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 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]. Typically, a page stores 2KB to 4KB of data, and a block has 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 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 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], [5]. Although data movement is common in all storage systems, the unique block erasure property of flash memories calls for special solutions.

In the data movement problem [3], [5], 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 [5] 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 [5], which is worst-case optimal. The algorithm in [5] 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 show the answer is positive by presenting an 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 [6]. The external memories of sensors often use flash memories, 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.

Due to the space limitation, we skip some details in multiple places. Interested readers are referred to [4] for the full paper.

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 [5].

Definition 1. Data Movement Problem

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

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

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

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 $(i,j) \in \{1,\dots,n\} \times \{1,\dots,m\}$.

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 $B_1, \dots, B_n$ has at least one page of data that needs 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 $\{D_{1,j_1}, D_{2,j_2}, \dots, D_{n,j_n}\}$ are called a *block-permutation data set* if $\{\alpha(1,j_1), \alpha(2,j_2), \dots, \alpha(n,j_n)\} = \{1,2,\dots,n\}$. Clearly, the data in a block-permutation data set belong to n different blocks (i.e., $B_1, \dots, B_n$) both before and after the data movement process. It is proved in [5] that the nm pages of data in $B_1, \dots, B_n$ 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 this section, we consider $\delta = 1$. Let y be the smallest integer in $\{1,2,\dots,n-2\}$ with this property: *for any $i \in \{y+3, y+4, \dots, n\}$ and $j \in \{1,\dots,m\}$, either $\alpha(i,j) \le y$ or $\alpha(i,j) \ge i-1$.* Our algorithm will use $n + y + 1 \le 2n - 1$ erasures. This is *worst-case optimal*, because there are known cases where $2n-1$ erasures are necessary [5]. Note that we can label the n blocks storing data as $B_1, \dots, B_n$ 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 known that there is a solution with $n + z + 1$ erasures if and only if we can label the n blocks as $B_1, \dots, B_n$ such that $y \le z$ [5]. Therefore, our algorithm can also be readily utilized in per-instance optimal solutions. However, it is NP hard to label $B_1, \dots, B_n$ such that y is minimized [5].

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 $i = 0, 1, \dots, n$, 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 $i = 1, \dots, n$, let D_i denote the data originally stored in B_i. For $i = 1, \dots, n$, we use $\alpha(i) \in \{1, \dots, n\}$ 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, $\forall\ i \in \{1,\dots,n\}$, $\alpha(\alpha^{-1}(i)) = i$.)

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 $i = 1, 2, \dots, y+1$, we write some coded data (which are the XOR of the original data and will be defined later) into B_{i-1}, then erase B_i.
2) *Stage two*: For $i = y+2, y+3, \dots, n$, 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 $i = y-1, y-2, \dots, 0$, 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 hyperedge) 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 $D_1, \dots, D_n$. The forest structure makes it very efficient to analyze the linear independency.

Let us show how the forest is obtained. For $i = 1, 2, \dots, y+1$, we define $\tilde{S}_i \subseteq \{i, i+1, \dots, n\}$ as a set that is recursively constructed as follows: 1) $i \in \tilde{S}_i$; 2) For any $j \in \tilde{S}_i$, if $\max\{j, y+1\} \le \alpha(j) < n$, then $\alpha(j) + 1 \in \tilde{S}_i$.

Lemma 3. *The $y+1$ sets $\tilde{S}_1, \tilde{S}_2, \dots, \tilde{S}_{y+1}$ form a partition of the set $\{1,2,\dots,n\} \setminus \{i | y+2 \le i \le n, \alpha(i) = i-1\}$.*

Proof: We need to prove that: 1) $\tilde{S}_i \cap \tilde{S}_j = \emptyset$ for any $i \ne j$; 2) For any $i \in \{1,\dots,n\}$, $i \notin \cup_{i=1}^{y+1} \tilde{S}_i$ if and only if $i \ge y+2$ and $\alpha(i) = i-1$.

