Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems

Cheng Huang, Minghua Chen, and Jin Li
Microsoft Research, Redmond, WA 98052

***Abstract*— We describe flexible schemes to explore the trade-offs between storage space and access efficiency in reliable data storage systems. Aiming at this goal, two fundamentally different classes of codes are introduced under the same naming umbrella – Pyramid Codes. The basic Pyramid Codes are simply derived from any existing codes (preferably MDS codes [18]), and thus all existing work on optimizing encoding/decoding directly apply. The generalized Pyramid Codes are radically advanced new codes, which can further improve reliability and/or access efficiency upon the basic Pyramid Codes. Moreover, we define a necessary condition for any failure pattern to be recoverable and show the generalized Pyramid Codes are optimal under the condition. To our best knowledge, this is the *first* work to define such a condition and the generalized Pyramid Codes are the *only* known non-MDS codes with such optimal property.**

I. INTRODUCTION

A promising direction in building large scale storage systems is to harness the collective storage capacity of massive commodity computers. While many systems demand high reliability (such as five 9s), individual components can rarely live up to that standard. Indeed, a recent study [25] on disk drives shows that the real-world reliability could be far less than expected.

On the other hand, large scale production systems (e.g. GFS [9]) have successfully demonstrated the feasibility of building reliable data storage systems using much less reliable commodity components. These systems often use replication schemes to ensure reliability, where each data block is replicated a few times. Trivial as it seems, there are sound reasons for such a choice. The simplicity of design, implementation and verification is perhaps the most important one. Another reason is because replication schemes often demonstrate good I/O performance. For instance, in a 3-replication scheme (each data block is stored with 2 additional replicas), writing a data block takes 3 write operations (1 write to itself and 2 to its replicas) and reading simply takes 1 read operation (from the original data block or either of the replicas).

On the downside, replication schemes consume several times more storage spaces than the data collection itself. In data centers, storage overhead directly translates into costs in hardware (disk drives and associated machines), as well as costs to operate systems, which include building space, power, cooling, and maintenance, etc. As a matter of fact, it is recently reported that over 55% of the cost of a typical data center, which provides Microsoft's Windows Live services, is due to building, power distribution and equipments [13]. In wide area storage applications, the storage overhead also means much less effective usage of allocated spaces. For instance, WheelFS [26] proposes to build a distributed storage system using PlanetLab machines, where users often have storage quotas. Shrinking the effective storage usage to $1/3$ or even less (as the reliability of individual PlanetLab machines is lower than that in data centers, higher replication ratio is often required) will *not* appear as an attractive solution.

Naturally, many *Erasure Resilient Coding* (ERC) based schemes (e.g. Oceanstore [16]) are proposed to reduce the storage overhead. In a typical ERC scheme, a certain mathematical transform maps k data blocks into n total blocks (k original data and $n-k$ redundant). Blocks often have the same size and can be physically mapped to bytes, disk sectors, hard drives, and computers, etc. When failures happen, failed blocks (or simply *erasures*) can be recovered using other available data and redundant blocks (imaginably, via the inverse of the mathematical transform). Such an ERC scheme is often called a (n, k)-ERC scheme. Assuming a $(16, 12)$-ERC scheme and a 3-replication scheme provide the same level of reliability (under certain failure conditions), it is obvious that the ERC scheme requires only 16 blocks in total, compared to 36 blocks by the replication scheme, to store 12 data blocks.

Apparently, the storage savings of ERC schemes are superior. However, beyond RAID systems [4], such schemes are yet to see any large scale production level adoption. We believe there are two fundamental obstacles. First, it's very difficult to get ERC schemes right. Consistency issue has long been a huge concern. Despite of numerous efforts to solve the problem, all solutions remain complicated. It's very challenging to design and implement, not even mention to verify, such schemes. Sometimes, the complexity simply scares engineering efforts away. Second, ERC schemes often suffer greatly on the I/O performance. In the $(16, 12)$-ERC scheme, writing a data block takes 5 write operations (1 write to itself, 4 reads of the redundant blocks, 4 operations to compute the change, and then 4 writes to the redundant blocks [1]), and reading takes 12 read operations when hitting a failed data block (in order to perform the inverse of the mathematical transform).

Some of the obstacles, however, can be largely avoided by exploring practical storage needs. Many production services have been successfully built on top of append-only storage systems, where write operations only append to the end of existing data and data is rarely modified once written. This is viable as many data collections are by large static (notable examples are ever exploding media content collections, due to the boom of portable music devices and Internet video). We believe that combining ERC schemes together with replication schemes is a right approach, and several goals can be achieved simultaneously: 1) simplified design, implementation and verification; 2) low storage overhead; and

3) high I/O performance. In particular, the reliability of new data is ensured by replication. Only completed data blocks are used to compute redundant blocks. Replicas are removed only after the redundant blocks are successfully stored. In this way, the consistency issue of ERC schemes is greatly alleviated. Moreover, writing a data block has exactly the same overhead as in a pure replication scheme, and the write performance is *not* affected. Note that the ERC part can be considered as post-processing, so it can be put off until the system utilization enters a valley period.

To this end, the remaining issue is how to improve the read performance. Indeed, the read performance has great impact on overall system performance, since it dictates peak system load and/or peak bandwidth usage. Further, as most distributed storage systems incur many more reads than writes, the read performance is a primary design concern. Instead of jumping from traditional ERC schemes to replication schemes, which *do* improve the read performance, but also increase the storage overhead significantly, we describe schemes such that the read performance can be improved with only moderate higher storage overhead. If traditional ERC schemes and replication schemes can be regarded as two extremes of the trade-offs between storage space and access efficiency, our schemes allow flexible exploration of the rest. Specifically, we introduce two fundamentally different classes of codes under the same naming umbrella – Pyramid Codes. The basic Pyramid Codes are simply derived from any existing codes (preferably MDS codes [18]) and thus all existing work on optimizing encoding/decoding directly apply. The generalized Pyramid Codes are radically advanced new codes, which can further improve reliability and/or access efficiency upon the basic Pyramid Codes. Moreover, we define a necessary condition for any failure pattern to be recoverable and show the generalized Pyramid Codes are optimal under the condition. To our best knowledge, this is the *first* work to define such a condition and the generalized Pyramid Codes are the *only* known non-MDS codes with such optimal property.

The rest of the paper is organized as follows. Section II describes the basic Pyramid Codes and Section III focuses on the generalized Pyramid Codes. Section IV lists some additional related work. We make concluding remarks in Section V, and more importantly, raise a few open issues.

II. Basic Pyramid Codes

Let a distributed storage system be composed of n blocks, where k blocks are *data blocks*, and the rest $m = n-k$ blocks are *redundant blocks*. Use $\mathbf{d}_i$ ($i = 1, \cdots, k$) to denote the data blocks, and $\mathbf{c}_j$ ($j = 1, \cdots, m$) to denote the redundant blocks.

A. Brief primer on MDS erasure resilient coding

Before presenting Pyramid Codes, let us briefly review maximum distance separable (MDS) [18] erasure resilient coding, which attracts particular attention in distributed storage system design. When an ERC scheme applies a (n, k) MDS code in a distributed storage system, $m = n - k$ redundant blocks are computed from k original data blocks. The MDS property guarantees that all the original data are accessible as long as any k among the n blocks are functional. That is, the system is resilient to arbitrary $n - k$ failures.

Many commonly used ERC schemes in storage systems are specific examples of the MDS codes. For example, the *simple parity* scheme, which is widely used in RAID-5 systems, computes the only redundant block as the binary sum (XOR) of all the data blocks. It is essentially a $(k + 1, k)$ MDS code. The replication scheme, which creates r replicas for each data block, is indeed a $(1 + r, 1)$ MDS code. Reed-Solomon codes [23] are a class of the most widely used MDS codes.

B. Basic Pyramid Codes: An example

Now we use an example to describe the basic Pyramid Codes, which can significantly improve the read performance. Our example constructs a Pyramid Code from a $(11, 8)$ MDS code, which could be a Reed-Solomon code, or other MDS codes, such as STAR [15]. (Note that MDS codes are *not* required, but Pyramid Codes constructed from MDS codes *do* have certain good properties, which will become clear later.) The Pyramid Code separates the 8 data blocks into 2 equal size groups $S_1 = \{\mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_4\}$ and $S_2 = \{\mathbf{d}_5, \mathbf{d}_6, \mathbf{d}_7, \mathbf{d}_8\}$. It keeps two of the redundant blocks from the MDS code unchanged (say $\mathbf{c}_2$ and $\mathbf{c}_3$). These two blocks are now called *global* redundant blocks, because they cover all the 8 data blocks. Next, a new redundant block is computed for group S_1, which is denoted as *group (or local)* redundant block $\mathbf{c}_{1,1}$. The computation is done as if computing $\mathbf{c}_1$ in the original MDS code, except for setting all the data blocks in S_2 to 0. Similarly, a group redundant block $\mathbf{c}_{1,2}$ is computed for S_2. It is easy to see that group redundant blocks are only affected by data blocks in the corresponding groups and *not* by other groups at all.

