote the labelling, then the minimum number of erasures the algorithm can achieve is $\min_\pi E_{min}$. We show that this is also *strictly optimal*. Note that here we consider the strong per-instance optimization.

Theorem 11. *For every given instance of the data movement problem, the minimum number of erasures needed to move data equals*

$$\min_\pi E_{min}.$$

Proof: First, consider a given solution to the data movement problem. Say that it uses $\Delta \leq \delta$ auxiliary blocks to really store intermediate data. Since every used block is erased at least once, and the solution uses $n + \Delta$ blocks, we can say that the solution uses $n + \Delta + y$ erasures in total for some $y \geq 0$. Let $\mathcal{P}$ be the set of blocks in $\{B_1, B_2, \ldots, B_n\}$ that are erased more than once. Clearly, $|\mathcal{P}| \leq y$. For those blocks in $\{B_1, \ldots, B_n\} \setminus \mathcal{P}$, since they are erased only once, the best strategy is to write into them the final data (i.e., the data required by the data movement problem to move into them) as soon as they are erased. Without loss of generality, let us denote the blocks in $\{B_1, \ldots, B_n\} \setminus \mathcal{P}$ by $B_{|\mathcal{P}|+1}, B_{|\mathcal{P}|+2}, \ldots, B_n$ and assume they are erased in the same order, namely, $B_{|\mathcal{P}|+1}$ is erased first and B_n is erased last. (This is just a matter of labelling.) We can see that during the data-movement process, the data originally stored in $B_{|\mathcal{P}|+1}, B_{|\mathcal{P}|+2}, \ldots, B_n$ can have duplicated copies written into the same set of blocks (that is, $B_{|\mathcal{P}|+1}, B_{|\mathcal{P}|+2}, \ldots, B_n$), and the number of such duplicated copies can be as much as $r(|\mathcal{P}|) \geq r(y)$; when that happens, a block among $\{B_{|\mathcal{P}|+1}, B_{|\mathcal{P}|+2}, \ldots, B_n\}$ is also erased. So to ensure we can recover all the original data, by the Singleton bound, Δ needs to satisfy the constraint that $r(|\mathcal{P}|) \leq (\Delta - 1)m$. Then, by the definition of E_{min}, it is not hard to see that the solution uses at least $\min_\pi E_{min}$ erasures. So at least $\min_\pi E_{min}$ erasures are needed. The other direction of the proof comes from the existence of our data-movement algorithm. ∎

It is NP hard to find the labelling π that minimizes E_{min}, because it is NP hard even when $\delta = 1$. However, we can easily get a 2-approximation algorithm by choosing parameters as specified in the following theorem.

Theorem 12. *For any labelling of the blocks $\{B_1, \ldots, B_n\}$, we can choose to use $\Delta = \min\{\delta, \lfloor n/2 \rfloor\}$ auxiliary blocks and $y - 2\Delta$ blocks among $\{B_1, \ldots, B_n\}$ to store the parity-check symbols of $\mathbb{C}$. Then the data movement algorithm will use*

$$\Delta + y + n = 2n - \min\{\delta, \lfloor n/2 \rfloor\}$$

erasures in total, which is a 2-approximation.

Proof: The chosen parameters Δ and y satisfy the constraint $r(y) \leq (\Delta - 1)m$. And every possible solution uses at least n erasures (even just to erase $B_1, \ldots, B_n$). ∎

IV. Data Aggregation

In this section, we generalize our study to *data aggregation*. It includes the data movement problem as a special case.

A. The Basic Data Aggregation Problem

In storage systems, it is common to aggregate data based on their attributes. That is, data of the same type need to be stored together; and for data of the same type, they can be stored in any order. The many examples include *wear leveling* [2], where the data of similar update frequencies are stored in the same blocks; and *databases* [5], where the data are sorted based on certain attributes. We call this the *basic data aggregation problem*. It will be extended to the case where the final data can be functions of the original data.