For $i \in \{1, \dots, y+1\}$, the integers in $\tilde{S}_i$ form a prefix of the sequence: $\{i,\ \alpha(i)+1,\ \alpha(\alpha(i)+1)+1,\ \alpha(\alpha(\alpha(i)+1)+1)+1,\ \dots\}$. 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 $i \ne j$.

$$\begin{array}{ccccccccccccccccccccc}
6^{\heartsuit} & 4^{\spadesuit} & 10^{\heartsuit} & 11^{\diamondsuit} & 2^{\diamondsuit} & 3^{\spadesuit} & 5^{\spadesuit} & 17^{\heartsuit} & 16^{\heartsuit} & 14^{\heartsuit} & 12^{\diamondsuit} & 1^{\spadesuit} & 16^{\diamondsuit} & 3^{\diamondsuit} & 17^{\spadesuit} & 21^{\diamondsuit} & 2^{\heartsuit} & 17^{\diamondsuit} & 6^{\diamondsuit} & 19^{\diamondsuit} & 4^{\heartsuit} \\
15^{\spadesuit} & 1^{\heartsuit} & 9^{\spadesuit} & 10^{\spadesuit} & 11^{\heartsuit} & 1^{\diamondsuit} & 9^{\diamondsuit} & 11^{\spadesuit} & 13^{\diamondsuit} & 14^{\diamondsuit} & 13^{\heartsuit} & 19^{\heartsuit} & 12^{\spadesuit} & 19^{\spadesuit} & 18^{\diamondsuit} & 20^{\heartsuit} & 5^{\diamondsuit} & 18^{\heartsuit} & 20^{\spadesuit} & 7^{\spadesuit} & 8^{\spadesuit} \\
7^{\diamondsuit} & 4^{\diamondsuit} & 8^{\diamondsuit} & 12^{\heartsuit} & 2^{\spadesuit} & 9^{\heartsuit} & 5^{\heartsuit} & 10^{\diamondsuit} & 16^{\spadesuit} & 14^{\spadesuit} & 13^{\spadesuit} & 15^{\diamondsuit} & 15^{\heartsuit} & 8^{\heartsuit} & 21^{\heartsuit} & 18^{\spadesuit} & 21^{\spadesuit} & 6^{\spadesuit} & 7^{\heartsuit} & 3^{\heartsuit} & 20^{\diamondsuit}
\end{array}$$

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.

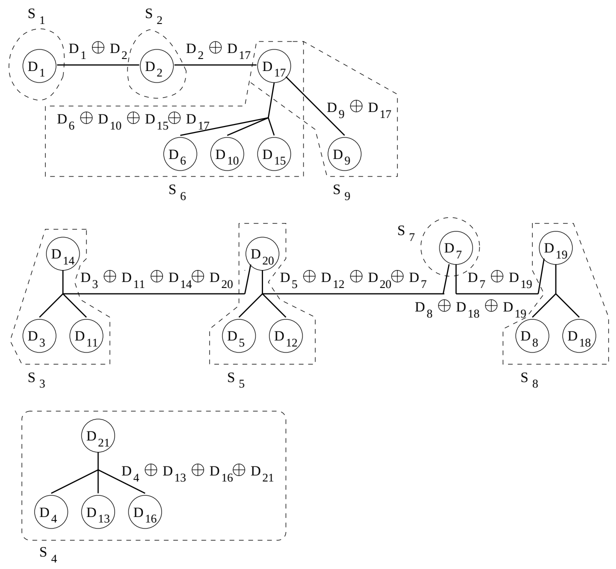

Fig. 2. Data movement with coding over $GF(2)$.

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 $i \notin \cup_{j=1}^{y+1} \tilde{S}_j$, then $i \geq y+2$. Consider $i \geq y+2$. 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 $\tilde{S}_x$. ■

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 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 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 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 Example 2, where $n = 21$. We can verify that $y = 8$ here. Consider the block-permutation data set labelled by $\heartsuit$, 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 $S_i = \tilde{S}_i$ for $i \in \{1, \ldots, 9\} \setminus \{6\}$, and $S_6 = \{6, 10, 15, 17\}$. And we get $(A_1, \ldots, A_8) = (1, 2, 14, 21, 20, 17, 7, 19)$. (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 $i = 0, 1, \ldots, z-1$, $\gamma(x_i) = x_{i+1 \mod z}$.)

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 *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 block B_i.*

The algorithm is proved correct by the following theorem.

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 present a sketch of the proof here. 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.)

During stages one and 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, and it is compensated for by the stored data that contains the form $\sum_{j \in \tilde{S}_i} D_j$. During stage three, the only data not stored in their original form are $D_{A_1}, \ldots, D_{A_y}$, which correspond to γ. And they can be recovered by decoding data using the permutation cycles in γ. For the detailed proof, please see [4]. ■

III. Data Movement with $\delta \geq 1$

We now study the data movement problem with $\delta \geq 1$ auxiliary blocks, which we denote by $B_{n+1}, B_{n+2}, \ldots, B_{n+\delta}$.