Algebraically, each data or redundant block can be represented as many *symbols* (or *elements*) in finite fields (or rings) [17]. The process of computing redundant blocks from data blocks is called *encoding*, and the process of computing failed data blocks from other data and redundant blocks called *decoding* (or *recovery*). Without loss of generality and yet to keep the presentation simple, we can assume each block is merely one symbol. Most ERC schemes apply linear block codes, where the redundant blocks are *linear* combinations of the data blocks. For instance, in the $(11, 8)$ MDS code, the redundant block $\mathbf{c}_1$ satisfies

$$\mathbf{c}_1 = \sum_{i=1}^{8} \alpha_i \mathbf{d}_i,$$

where α_i's are symbols in the same field (or ring). Based on this representation, the new redundant blocks in the Pyramid Code satisfy

$$\mathbf{c}_{1,1} = \sum_{i=1}^{4} \alpha_i \mathbf{d}_i, \quad \mathbf{c}_{1,2} = \sum_{i=5}^{8} \alpha_i \mathbf{d}_i.$$

Hence, $\mathbf{c}_{1,1} + \mathbf{c}_{1,2} = \mathbf{c}_1$ (all $\sum$'s and $+$'s are binary sum). To this end, the group redundant blocks can be interpreted as the *projection* of the original redundant block in the MDS code onto each group (by setting the data blocks in all other

groups to 0). Alternatively, given the group redundant blocks, they can be combined (again, binary sum) to compute the original redundant block in the MDS code. The Pyramid Code constructed in this example is shown in Figure 1. For convenience, we define the concept of *configuration*, which represents all data block subsets used to compute the redundant blocks. For instance, the configuration of this code is $\mathbf{c}_{1,1} : S_1$, $\mathbf{c}_{1,2} : S_2$, and $\mathbf{c}_2, \mathbf{c}_3 : S_1 \cup S_2$.

Now, we examine interesting properties of the Pyramid Code. First of all, the Pyramid Code has the same write overhead as the original MDS code. Whenever any data block is updated, the Pyramid Code needs to update 3 redundant blocks (both $\mathbf{c}_2$, $\mathbf{c}_3$, plus either $\mathbf{c}_{1,1}$ or $\mathbf{c}_{1,2}$), while the MDS code also updates 3 redundant blocks ($\mathbf{c}_1$, $\mathbf{c}_2$ and $\mathbf{c}_3$).

Secondly, we claim that the $(12, 8)$ Pyramid Code can also recover arbitrary 3 erasures, the same as the original $(11, 8)$ MDS code. To show this, assume there are arbitrary 3 erasures out of the 12 total blocks, which can fall into one of the following two cases: 1) both $\mathbf{c}_{1,1}$ and $\mathbf{c}_{1,2}$ are available; or 2) at least one of them is unavailable. In the first case, $\mathbf{c}_1$ can be computed from $\mathbf{c}_{1,1}$ and $\mathbf{c}_{1,2}$. Then, it becomes recovering 3 erasures from the original $(11, 8)$ MDS code, which is certainly doable. In the second case, it is impossible to compute $\mathbf{c}_1$. However, other than $\mathbf{c}_{1,1}$ or $\mathbf{c}_{1,2}$, there are at most 2 failed blocks. Hence, from the perspective of the original MDS code, there are at most 3 failures ($\mathbf{c}_1$ and those 2 failed blocks) and thus is decodable.

Third, the Pyramid Code is superior in terms of the read overhead. When only one data block fails, the Pyramid Code can decode using local redundant blocks, which leads to read overhead of 4, compared to 8 in the MDS code. Finally, note that the gain of the Pyramid Code in terms of the read overhead comes at the cost of using one additional redundant block. Hence, this example literally demonstrates the *core* concept of how the Pyramid Codes can trade storage space for access efficiency.

Next, we show detailed comparisons between the two codes. Two performance metrics are used here. When the number of failed blocks (could be data or redundant) is given (denoted as r), there are $\binom{n}{r}$ possible failure cases. The first metric, *recoverability*, represents the ratio between the number of recoverable cases and the total cases. It is directly related to the reliability of the overall storage system and one can link these two using failure probability models (we do *not* expand along this direction, as it is *not* the focus of this paper). The second metric, *average read overhead*, represents the average overhead to access each data block. Consider an example of 1 block failure in the $(11, 8)$ MDS code. If the failure is a redundant block ($3/11$ chance), then the data blocks can be accessed directly, so the average read overhead is 1. Otherwise, the failure is a data block ($8/11$ chance), then the read overhead is 8 for the failed data block and 1 for the rest 7 data blocks. Hence, the average read overhead is $(8+7)/8$. Altogether, the average read overhead is $1 \times 3/11 + (8+7)/8 \times 8/11 = 1.64$. Using these two metrics, the detailed comparisons are shown in Figure 2. We observe that the additional redundant block in Pyramid Code reduces the read overhead under all failure patterns, compared to the MDS code. Moreover, the Pyramid Code has good chance to battle additional failures (e.g. the 4^{th} failure in this case).

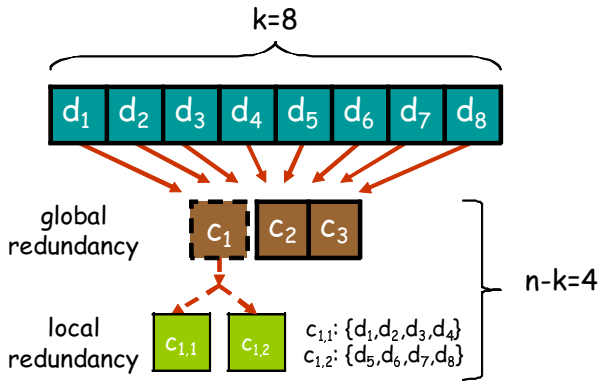


Fig. 1. Construction of a $(12, 8)$ basic Pyramid Code from a $(11, 8)$ MDS code.

# of failed blocks		0	1	2	3	4
MDS code (11, 8)	recoverability (%)	100	100	100	100	0
	avg. read overhead	1.0	1.64	2.27	2.91	-
Pyramid Code (12, 8)	recoverability (%)	100	100	100	100	68.89
	avg. read overhead	1.0	1.25	1.74	2.37	2.83

Fig. 2. Comparison between the MDS code and the basic Pyramid Code.

C. Generalization of basic Pyramid Codes

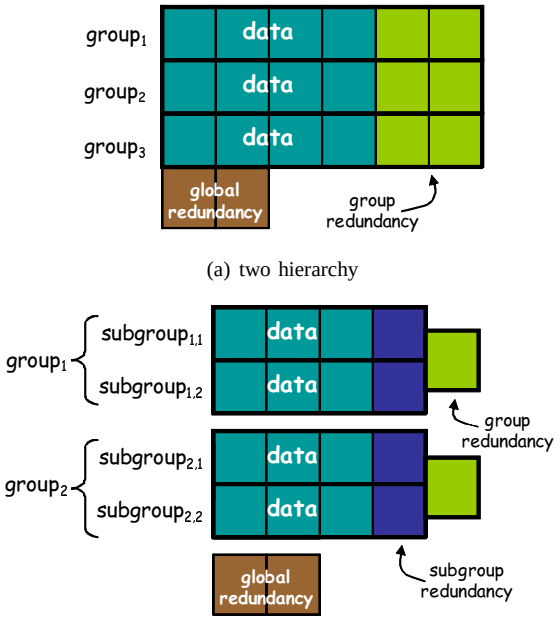
In general, a Pyramid Code can be constructed as follows. It starts with a (n, k) code (preferably a MDS code), and separates the data blocks into L disjoint groups (denoted as S_l, $l = 1, \cdots, L$), where group S_l contains k_l blocks (i.e., $|S_l| = k_l$). Next, it keeps m_1 out of the m redundant blocks unchanged, and computes $m_0 = m - m_1$ new redundant blocks for each group S_l. The j^{th} group redundant block for group S_l (denoted as $\mathbf{c}_{j,l}$) is simply a projection of the j^{th} redundant block in the original code (i.e., $\mathbf{c}_j$) onto the group S_l. In another word, $\mathbf{c}_{j,l}$ is computed the same as $\mathbf{c}_j$ in the original code, but simply setting all groups other than S_l to 0. Again, the combination of all $\mathbf{c}_{j,l}$'s for the same l yields the redundant block $\mathbf{c}_j$ in the original code. Moreover, if a Pyramid Code is constructed from a MDS code, it satisfies the following property.