Definition 13. Basic Data Aggregation Problem

Consider n blocks storing data in a NAND flash memory, where every block has m pages. They are denoted by $B_1, \ldots, B_n$. There are also δ empty blocks denoted by $B_{n+1}, \ldots, B_{n+\delta}$. The m pages in block B_i are denoted by $p_{i,1}, \ldots, p_{i,m}$, for $i = 1, \ldots, n + \delta$. The data initially stored in $p_{i,j}$ are denoted by $D_{i,j}$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. Let $k \leq n$ be a positive integer. Let $\alpha(i, j)$ and $\mathcal{C}(i)$ be two functions:

$$\alpha(i, j) : \{1, \ldots, n\} \times \{1, \ldots, m\} \to \{1, \ldots, k\};$$

$$\mathcal{C}(i) : \{1, \ldots, n + \delta\} \to \{1, \ldots, k\} \cup \{\perp\}.$$

They satisfy the condition that for $i = 1, \ldots, k$, $|\{(i', j') \in \{1, \ldots, n\} \times \{1, \ldots, m\} | \alpha(i', j') = i\}| = m \cdot |\{j \in \{1, \ldots, n + \delta\} | \mathcal{C}(j) = i\}|$. We say that the data $D_{i,j}$ are of the color $\alpha(i, j)$, and that the block B_i is of the color $\mathcal{C}(i)$ if $\mathcal{C}(i) \in \{1, \ldots, k\}$. If $\mathcal{C}(i) = \perp$, then we say B_i is colorless.

The functions $\alpha(i, j)$ and $\mathcal{C}(i)$ specify the desired data aggregation. Specifically, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, the data $D_{i,j}$ need to be moved into a block of the matching color $\alpha(i, j)$. The colorless blocks need to be erased in the end. To ensure data integrity, at any moment of the data aggregation process, the data stored in the flash memory blocks should be sufficient for

recovering all the original data. The objective is to minimize the total number of block erasures in the data aggregation process.

The colors of the data represent their types, and we want to store the data of the same type together. For the data aggregation problem, we can also classify solutions based on whether they use coding or not. Let us first specify a coding-based solution.

Let x denote the number of colorless blocks among $B_1, \ldots, B_n$. (This means there are x colored blocks among $B_{n+1}, \ldots, B_{n+\delta}$.) We first shift the colors of the x colored blocks in $B_{n+1}, \ldots, B_{n+\delta}$ to the x colorless blocks in $B_1, \ldots, B_n$. We then arbitrarily decide which page each data $D_{i,j}$ should be moved to, as long as the colors of the data and the block match. We then use the data-movement algorithm to move the data. As the final step, we copy the data from the x colorless blocks in $B_1, \ldots, B_n$ to the x colored blocks in $B_{n+1}, \ldots, B_{n+\delta}$, then erase the x colorless blocks in $B_1, \ldots, B_n$. This way, this coding-based algorithm solves the data aggregation problem using at most $2n - \min\{\delta, \lfloor \frac{n}{2} \rfloor\} + x \leq 2n - \min\{\delta, \lfloor \frac{n}{2} \rfloor\} + \min\{n, \delta\} \leq 2n + \lceil n/2 \rceil = \Theta(n)$ erasures.

We will prove the benefit of coding by rigorously proving that when coding is not used, it is necessary for all algorithms to use

$$\Omega(n \log_\delta k / \log_\delta^* n)$$

erasures in the worst case.[2] We first present an algorithm without coding that uses at most $n\lceil \log_\delta k \rceil + \frac{3n}{2} = O(n \log_\delta k)$ erasures, which is very close to the proved lower bound.[3]

When coding is not used, the data are simply copied from page to page. To see how the algorithm works, let us first consider the special case where $k = \delta$.

Algorithm 14. DATA AGGREGATION FOR $k = \delta$

We label the empty blocks $B_{n+1}, B_{n+2}, \ldots, B_{n+\delta}$ with the integers $1, 2, \ldots, \delta$. Then, we perform the iteration described below. During the following iteration, whenever a block labelled by an integer $i \in \{1, \ldots, \delta\}$ becomes full (namely, when m pages of data have been written into it), we find a block that is empty at this moment, and give the label i to the empty block. The full block will no longer be labelled.

Let $Q \subseteq \{B_1, B_2, \ldots, B_n\}$ denote the set of blocks whose data have at least two different colors. That is, for $i \in \{1, \ldots, n\}$, $B_i \in Q$ if and only if $|\{\alpha(i,1), \alpha(i,2), \ldots, \alpha(i,m)\}| \geq 2$. The iteration is:

- *While $Q \neq \emptyset$, do:*
  - *Choose a block $B_i \in Q$.*
  - *For $j = 1$ to m, do: Write the data $D_{i,j}$ to a block labelled by $\alpha(i,j)$.*
  - *Erase block B_i, and remove B_i from Q.*

The above algorithm uses at most n erasures, and when it ends, for each of the n non-empty blocks, its data have the same color. (But the color of a block is not necessarily the same color of its data.) Let us prove that the algorithm can end successfully. The key is to show that during the iteration, whenever a labelled block becomes full, there must exist an empty unlabelled block (to be given the label). It is true because when a labelled block becomes full, there are at most $(\delta - 1)m$ empty pages in the labelled blocks. If we use B_i to denote the unlabelled block whose data are being copied to the labelled blocks, then at this moment, if all the data in B_i have been copied, B_i will be erased and become the empty block; otherwise, by the pigeonhole principle, there must be an empty page in another unlabelled block, which means that that unlabelled block is empty. (Note that initially, there are δm empty pages.) So the algorithm can end successfully.