For $y = 1, 2, \ldots, n-2$ and $k = y+1, y+2, \ldots, n$, define $\mathcal{R}(y,k) = \{(i,j) | k < i \leq n, 1 \leq j \leq m, y < \alpha(i,j) < k\}$. 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 $B_{y+1}, \ldots, B_{k-1}$. Define $r(y) = \max_{k \in \{y+1, y+2, \ldots, n\}} |\mathcal{R}(y,k)|$. For $\Delta = 1, 2, \ldots, \delta$, define $\eta(\Delta) = \min\{y \in \{1, 2, \ldots, n-2\} \mid r(y) \leq (\Delta - 1)m\}$. 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 original data $D_{1,1}, D_{1,2}, \ldots, D_{n,m}$, and $P_1, P_2, \ldots, P_{(\Delta_{min} + \eta(\Delta_{min}))m}$ are the parity-check symbols. We can use the generalized Reed-Solomon code as $\mathbb{C}$. The algorithm has three steps: *(1) For $i = 1, 2, \ldots, \Delta_{min}$, we write the data $P_{(i-1)m+1}, P_{(i-1)m+2}, \cdots, P_{(i-1)m+m}$ into the block B_{n+i}; (2) 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) 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 $B_{n+1}, B_{n+2}, \ldots, B_{n+\Delta_{min}}$.* The correctness of the algorithm is proved in [4].

There are $n!$ ways to label the n blocks originally 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 present the proof in [4] that this is also *strictly optimal* for the given instance. However, it is NP hard to choose the best π. Nevertheless, we can always obtain a 2-approximation by selecting parameters as specified in the theorem below. (For the detailed analysis, please refer to [4].)

Theorem 10. *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 $y = n - 2\Delta$ 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.*

IV. Data Aggregation

We now generalize our study to *data aggregation*. It includes the data movement problem as a special case. We first study the *basic data aggregation problem*, where data of the same type need to be stored together, but their order is not specified. It will be extended later to the case where the final data can be functions of the original data.

Definition 11. 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 $B_{n+1}, \ldots, B_{n+\delta}$. The m pages in block B_i are denoted by $p_{i,1}, \ldots, p_{i,m}$, and the data initially stored in $p_{i,j}$ are denoted by $D_{i,j}$. Let $k \leq n$ be a positive integer. Let $\alpha(i,j)$ and $\mathcal{C}(i)$ be two functions:

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

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

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. The objective is to minimize the total number of block erasures.

	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 $GF(2)$.

We present a coding-based solution using at most $2.5n + 1 = O(n)$ erasures in [4]. We shall 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.[1] 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. To see how it works, let us first consider the special case where $k = \delta$.

Algorithm 12. 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.) The correctness of the algorithm is proved in [4]. We now use Algorithm 12 as a building block to solve the data aggregation problem in Definition 11.

Algorithm 13. 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 12 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 12 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.

[1] $\log^*_\delta n$ 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 $\Omega(n \log_\delta k)$.

In the above algorithm, each of the $\lceil \log_\delta k \rceil$ rounds using Algorithm 12 as a subroutine takes at most n erasures, and the last step takes at most $3n/2$ erasures. So Algorithm 13 uses at most $n\lceil \log_\delta k \rceil + \frac{3n}{2}$ erasures. The following theorem shows that this is nearly optimal. (For its detailed proof, please see [4].) The theorem rigorously proves the benefit of coding.

Theorem 14. *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.*

The data aggregation problem can be extended to the case where the final data can be functions of the original data. Due to the space limitation, we leave the detailed analysis in [4].

Acknowledgment

This work was supported in part by the NSF CAREER Award CCF-0747415, NSF grant ECCS-0802107, ISF grant 480/08, and Caltech Lee Center for Advanced Networking.

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 Allerton Conference*, 2009, pp. 1031-1038.

[4] A. Jiang, M. Langberg, R. Mateescu and J. Bruck, "Data movement and aggregation in flash memories," Caltech Technical Report, Jan. 2010. Online: $http://www.paradise.caltech.edu/etr.html$.

[5] 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 ISIT*, Seoul, Korea, June 28 - July 3, 2009, pp. 1229-1233.

[6] S. Lee and B. Moon, "Design of flash-based DBMS: An in-page logging approach," in *Proc. ACM SIGMOD*, 2007, pp. 55-66.