Theorem 1: A basic Pyramid Code constructed from a (n, k) MDS code can recover arbitrary $m = n - k$ erasures, and each group itself is a $(k_l + m_0, k_l)$ MDS code.

Proof: The first part can be shown using a similar argument as in the previous example. Let's consider an arbitrary failure pattern with m erasures. Assuming r (out of m) erasures are among the group redundant blocks, then there are $m - r$ erasures among all the data blocks and the global redundant blocks. There are only two cases: 1) $r \geq m_0$; or 2) $r < m_0$. When $r \geq m_0$, simply assume all the group redundant blocks are failed (treating them as erasures as well). From the perspective of the original MDS code, none of the m_0 redundant blocks can be computed, which means m_0

erasures. Together with the $m-r$ erasures in the rest data and redundant blocks, there are $m_0+m-r \leq m$ erasures in total. Hence, such failure patterns are recoverable. Otherwise, when $r < m_0$, the worst case is that all the group redundant blocks $\mathbf{c}_{j,l}$'s have different j's, which means they will keep r out of the m_0 redundant blocks as erasures. Even so, from the perspective of the original MDS code, there are at most $r+(m-r) = m$ erasures in total. Hence, such failures patterns are also recoverable.

The second part is quite intuitive and can be proven by contradiction. If there exists one group S_l, which is a $(k_l + m_0, k_l)$ code but not MDS, this group must fail to recover a certain erasure pattern with m_0 failures. Now, consider a special example, where all data blocks in groups other than S_l simply have 0 values. Then, group S_l together with the m_1 global redundant blocks is equivalent to the original MDS code. Assuming the m_1 redundant blocks are additional erasures, still, the original MDS code is able to recover all data blocks, because there are $m_0+m_1 = m$ erasures in total. Since the global redundant blocks are *not* used at all, this literally means group S_l is also recoverable, which apparently contradicts with our assumption. ■



(a) two hierarchy

(b) three hierarchy

Fig. 3. Multi-hierarchical extension of basic Pyramid Codes.

D. Decoding of basic Pyramid Codes

To this end, the recovery procedure should become straightforward and we briefly summarize as follows.

- **Step** 1: start from the group level. For each group, if the available redundant blocks are no less than the failed data blocks, recover all the failed data blocks and mark all the blocks (both data and redundant) available. Whether or not the failed redundant blocks are actually computed depends on whether they are used in the following step.
- **Step** 2: move to the global level. For each j ($1 \leq j \leq m_0$), if all the group redundant block $\mathbf{c}_{j,l}$'s (over all l's) are marked as available, mark $\mathbf{c}_j$ (a redundant block in the original code) available as well. On the global level, if there are no less available redundant blocks than failed data blocks, recover all the failed blocks (otherwise, remaining failed blocks are declared unrecoverable). Moreover, when a combined $\mathbf{c}_j$ is used, it should be computed (as well as its corresponding $\mathbf{c}_{j,l}$'s if they are not yet available).

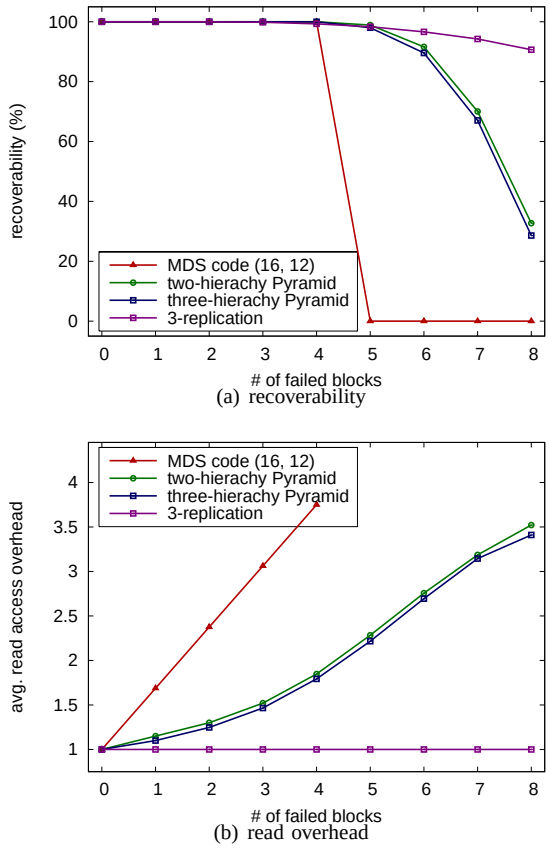
E. A multi-hierarchical extension of basic Pyramid Codes

Figure 3(a) shows another example of the basic Pyramid Codes, which is constructed from a $(16, 12)$ code. It can be considered as having two hierarchies, the global level and the group level. Then, it is conceivable that the basic Pyramid Codes can be readily extended to multiple hierarchies. For instance, Figure 3(b) shows an example of 3-hierarchy. The data blocks are separate into 2 groups, then each group further separated into 2 subgroups. Correspondingly, some redundant blocks are kept global and some are projected to compute group redundant blocks. Further, some group redundant blocks are projected to compute subgroup ones. The particular example in Figure 3(b) ends up with 2 global redundant blocks, 1 group redundant block for each group and 1 subgroup redundant block for each subgroup.

As a simple exercise, Theorem 1 can also be extended to basic Pyramid Codes with multiple hierarchies. Similarly, the decoding of multi-hierarchy basic Pyramid Codes will start with the lowest level and gradually move to the global level. *This is very similar to climbing up a Pyramid, just as the name of the codes suggests.* Note that groups (or subgroups) in the basic Pyramid Codes are *not* required to be of the same size, although it is often the case in practice for convenience.

F. MDS codes, Pyramid Codes and Replication schemes

This subsection presents comparisons among MDS codes, Pyramid Codes and replication schemes. To keep things simple, the number of data blocks is fixed to be 12 (i.e., $k = 12$). A MDS code has 4 redundant blocks and thus its $n = 12 + 4 = 16$. Two Pyramid Codes (both are $(20, 12)$ codes) are constructed from the MDS code, whose configurations are shown in Figure 3. Finally, a 3-replication scheme creates 2 replicas for each data block (hence, it is equivalent to 12 individual $(3, 1)$ MDS codes, or one jumble $(36, 12)$ code). The comparisons are shown in Figure 4 and we make the following observations. From MDS codes to Pyramid Codes and then replication schemes, the trend clearly demonstrates that adding more storage spaces can reduce the read overhead, as well as increase the recoverability. Moreover, when the storage overhead is the same (e.g. the two Pyramid Codes with different configurations), schemes with higher recoverability also occur higher read overhead. Hence, on one hand, Pyramid Codes can be used to explore more flexible ways to trade storage space for access efficiency. On the other hand, even under the same storage overhead, the configuration of Pyramid



(a) recoverability

(b) read overhead

Fig. 4. Comparison of multi-hierarchy basic Pyramid Codes with MDS codes and replication schemes (the two Pyramid Codes are both $(20, 12)$, the 3-replication is essentially a jumble $(36, 12)$ code).

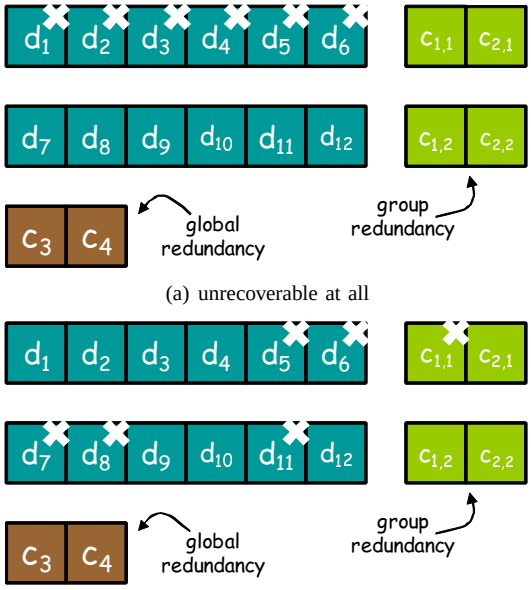
Codes should be carefully considered, such that the read overhead can be minimized, while the reliability requirement is also satisfied.