We now use Algorithm 14 as a building block to solve the data aggregation problem in Definition 13.

Algorithm 15. DATA AGGREGATION WITHOUT CODING

First, divide the set of k colors, $\{1, 2, \ldots, k\}$, into δ subsets $S_1, S_2, \ldots, S_\delta$ as evenly as possible. (That is, each S_i contains either $\lceil k/\delta \rceil$ or $\lfloor k/\delta \rfloor$ colors.) See every S_i as a "super color" and use Algorithm 14 to move the data, so that in every non-empty block, all the data are of the same "super color." Then, for every $i \in [\delta]$, divide S_i into δ subsets $S_{i,1}, S_{i,2}, \ldots, S_{i,\delta}$ as evenly as possible, and use Algorithm 14 to move the data of super color S_i so that in the end, in every non-empty block, all the data have their colors belong to the same $S_{i,j}$. Repeat this process $\lceil \log_\delta k \rceil$ times, so that in the end, the data in every non-empty block have the same color. As the last step, we move the data to their target blocks (that is, blocks of the same color as the data) by copying the data from block to block.

In the above algorithm, each of the $\lceil \log_\delta k \rceil$ rounds using Algorithm 14 as a subroutine takes at most n erasures, and the last step takes at most $3n/2$ erasures. So Algorithm 15 uses at most $n\lceil \log_\delta k \rceil + \frac{3n}{2}$ erasures. We now prove that this is nearly optimal.

Theorem 16. *For $m \geq \log_\delta n / \log_\delta^* n$, when coding is not used, no data-aggregation algorithm can use less than*

$$\Omega(n \log_\delta k / \log_\delta^* n)$$

erasures in the worst case.

Proof: The proof follows from the authors' related result on data movement in [3]. Namely, in [3] we show a lower bound of $\Omega(n \cdot \min\{\log_\delta n / \log_\delta^* n, m\})$ for the special case in which the number of colors k is equal to the number of data blocks n. For our setting of m, this lower bound equals $\Omega(n \log_\delta n / \log_\delta^* n)$. The proof in [3] is based on the study of a certain *configuration* graph G that models all possible erasure strategies. The node set of G represents possible configurations of the data aggregation problem, while the edge

[2] $\log_\delta^* n$ 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. Since $\log_\delta^* n$ is practically a very small number, this lower bound is very close to $\Omega(n \log_\delta k)$.

[3] For algorithms without coding, we assume $\delta \geq 2$, because it is known that if $\delta = 1$, there are unsolvable instances if coding is not used [4].

set corresponds to the ability to move from one configuration to another using a single erasure. Roughly speaking, in [3], a lower bound on the number of erasures needed in the data aggregation problem (for k) is obtained by analyzing the diameter of G.

For the generalized lower bound, we present a reduction to the results of [3]. We assume that $k \geq \delta$ (otherwise there is a trivial tight lower bound of $\Omega(n)$). Assume that the data aggregation problem for $k \leq n$ colors can be solved using $E(n,k,\delta)$ block erasures. Let $c > 0$ satisfy $n = k^c$, namely $c = \log_k n$. Recursively using the solution for k colors (using the notion of "super colors" described in Algorithm 15), one can solve the data aggregation problem for k^2 colors using $E(n,k,\delta) + kE(n/k,k,\delta)$ erasures. In general, using a recursion of depth $\lceil c \rceil$, one can solve the data aggregation problem for n colors using $\sum_{i=0}^{\lceil c \rceil} k^i E(n/k^i,k,\delta)$ erasures.

Now assume by way of contradiction that $E(n,k,\delta) = o(n \log_\delta k / \log_\delta^* n)$ for some value of k. By the discussion above, this implies that the problem for n colors can be solved using less than $\sum_{i=0}^{\lceil c \rceil} k^i E(n/k^i,k,\delta) = o(n \log_k n \log_\delta k / \log_\delta^* n) = o(n \log_\delta n / \log_\delta^* n)$ erasures, a contradiction. In the calculation above, we use the bound $E(n/k^i,k,\delta) = o(n/k^i \cdot \log_\delta k / \log_\delta^*(n/k^i))$ for $i \leq \lceil c(1 - 1/\log_\delta^* n) \rceil$ and the bound $E(n/k^i,k,\delta) = o(n/k^i \cdot \log_\delta k)$ for remaining values of i. ■

The above result rigorously proves the benefit of coding.