III. Generalized Pyramid Codes

In this section, we describe *generalized Pyramid Codes*, which are not trivial extensions of the basic Pyramid Codes, but rather radically advanced new ERC schemes. They also go beyond the structure of the basic Pyramid Codes, where groups lower in the hierarchy are always nested in upper ones. In the generalized Pyramid Codes, groups may overlap with each other. Nevertheless, we use the common name *Pyramid Codes* to categorize both classes of codes, as they both aim at the same goal of trading storage space for access efficiency, and also follow the same concept of *climbing up a Pyramid* during failure recovery.

A. Motivation

We use an example to explain the need to investigate beyond the basic Pyramid Codes. Figure 5 shows a configuration of a $(18, 12)$ basic Pyramid Code, which is constructed from a $(16, 12)$ MDS code. The code has 2 groups. Within each group, 6 data blocks are protected by 2 redundant blocks (thus a $(8, 6)$ MDS code). Additionally, there are 2 global redundant blocks which protect the entire 12 data blocks. From the



(a) unrecoverable at all

(b) unrecoverable as a basic Pyramid Code

Fig. 5. Examples to motivate the generalized Pyramid Codes (6 erasures each, marked by "x").

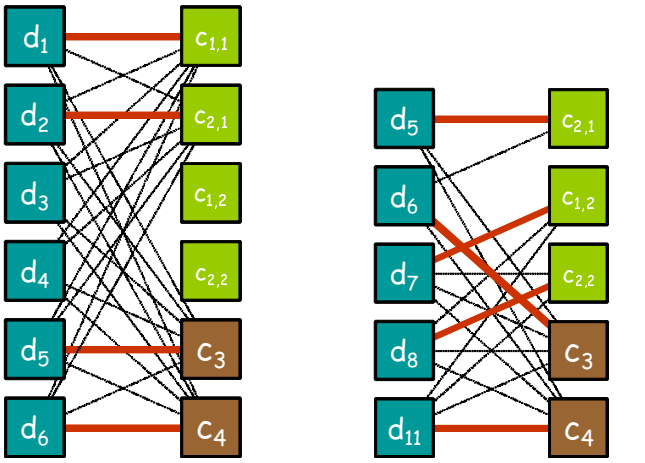
previous section, we know that the code can recover arbitrary 4 erasures. Since it has 6 redundant blocks, interesting questions to ask are: 1) what 5-erasure and 6-erasure patterns can it recover? Apparently, due to information theory limits, the code can *not* recover more than 6 erasures; and 2) more generally, can the recoverability be further improved?

In particular, we examine the two erasure patterns. The first pattern has 6 erasures and is shown in Figure 5(a). There are 6 data blocks and 2 redundant blocks available in the second group. Hence, those 2 redundant blocks are *not* useful for recovery and could be removed. Now, we are left with 6 erasures, but only 4 redundant blocks. Therefore, this erasure pattern is unrecoverable at all. The second erasure pattern is shown in Figure 5(b). It turns out that the basic Pyramid Code can *not* recover this pattern either. Following the decoding procedure, the decoder can *not* make progress within each group, where there are more failed data blocks than available redundant blocks. Hence, it moves to the global level, where it computes one more redundant block $\mathbf{c}_2$ from $\mathbf{c}_{2,1}$ and $\mathbf{c}_{2,2}$. Still, on the global level, there are only 3 available redundant blocks and yet 5 data erasures. Hence, the decoder can *not* proceed either. But, is this pattern ever possible to recover at all? In the following, we give a positive answer by presenting the generalized Pyramid Codes.

We first present a necessary condition of recoverability. If an erasure pattern is ever recoverable by any ERC scheme, the necessary condition has to be satisfied. Note that the opposite is *not* true. As a matter of fact, MDS codes are the only known codes, where an erasure pattern is recoverable whenever the necessary condition is satisfied (i.e., the condition also becomes sufficient). To our best knowledge, this is the *first* work to present such a condition. Even better, we also present the construction of non-MDS generalized Pyramid Codes, which are able to recover any erasure patterns as long

as the necessary condition is satisfied. In another word, they are the *only* non-MDS codes where the necessary condition also becomes sufficient. In this sense, the generalized Pyramid Codes are optimal (of course, MDS codes are optimal by nature).

B. Necessary condition of recoverability



(a) unrecoverable pattern (full-size matching does *not* exist)

(b) recoverable pattern (full-size matching exists)

Fig. 6. Tanner graphs (bold edges show maximum matchings).

The recoverability of *any* ERC scheme (not just Pyramid Codes) can be easily verified using a Tanner graph, which is a common tool frequently used in the study of erasure resilience coding. A Tanner graph is a bipartite graph, where nodes on the left part of the graph represent data blocks (*data nodes* hereafter), and nodes on the right represent redundant blocks (*redundant nodes*). An edge is drawn between a data node and a redundant node, if the corresponding redundant block is computed from the corresponding data block. Given an erasure pattern, a *simplified* Tanner graph (denoted as T) can be plotted to show only the *failed* data blocks and the *available* redundant blocks. For instance, the Tanner graphs corresponding to the erasure patterns in Figure 5 are shown in Figure 6.

Furthermore, we define *matching* (denoted by M) as a set of edges in a Tanner graph, where no two edges connect at the same node. The size of the matching $|M|$ equals to the number of edges in the set. Define *maximum matching* (denoted by M_m) as a matching with the maximum number of edges. Also, if $|M_m|$ equals to the number of data nodes, such a matching is called a *full-size matching* (denoted by M_f). For example, the Tanner graph in Figure 6(b) contains a full-size matching, while the one in Figure 6(a) does *not*. With these definitions, the necessary condition of recoverability is stated in the following theorem. (Note that when there is no ambiguity, blocks and nodes are used interchangeably, so as the recovery of an erasure pattern and the recover of a Tanner graph.)

Theorem 2: For any linear ERC scheme (not just Pyramid Codes), an erasure pattern is recoverable *only if* the corresponding Tanner graph contains a full-size matching.

Proof: We prove this theorem by contradiction. Examining an arbitrary *recoverable* erasure pattern, whose corresponding Tanner graph T consists of r_d data nodes and r_c redundant nodes. (Again, this means the erasure pattern has r_d failed data blocks and r_c available redundant blocks.) Obviously, $r_d \leq r_c$. Now, let's assume T does *not* contain a full-size matching. Then, the size of its maximum matching M_m is less than r_d, i.e., $|M_m| < r_d$. Based on the König-Egerváry Theorem [24] in graph theory, in a bipartite graph, the maximum size of a matching is equal to the minimum size of a node cover. Hence, a *minimum node cover* (denoted by N_c), which contains a minimum set of nodes covering all edges in T, has $|M_m|$ nodes, i.e., $|N_c| = |M_m|$. Let n_d be the number of data nodes in N_c, then $|M_m| - n_d$ is the number of redundant nodes in N_c. It is clear that $n_d \leq |M_m| < r_d$.

Now let us assume all the data blocks in N_c are somehow known (not erasures any more), then we can deduce a new erasure pattern with less failed blocks, which corresponds to a new Tanner graph T'. Any redundant node that is not in N_c can be removed from T', because those redundant nodes can only connect to the data nodes in N_c (otherwise, there will be edges in T not covered by N_c) and thus isolated in T'. Hence, there are at most $|M_m| - n_d$ redundant nodes left in T'. On the other hand, there are still $r_d - n_d$ (positive value) data nodes left. As $|M_m| - n_d < r_d - n_d$, there are less redundant nodes than the data nodes, and thus T' is not recoverable. Therefore, T should not be recoverable either, which contradicts with the assumption. ■

To this end, we have proven the necessary condition of recoverability. For interested readers, the same condition is also studied using an alternative set representation in a companion paper [3] (called *Maximally Recoverable* property there). Next, we present the generalized Pyramid Codes, which are optimal as the necessary condition also becomes sufficient.