B. Data Aggregation with Functions

We now study data aggregation with functions. That is, at the end of data aggregation, the stored data can be functions of the original data. Such aggregation is more general and very useful in databases, statistical applications, etc., where original data are aggregated to provide summary data [5].

The data aggregation problem considered here is defined as follows.

Definition 17. DATA AGGREGATION PROBLEM WITH FUNCTIONS

Consider n blocks storing data in a NAND flash memory, where every block has m pages. They are denoted by $B_1, \ldots, B_n$. There are also δ empty blocks denoted by $B_{n+1}, \ldots, B_{n+\delta}$. The m pages in block B_i are denoted by $p_{i,1}, \ldots, p_{i,m}$, for $i = 1, \ldots, n+\delta$. The data initially stored in $p_{i,j}$ are denoted by $D_{i,j}$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. Let $k \leq n + \delta$ be a positive integer.

For $i = 1, \ldots, k$ and $j = 1, \ldots, m$, $S_{i,j}$ is a subset of the data $\{D_{i',j'} | 1 \leq i' \leq n, 1 \leq j' \leq m\}$, and $F_{i,j}$ is an aggregation function that takes $S_{i,j}$ as input and outputs one page of aggregated data.

Let $\alpha(i,j)$ and $\mathcal{C}(i)$ be two functions:

$$\alpha(i,j) : \{1, \ldots, k\} \times \{1, \ldots, m\} \to \{1, \ldots, k\};$$

$$\mathcal{C}(i) : \{1, \ldots, n+\delta\} \to \{1, \ldots, k\} \cup \{\perp\}.$$

They satisfy the condition that for $i = 1, \ldots, k$, $|\{(i',j') \in \{1, \ldots, k\} \times \{1, \ldots, m\} | \alpha(i',j') = i\}| = m$, and $|\{j \in \{1, \ldots, n+\delta\} | \mathcal{C}(j) = i\}| = 1$. We say that the output of the aggregation function $F_{i,j}$ is of the color $\alpha(i,j)$, and that the block B_i is of the color $\mathcal{C}(i)$ if $\mathcal{C}(i) \in \{1, \ldots, k\}$. If $\mathcal{C}(i) = \perp$, then we say B_i is colorless.

The functions $\alpha(i,j)$ and $\mathcal{C}(i)$ specify where to store the output data of the aggregation functions. Specifically, for $i = 1, \ldots, k$ and $j = 1, \ldots, m$, the output of $F_{i,j}$ needs to be stored in a block of the matching color $\alpha(i,j)$. The colorless blocks need to be erased in the end. To ensure data integrity, at any moment of the data aggregation process, the data stored in the flash memory blocks should be sufficient for computing all the aggregation functions. The objective is to minimize the total number of block erasures in the data aggregation process.

Here we see the aggregation functions $F_{i,j}$ as general functions. That is, the output of an aggregation function cannot be used to recover any original data or to help compute any other aggregation function. This means that no data in $S_{i,j}$ can be removed from the flash memory before the output of the function $F_{i,j}$ is stored.

An interesting question for the above data aggregation problem with functions is to determine whether a solution exists. For the data movement problem and the basic data aggregation problem in Definition 13, it is simple: a solution exists as long as $\delta \geq 1$. However, for the problem here, the aggregation function outputs need to be stored before the inputs can be erased. In the worst case, where every $S_{i,j}$ consists of all the original data, no original data can be erased before all the function outputs are stored, so $\delta \geq k$ becomes both sufficient and necessary.

Let us use a bijective function

$$\pi : \{1, 2, \ldots, km\} \to \{1, \ldots, k\} \times \{1, \ldots, m\}$$

to denote the order in which the aggregation functions are computed in the data aggregation solution. That is, in the solution, for any $i < j$, the output of $F_{\pi(i)}$ is stored before the output of $F_{\pi(j)}$. Since a page of data $D_{i,j}$ is ready to be erased once all the functions using it as input are computed, we can derive a bijective function

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

as follows: for any $i_1, i_2 \in \{1, \ldots, n\}$ and $j_1, j_2 \in \{1, \ldots, m\}$, if $\max\{i \in \{1, \ldots, km\} | D_{i_1,j_1} \in S_{\pi(i)}\} < \max\{i \in \{1, \ldots, km\} | D_{i_2,j_2} \in S_{\pi(i)}\}$, then $\varpi^{-1}(i_1,j_1) < \varpi^{-1}(i_2,j_2)$. Then from the first to the last, the order in which the original data become ready to be erased is $D_{\varpi(1)}, D_{\varpi(2)}, \ldots, D_{\varpi(nm)}$.