C. Construction of generalized Pyramid Codes

1) Matrix representation of ERC schemes: The encoding process of an ERC scheme is a mathematical computation of the redundant blocks from the data blocks, which can be represented using matrix multiplication as $\mathbf{C}_m = \mathbf{G}_m \times \mathbf{D}$, where $\mathbf{C}_m$ represents the redundant blocks, as $[\mathbf{c}_1, \mathbf{c}_2, \cdots, \mathbf{c}_m]^T$, and $\mathbf{D}$ the data blocks, as $[\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_k]^T$. Again, for simplicity, each entry of $\mathbf{C}$ and $\mathbf{D}$ is merely *one* symbol in a finite field (or ring). $\mathbf{G}_m$ is a $k \times m$ *generator matrix*, whose entries are symbols in the same field (or ring). Note that $\mathbf{G}_m$ completely determines the ERC scheme. For convenient purpose, the data blocks can be regarded as special redundant blocks and the encoding process can thus be represented as $\mathbf{C} = \mathbf{G} \times \mathbf{D}$, where $\mathbf{C} = [\mathbf{D}\ \mathbf{C}_m]^T$ and $\mathbf{G} = [\mathbf{I}\ \mathbf{G}_m]^T$. Now, the new generator matrix $\mathbf{G}$ is a $k \times n$ matrix, which contains a $k \times k$ identity matrix on the top and $\mathbf{G}_m$ on the bottom. When block failures happen, some data and redundant blocks become erasures. In the algebraic representation, this is equivalent to saying that some entries in $\mathbf{C}$ are missing. If we cross out all missing entries in $\mathbf{C}$ (make it $\mathbf{C}_s$) and corresponding rows in $\mathbf{G}$ (make it $\mathbf{G}_s$), we will get the following

$$\mathbf{C}_s = \mathbf{G}_s \times \mathbf{D}. \tag{1}$$

Here, $\mathbf{G}_s$ is a *generator submatrix*, obtained from $\mathbf{G}$ by eliminating rows corresponding to erasure blocks (both data and redundant). To this end, both $\mathbf{C}_s$ and $\mathbf{G}_s$ are completely known, while $\mathbf{D}$ contains some unknowns (those failed data blocks). The recovery is essentially solving the set of linear equations in Eq(1) for those unknowns. It is clear that the necessary and sufficient condition of recoverability is that Eq(1) should be solvable (please see [20] for a tutorial).

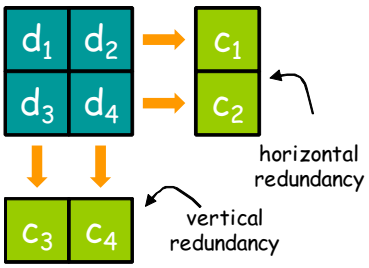


Fig. 7. Example of generalized Pyramid Code construction.

2) A construction example: Now, we use a simple example to illustrate the construction of the generalized Pyramid Codes. Let's consider the configuration shown in Figure 7, which is $\mathbf{c}_1$: $\{\mathbf{d}_1, \mathbf{d}_2\}$, $\mathbf{c}_2$: $\{\mathbf{d}_3, \mathbf{d}_4\}$, $\mathbf{c}_3$: $\{\mathbf{d}_1, \mathbf{d}_3\}$, and $\mathbf{c}_4$: $\{\mathbf{d}_2, \mathbf{d}_4\}$. In the matrix presentation, it is $\mathbf{C} = \mathbf{G} \times \mathbf{D}$, where $\mathbf{C} = [\mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_4, \mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, \mathbf{c}_4]^T$, $\mathbf{D} = [\mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_4]^T$ and

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ g_{5,1} & g_{5,2} & 0 & 0 \\ 0 & 0 & g_{6,3} & g_{6,4} \\ g_{7,1} & 0 & g_{7,3} & 0 \\ 0 & g_{8,2} & 0 & g_{8,4} \end{bmatrix}, \tag{2}$$

where $\mathbf{g}_i$ is the entire row i of $\mathbf{G}$, and $g_{i,j}$ the entry at row i and column j ($1 \leq i \leq n$, $1 \leq j \leq k$). Using $\mathbf{c}_1$ as an example, it is computed as $\mathbf{c}_1 = g_{5,1}\mathbf{d}_1 + g_{5,2}\mathbf{d}_2$. It should be clear that $g_{5,3} = g_{5,4} \equiv 0$, as $\mathbf{c}_1$ is *not* computed from $\mathbf{d}_3$ and $\mathbf{d}_4$. The construction of a generalized Pyramid Code is essentially to fill in all non-zero entries in $\mathbf{G}$, such that the code is optimal. The algorithm works as follows.

We start with an identity matrix $\mathbf{G} = \mathbf{I}_{4\times 4}$ and add $\mathbf{g}_m$'s ($5 \leq m \leq 8$) *one by one*. First, we elaborate on how to add $\mathbf{g}_5$. Note that the physical meaning of adding $\mathbf{g}_5$ to $\mathbf{G}$ is defining the computation of the first redundant block $\mathbf{c}_1$. To achieve optimality, it is desirable that if any data block is failed, it should be recoverable from $\mathbf{c}_1$ together with the rest 3 data blocks. In terms of the matrix representation, this is equivalent to saying that a generator submatrix $\mathbf{G}_s$, formed by $\mathbf{g}_5$ and any 3 rows out of $\mathbf{g}_1$ to $\mathbf{g}_4$, should be invertible. Let's focus on one specific case, where $\mathbf{G}_s$ consists of $\mathbf{g}_1$, $\mathbf{g}_2$, $\mathbf{g}_3$ and $\mathbf{g}_5$. The invertibility (or non-singularity) of $\mathbf{G}_s$ requires that $\mathbf{g}_5$ is independent of $\mathbf{g}_1$, $\mathbf{g}_2$ and $\mathbf{g}_3$. Denote $\mathbf{S}$ as the submatrix composed of $\mathbf{g}_1$, $\mathbf{g}_2$ and $\mathbf{g}_3$. Then, $\mathbf{g}_5$ should *not* be contained in the subspace spanned by $\mathbf{S}$, denoted as $\mathrm{span}(\mathbf{S})$. This is further equivalent to requiring that $\mathbf{g}_5$ should *not* be orthogonal to the *null space* of $\mathbf{S}$. Since the rank of $\mathbf{S}$ is 3, its null space is simply a vector, denoted as $\mathbf{u}$. Hence, the orthogonality boils down to require that the dot product of $\mathbf{u}$ and $\mathbf{g}_5$ is non-zero, i.e., $\mathbf{u} \cdot \mathbf{g}_5 \neq 0$. To accommodate any 3 rows out of $\mathbf{g}_1$ to $\mathbf{g}_4$, we simply enumerate through all sub-matrices composed by any 3 rows ($\binom{4}{3}$ cases in total), where each submatrix corresponds to one null space vector. A null space matrix $\mathbf{U}$ is built to hold all the null space vectors, and the ultimate goal is to find a $\mathbf{g}_5$, such that $\forall\ \mathbf{u} \in \mathbf{U}$, $\mathbf{u} \cdot \mathbf{g}_5 \neq 0$.

In this particular case, the null space matrix is as simple as

$$\mathbf{U} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \tag{3}$$

Then, finding a desirable $\mathbf{g}_5$ is straightforward even by trial-and-error. For instance, our random choice yields desirable outcomes: $g_{5,1} = 1$ and $g_{5,2} = 142$ (all values are in a finite field GF(2^8), which is generated using $x^8+x^4+x^3+x^2+1$ as the prime polynomial [17]). Straightforward as it seems, there are two things deserve special attention. First, note that $\mathbf{g}_5$ only has 2 non-zero entries: $g_{5,1}$ and $g_{5,2}$. Apparently, $\mathbf{u}_1 \cdot \mathbf{g}_5 \equiv 0$ and $\mathbf{u}_2 \cdot \mathbf{g}_5 \equiv 0$. Taking $\mathbf{u}_1$ as an example, which is the null space vector of the subspace spanned by $\mathbf{g}_1$, $\mathbf{g}_2$ and $\mathbf{g}_3$. It is clear that $\mathbf{u}_1 \cdot \mathbf{g}_5 \equiv 0$ dictates the impossibility of recovery d_4 from d_1, d_2, d_3 and c_1. This is quite obvious from the configuration in Figure 7. Hence, for cases as such ($\mathbf{u}_i \cdot \mathbf{g} \equiv 0$), we simply skip those $\mathbf{u}_i$'s. Second, although it is very simple in this case, finding a desirable $\mathbf{g}_i$ with given $\mathbf{U}$ is *nontrivial* in general. For that, we will present an effective algorithm separately, after the description of the complete construction procedure.

Now that one $\mathbf{g}_5$ is found, we append it to $\mathbf{G}$, whose size then becomes 5×4. The next step is to add $\mathbf{g}_6$. The criterion is similar: any generator submatrix $\mathbf{G}_s$, formed by $\mathbf{g}_6$ and 3 out of $\mathbf{g}_1$ to $\mathbf{g}_5$, should be invertible. The procedure is also similar: enumerating through all sub-matrices composed of any 3 rows from $\mathbf{G}$ and computing the null space vectors. Note that the null space matrix $\mathbf{U}$ from the previous round can be reused, and thus we only need to consider additional sub-matrices formed by $\mathbf{g}_5$ and any 2 out of $\mathbf{g}_1$ to $\mathbf{g}_4$. Note that all the sub-matrices do *not* have rank 3 now. For instance, the one formed by $\mathbf{g}_1$, $\mathbf{g}_2$ and $\mathbf{g}_5$ only has rank 2. Again, examining the configuration in Figure 7, we know that there is no way to recover $\mathbf{d}_3$, $\mathbf{d}_4$ from $\mathbf{d}_1$, $\mathbf{d}_2$, $\mathbf{c}_1$ and $\mathbf{c}_2$, no matter how $\mathbf{c}_2$ is computed. Hence, if a submatrix has rank less than 3, it is simply skipped. Otherwise, its null space vector is computed and appended to $\mathbf{U}$. Once $\mathbf{U}$ is updated, a desirable $\mathbf{g}_6$ is found (using the same algorithm described separately as follows) such that $\forall\ \mathbf{u} \in \mathbf{U}$, $\mathbf{u} \cdot \mathbf{g}_6 \neq 0$. The above procedure is repeated until all $\mathbf{g}_m$'s ($5 \leq m \leq 8$) are added to $\mathbf{G}$ (the complete $\mathbf{G}$ is shown later in this section).

3) An algorithm for finding $\mathbf{g}_m$: Given $\mathbf{U}$, what is the general algorithm to find a row vector $\mathbf{g}_m$, such that $\forall\ \mathbf{u} \in \mathbf{U}$, $\mathbf{u} \cdot \mathbf{g}_m \neq 0$? The algorithm starts with a random vector $\mathbf{g}_m$ (of course, certain entries are kept constant zeros, e.g. $g_{5,3}$, $g_{5,4}$, etc.). It checks the dot product of $\mathbf{u}_1$ and $\mathbf{g}_m$. If $\mathbf{u}_1 \cdot \mathbf{g}_m \neq 0$, then keep $\mathbf{g}_m$ and move on to $\mathbf{u}_2$. The process continues until

it encounters the first null space vector $\mathbf{u}_j \in \mathbf{U}$, which makes $\mathbf{u}_j \cdot \mathbf{g}_m = 0$. As mentioned before, if $\mathbf{u}_j \cdot \mathbf{g}_m \equiv 0$ (i.e., non-zero entries of $\mathbf{g}_m$ always correspond to zero entries of $\mathbf{u}_j$), $\mathbf{u}_j$ is simply skipped. Otherwise, the algorithm *augments* $\mathbf{g}_m$ (make it $\mathbf{g'}_m$) such that the following two conditions are satisfied: 1) $\mathbf{u}_j \cdot \mathbf{g'}_m \neq 0$; and 2) all previous $\mathbf{u}$'s are *not* affected, i.e., $\mathbf{u}_i \cdot \mathbf{g'}_m \neq 0$ $(i < j)$ still hold.

The first condition can be satisfied by setting $\mathbf{g'}_m = \mathbf{g}_m + \epsilon \mathbf{u}_j$ ($\epsilon \neq 0$), as any non-zero ϵ satisfies

$$\mathbf{u}_j \cdot \mathbf{g'}_m = \mathbf{u}_j \cdot (\mathbf{g}_m + \epsilon \mathbf{u}_j) = \epsilon \mathbf{u}_j \cdot \mathbf{u}_j \neq 0.$$

Now, we simply need to tweak ϵ such that the second condition is also satisfied. Formally, this involves finding a ϵ such that

$$\forall\ \mathbf{u}_i\ (1 \leq i < j),\ \mathbf{u}_i \cdot \mathbf{g'}_m \neq 0, \tag{4}$$

which turns out to be quite straightforward. We compute all ϵ's that violate Eq(4) (call them *bad* ϵ's) and construct a set to hold them (denote as $\mathcal{E}_{bad}$). As long as we pick a ϵ out of the set $\mathcal{E}_{bad}$, it is guaranteed to satisfy the second condition. To construct $\mathcal{E}_{bad}$, simply compute all the bad ϵ_i's $(1 \leq i < j)$, where each ϵ_i satisfies $\mathbf{u}_i \cdot \mathbf{g'}_m = 0$ such that:

$$\epsilon_i = \frac{\mathbf{u}_i \cdot \mathbf{g}_m}{\mathbf{u}_i \cdot \mathbf{u}_j}.$$

To this end, as long as the number of bad ϵ_i's in $\mathcal{E}_{bad}$ (i.e., $|\mathcal{E}_{bad}|$) is less than the number of symbols in the finite field (or ring), a desirable ϵ is guaranteed to be found. In the worst case, all bad ϵ_i's happen to be unique during the final round (i.e., finding $\mathbf{g}_n$), then $|\mathcal{E}_{bad}| = \binom{n}{k-1}$ (the number of null space vectors in $\mathbf{U}$). Still, as long as the field size is greater than $\binom{n}{k-1}$, the construction of a generalized Pyramid Code is guaranteed to succeed. Note that this field size bound is *very* loose. Indeed, our empirical experience shows that, in practice, many bad ϵ_i's collide and thus the required field size turns out to be much smaller. We defer both theoretical and empirical efforts in quantifying the field size to future work.

4) The summary of the construction procedure: Using the simple example, we have described all the details in the construction of the generalized Pyramid Codes. Here, we brief summarize the entire procedure (refer to Figure 8 for the complete details).

- **Step 1:** Start with a $k \times k$ identity matrix $\mathbf{G} = \mathbf{I}_{k \times k}$ and construct an empty null space matrix $\mathbf{U}$.
- **Step 2:** Update $\mathbf{U}$. Enumerate through all sub-matrices $\mathbf{S}$ formed by any $k-1$ rows from $\mathbf{G}$. If the rank of $\mathbf{S}$ is $k-1$, compute its null space vector and append to $\mathbf{U}$. Otherwise, skip $\mathbf{S}$.
- **Step 3:** Find a $\mathbf{g}_m$ such that $\forall\ \mathbf{u} \in \mathbf{U}$, $\mathbf{u} \cdot \mathbf{g}_m \neq 0$, which adopts the previously described algorithm. Update $\mathbf{G}$ by adding $\mathbf{g}_m$ to it.
- **Step 4:** Repeat Step 2 and 3 until the entire generator matrix $\mathbf{G}$ is completed.

D. Optimality of generalized Pyramid Codes

In this section, the optimality of the generalized Pyramid Codes is shown by the following theorem.

```
1: G := I_{k×k}, U := I_{k×k}
2: for m = k + 1 : n do
3:    // g_m^{zero}: boolean array marking constant zero entries
4:    // t: index of the t^{th} entry in g_m
5:    for t = 1 : k do
6:       g_m[t] := random value in the field
7:       if g_m^{zero}[t] = true then
8:          g_m[t] := 0
9:    ug := null // ug: all dot products of u_i · g_m
10:   for j = 1 : |U|, u_j ∈ U do
11:      if u_j · g_m ≠ 0 then
12:         ug[j] := u_j · g_m, repeat
13:      if u_j · g_m ≡ 0 then
14:         ug[j] := 0, repeat
15:      // E_bad: all bad ε_i's, uu: all dot products of u_i · u_j
16:      E_bad := null, uu := null
17:      for i = 1 : j − 1 do
18:         uu[i] := u_i · u_j
19:         if uu[i] = 0 then
20:            repeat
21:         E_bad := E_bad + {ug[i]/uu[i]}
22:      ε = random value out of E_bad
23:      // argument g_m and update ug
24:      g_m := g_m + εu_j
25:      for i = 1 : j do
26:         ug[i] := ug[i] + εuu[i]
27:      for t = 1 : k, g_m^{zero}[t] = true do
28:         g_m[t] := 0
29:      ug[j] := u_j · g_m
30:      // update U and add g_m to G
31:      for S' = {k-2 rows in G} do
32:         S = S' + {g_m}
33:         if rank(S) = k − 1 then
34:            u := null space vector of S
35:            U := U + {u}
36:      G := G + {g_m}
37: return
```

Fig. 8. Construction of generalized Pyramid Codes.