In the following, we present an algorithm that uses $O(n+k)$ erasures, which is asymptotically optimal for the worst-case performance. We note that it is beyond the scope of this paper to decide when a solution exists, and we assume that here δ is sufficiently large to make the algorithm work. The algorithm, however, does make an effort to reduce the requirement on δ by erasing the original data from the flash memory as soon as possible, in order to make room to store the aggregation function outputs. It also leaves the choice of the function π

open. We note that there exist many approaches to optimize the choice of π, but for simplicity we skip the details here.

The idea of the algorithm is to first store the original data in blocks based on the order in which they will be ready to be erased (i.e., the order determined by ϖ). This way, we can erase the blocks storing the original data sequentially based on the same order.

Algorithm 18. DATA AGGREGATION WITH FUNCTIONS

Step one: Choose the function π, and compute ϖ.

Step two: Use the data movement algorithm to store the original data $\{D_{i,j} \mid i \in \{1,\dots,n\}, j \in \{1,\dots,m\}\}$ in $B_1,\dots,B_n$, such that for any two pages of data D_{i_1,j_1} (let us say it is moved to block B_{t_1}) and D_{i_2,j_2} (let us say it is moved to block B_{t_2}), if $\varpi^{-1}(i_1,j_1) < \varpi^{-1}(i_2,j_2)$, then $t_1 \le t_2$. (This step uses at most $2n - \min\{\delta, \lfloor n/2 \rfloor\}$ erasures.)

Step three: Sequentially compute $F_{\pi(1)}, F_{\pi(2)}, \dots, F_{\pi(km)}$ and write their output data into the empty blocks. (Use a new block only when the previous block becomes full of function outputs.) At the same time, sequentially erase $B_1, B_2, \dots, B_n$, where a block B_i (for $i \in \{1,\dots,n\}$) is erased as soon as the functions that use its data as inputs have all been computed. (This step uses n erasures.)

Step four: Use the data movement algorithm to move the km pages of function outputs to their destination locations. (This step uses no more than $2k + \lceil k/2 \rceil$ erasures.)

The above algorithm uses at most $3n + 2k + \lceil k/2 \rceil - \min\{\delta, \lfloor n/2 \rfloor\}$ erasures. We can further reduce the number of erasures in the following way. The idea is that since the km function outputs can be computed together at any time, in *step three* of Algorithm 18, when we write the function outputs into the blocks, we can use an MDS code to encode the function outputs, and write the codeword symbols (which are parity-check symbols of the function outputs) into the blocks. This way, in *step four* of the algorithm we can write the function outputs to their destination locations using at most $k+1$ erasures. Therefore, Algorithm 18 can solve the data aggregation problem using at most $3n + k - \min\{\delta, \lfloor n/2 \rfloor\} + 1$ erasures.

V. CONCLUSIONS

This paper studies data movement and data aggregation in flash memories, with the objective of minimizing the number of block erasures. They are both common in storage systems, and due to the unique block erasure property of flash memories, new solutions are needed. We present a very efficient data movement algorithm with coding over $GF(2)$ when only one auxiliary block is used, and extend the study to more auxiliary blocks. We present algorithms for data aggregation, and rigorously prove the benefit of coding for this problem. As future research, we will explore more specific data aggregation applications, approximation algorithms for per-instance optimization, and algorithms with low coding complexity.

REFERENCES

[1] P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni, *Flash memories*. Kluwer Academic Publishers, 1999.

[2] 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.

[3] A. Jiang, M. Langberg, R. Mateescu and J. Bruck, "Data movement in flash memories," in *Proc. 47th Annual Allerton Conference on Communication, Control and Computing (Allerton09)*, Monticello, IL, September 2009.

[4] 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, June 28 - July 3, 2009, pp. 1229-1233.

[5] S. Lee and B. Moon, "Design of flash-based DBMS: An in-page logging approach," in *Proc. ACM International Conference on Management of Data (SIGMOD)*, 2007, pp. 55-66.

[6] S. Nath and A. Kansal, "FlashDB: Dynamic self-tuning database for NAND flash," in *Proc. 6th International Conference on Information Processing in Sensor Networks (IPSN)*, 2007, pp. 410-419.