Theorem 3: In the generalized Pyramid Codes, any *possible* recoverable erasure pattern (i.e., its corresponding Tanner graph contains a full-size matching) can indeed be recovered.

Proof: We prove the theorem by induction on the construction algorithm (details in Figure 8). Recall that $\mathbf{g}_1$ to $\mathbf{g}_k$ simply form a identity matrix $\mathbf{I}_{k \times k}$, so the base case is when $\mathbf{g}_{k+1}$ is added to $\mathbf{G}$. Consider an erasure pattern where one data block (say $\mathbf{d}_k$) is failed. If the block is recoverable, then the corresponding Tanner graph has a full-size matching (simply size 1, an edge between $\mathbf{d}_k$ and $\mathbf{c}_1$). In this case, the decoding equations are $\mathbf{C}_s = \mathbf{G}_s \times \mathbf{D}$, where $\mathbf{C}_s = [\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_{k-1}, \mathbf{c}_1]^T$ and $\mathbf{D} = [\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_k]^T$. Also, the generator submatrix is

$$\mathbf{G}_s = \begin{bmatrix} \mathbf{I}_{(k-1)\times(k-1)} & 0 \\ \mathbf{g}'_{k+1} & g_{k+1,k} \end{bmatrix}, \tag{5}$$

where $\mathbf{g}'_{k+1}$ is of size $1 \times (k-1)$. Based on simple matrix row elementary operations, it is easy to show that $g_{k+1,k}$ exists such that $\mathbf{G}_s$ is invertible and thus the erasure pattern is recoverable. Thus far, we have shown the existence of $\mathbf{g}_{k+1}$. Then, the algorithm ensures that once it terminates and a $\mathbf{g}_{k+1}$ is found, all generator sub-matrices as $\mathbf{G}_s$ are indeed invertible.

Now, let's assume the theorem holds after the construction algorithm adds up to $(n-1)$ $\mathbf{g}_m$'s $(1 \leq m < n)$ to $\mathbf{G}$. Next, we want to show that the theorem still holds when a new row vector $\mathbf{g}_n$ is added. Consider an arbitrary failure pattern, whose Tanner graph T contains a full-size matching. Assume the pattern contains r failed data blocks (say $\mathbf{d}_{k-r+1}$, $\cdots$, $\mathbf{d}_k$) and r available erasure blocks (say $\mathbf{c}_1$, $\cdots$, $\mathbf{c}_{r-1}$ plus $\mathbf{c}_m$). Apparently, we only need to consider patterns including $\mathbf{c}_m$. Otherwise, the erasure pattern is naturally recoverable by the induction assumption. Without loss of generality, let $\mathbf{d}_k$ be connected to $\mathbf{c}_m$ in the matching. Then, the decoding equations can be written as $\mathbf{C}_s = \mathbf{G}_s \times \mathbf{D}$, where $\mathbf{C}_s = [\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_{k-r}, \mathbf{c}_1, \mathbf{c}_2, \cdots, \mathbf{c}_{r-1}, \mathbf{c}_m]^T$ and the generator submatrix is

$$\mathbf{G}_s = \begin{bmatrix} {\mathbf{G}^1}_{(k-1)\times(k-1)} & {\mathbf{G}^2}_{(k-1)\times 1} \\ \mathbf{g}'_n & g_{n,k} \end{bmatrix}. \tag{6}$$

Here, $\mathbf{g}'_n$ of size $1 \times (k-1)$. We want to show that $g_{n,k}$ exists such that $\mathbf{G}_s$ is invertible.

Next, let's modify the erasure pattern slightly. Assume $\mathbf{d}_k$ is now available and $\mathbf{c}_m$ becomes erasure. Hence, the new erasure pattern contains $r-1$ failed data blocks and $r-1$ available redundant blocks. Apparently, its corresponding Tanner graph T' contains a full-size matching of size $r-1$ (simply removing the edge between $\mathbf{d}_k$ and $\mathbf{c}_m$ from the full-size matching in T). Based on the induction assumption, this pattern is recoverable. Again, the decoding equations can be written (with slight rearrangement to put $\mathbf{d}_k$ as the last entry) as $\mathbf{C}'_s = \mathbf{G}'_s \times \mathbf{D}$, where $\mathbf{C}'_s = [\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_{k-r}, \mathbf{c}_1, \mathbf{c}_2, \cdots, \mathbf{c}_{r-1}, \mathbf{d}_k]^T$ and differs from $\mathbf{C}_s$ only at the last entry. Moreover, the generator submatrix is

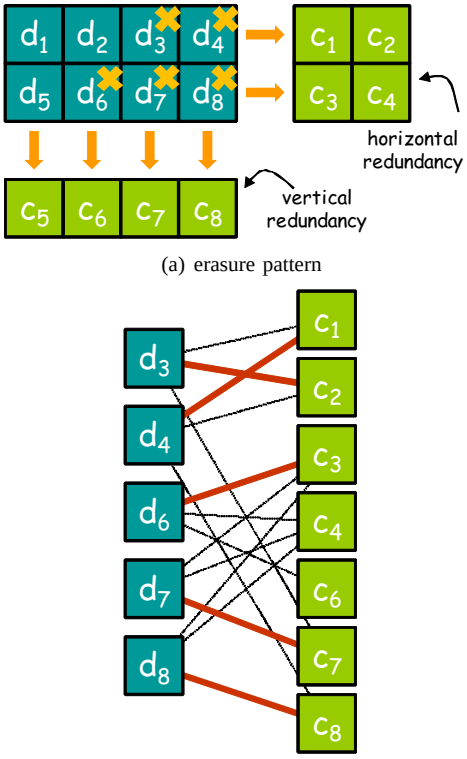
$$\mathbf{G}'_s = \begin{bmatrix} {\mathbf{G}^1}_{(k-1)\times(k-1)} & {\mathbf{G}^{2'}}_{(k-1)\times 1} \\ \{0\}_{1\times(k-1)} & 1 \end{bmatrix}, \tag{7}$$

where ${\mathbf{G}^1}_{(k-1)\times(k-1)}$ is the same $(k-1)\times(k-1)$ matrix as in Eq(6). Since the new erasure pattern is recoverable, $\mathbf{G}'_s$ is invertible, as well as ${\mathbf{G}^1}_{(k-1)\times(k-1)}$.

Examining Eq(6), it is easy to conclude that $g_{n,k}$ exists such that $\mathbf{G}_s$ is also invertible. Thus far, the existence of $\mathbf{g}_n$ is proven. Similarly, upon the termination of the construction algorithm, once a $\mathbf{g}_n$ is found, it is guaranteed that all generator sub-matrices as $\mathbf{G}_s$ are always invertible. The proof is complete. ■

E. Decoding of generalized Pyramid Codes

When block failures happen in ERC schemes, two types of recovery could be triggered: 1) recovery of all the failed blocks, including data and redundant blocks; and 2) recovery of a particular data block being actively accessed. Correspondingly, the *access overhead* can also be categorized into: 1) recovery overhead; and 2) read overhead.



(a) erasure pattern

(b) Tanner graph

Fig. 9. Decoding of generalized Pyramid Codes.

Given an erasure pattern, we define an *access path* as a sequence of blocks to be accessed in order to recover the desirable blocks. Different access paths often bear different overheads. For instance, Figure 9(a) shows a configuration of a generalized Pyramid Code, as well as an erasure pattern with 5 failed data blocks. If it is desirable to recover data block $\mathbf{d}_6$, there are at least two viable access paths: 1) recover $\mathbf{d}_6$ directly from $\mathbf{d}_2$ and $\mathbf{c}_6$; or 2) first recover $\mathbf{d}_3$ from $\mathbf{d}_1$, $\mathbf{d}_2$, $\mathbf{c}_1$ and $\mathbf{c}_2$, then recover $\mathbf{d}_7$ from $\mathbf{d}_3$ and $\mathbf{c}_7$, and finally recover $\mathbf{d}_6$ from $\mathbf{d}_5$, $\mathbf{d}_7$, $\mathbf{c}_3$ and $\mathbf{c}_4$. Apparently, these two access paths have significantly different overheads. Similarly, if it is desirable to recover all the failed data blocks, there might also be a few access paths with different overheads. In this section, we describe algorithms to find access paths with either minimum recovery overhead or minimum read overhead. This is contrast to the basic Pyramid Codes, where finding access path with minimum overhead is straightforward, because decoding should always start from the lowest level in the hierarchy and gradually move up.

1) Minimum recovery overhead:

Theorem 4: In the generalized Pyramid Codes, to recover the failed blocks in an erasure pattern with d failed data blocks and c failed redundant blocks, the minimum access path will include exactly d available redundant blocks. It will also include every available data block, from which the failed redundant blocks are originally computed.

Proof: Apparently, to recover d failed data blocks, at least d redundant blocks are needed. Hence, the minimum access path includes at least d redundant blocks. Next, we prove that

including more than d redundant blocks will *only* increase the recovery overhead.

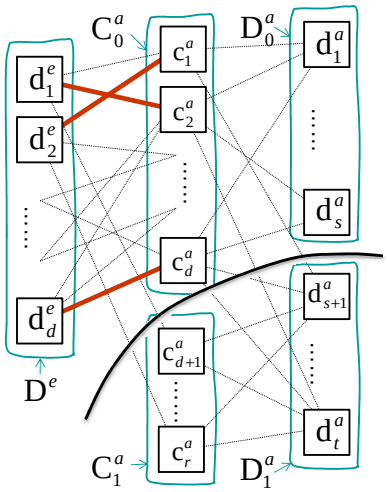


Fig. 10. Minimum recover overhead for failed data blocks.

Both failed data and redundant blocks need to be recovered. We first show that, to recover the d failed data blocks (denoted as $\mathbf{D}^e = \{\mathbf{d}_1^e, \cdots, \mathbf{d}_d^e\}$), the minimum access path should include exactly d available redundant blocks. Proving by contradiction, let's assume the minimum access path in fact includes r available redundant blocks (denoted as $\mathbf{C}^a = \{\mathbf{c}_1^a, \cdots, \mathbf{c}_r^a\}$) and $r > d$. Further, assume the minimum access path includes s additional available data blocks (denoted as $\mathbf{D}_0^a = \{\mathbf{d}_1^a, \cdots, \mathbf{d}_s^a\}$), which is shown to the right of the Tanner graph in Figure 10. Since $\mathbf{D}^e$ is recoverable, there must exist a full-size matching between $\mathbf{D}^e$ and $\mathbf{C}^a$. Without loss of generality, assume the matching connects the first d nodes in $\mathbf{C}^a$, denoted as $\mathbf{C}_0^a = \{\mathbf{c}_1^a, \cdots, \mathbf{c}_d^a\}$. Then, find another access path, which includes only redundant blocks in $\mathbf{C}_0^a$. Of course, this access path need to include additional available data blocks (denoted as $\mathbf{D}_1^a = \{\mathbf{d}_{s+1}^a, \cdots, \mathbf{d}_t^a\}$). Based on the assumption, the recovery overhead of this access path is *not* minimum (the path includes $\mathbf{C}_0^a$, $\mathbf{D}_0^a$ and $\mathbf{D}_1^a$). Hence, $d + t > r + s$ (i.e., $|\mathbf{C}_1^a| < |\mathbf{D}_1^a|$). Note that each node in $\mathbf{D}_1^a$ is connected to at least one node in $\mathbf{C}_0^a$. Since $\mathbf{D}_1^a$ is *not* included in the minimum access path, their values must have been cancelled out by $\mathbf{C}_1^a$ during decoding. For this reason, each node in $\mathbf{C}_1^a$ should connect to at least one node in $\mathbf{D}_1^a$.

Now, let's consider only nodes in $\mathbf{C}_1^a$, $\mathbf{D}_1^a$ and edges between them. We claim that there must exist a full-size matching between $\mathbf{C}_1^a$ and $\mathbf{D}_1^a$. Assuming this is *not* true, then, the maximum matching size will be less than $|\mathbf{C}_1^a|$, so as the size of the corresponding minimum node cover N_c (recall that maximum matching and minimum node cover are equivalent in the bipartite graph). Denote $\mathbf{C}_1^{a\prime}$ as those nodes in $\mathbf{C}_1^a$ while *not* in N_c, and denote $\mathbf{D}_1^{a\prime}$ as those nodes in $\mathbf{D}_1^a$ while also in N_c. Based on the property of node cover, each node in $\mathbf{C}_1^{a\prime}$ is connected to at least one node in $\mathbf{D}_1^{a\prime}$. On the other hand, $|\mathbf{C}_1^{a\prime}| > |\mathbf{D}_1^{a\prime}|$ (based on the assumption). Now that nodes in $\mathbf{C}_1^{a\prime}$ do *not* connect to other nodes in $\mathbf{D}_1^a$, at least one of them can be removed from the minimum access path without affecting the recoverability. This means the cost of the minimum access path can be further reduced, which is certainly a contradiction. Therefore, there must exist a full-size matching between $\mathbf{C}_1^a$ and $\mathbf{D}_1^a$.

Without loss of generality, assume this matching connects $\mathbf{C}_1^a$ to the first $r - d$ nodes in $\mathbf{D}_1^a$ (denote them as $\mathbf{D}_1^{a\prime\prime}$). Now, let's consider a new erasure pattern, which consists of $\mathbf{D}^e$, $\mathbf{D}_1^{a\prime\prime}$ and one more node from $\mathbf{D}_1^a$ (say $\mathbf{d}_t^a$). Using the redundant block in $\mathbf{C}^a$ ($\mathbf{C}_0^a$ and $\mathbf{C}_1^a$) and the data blocks in $\mathbf{D}_0^a$, it is clear that $\mathbf{D}^e$ can be recovered. Next, we examine the remaining Tanner graph. It contains a full-size matching, which consists of a matching of size $r - d$ between $\mathbf{C}_1^a$ and $\mathbf{D}_1^{a\prime\prime}$, together with an additional edge between $\mathbf{d}_t^a$ and at least one node in $\mathbf{C}_0^a$. Therefore, the rest failed data blocks can also be decoded. To this end, we have demonstrated a case, where $d + (r - d) + 1 = r + 1$ failed data blocks are recovered from only r redundant blocks. This creates a contradiction. Therefore, the minimum access path should include exactly d available redundant blocks.

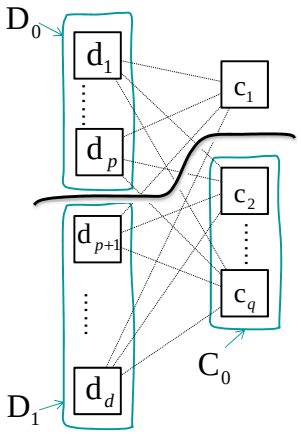


Fig. 11. Minimum recover overhead for failed redundant blocks.

In the second part of the proof, we show that to recover the failed redundant blocks, the minimum access path includes every data block, from which these redundant blocks are originally computed. In another word, no redundant block can be computed from the combination of data blocks and redundant blocks with less overhead. Using contradiction argument, let's assume this claim is *not* true on one particular redundant block $\mathbf{c}_1$. Instead of computing from d data blocks (say $\mathbf{d}_1, \cdots, \mathbf{d}_d$), assume $\mathbf{c}_1$ can instead be computed with a minimum overhead from p data blocks (denoted as $\mathbf{D}_0 = \{\mathbf{d}_1, \cdots, \mathbf{d}_p\}$) together with $(q-1)$ redundant blocks (denoted as $\mathbf{C}_0 = \{\mathbf{c}_2, \cdots, \mathbf{c}_q\}$), where $p + (q - 1) < d$ (shown in Figure 11). Under this assumption, there must exist a full-size matching between the rest $(d - p)$ data blocks (denoted as $\mathbf{D}_1$) and $\mathbf{C}_0$. (Otherwise, we can examining the corresponding minimum node cover and show that at least one node in $\mathbf{C}_0$ could be computed from $\mathbf{D}_0$ and the rest blocks in $\mathbf{C}_0$. This means $\mathbf{c}_1$ can be computed even if this node is removed from $\mathbf{C}_0$, which further implies even less overhead to compute $\mathbf{c}_1$.) Hence, the maximum matching size between $\mathbf{D}_1$ and $\mathbf{C}_0$ is $(q - 1)$, and denote the $(q - 1)$ matching nodes from

$\mathbf{D}_1$ as $\mathbf{D}_1{}' = \{\mathbf{d}_{p+1}, \cdots, \mathbf{d}_{p+q-1}\}$. Now, let's consider a particular erasure pattern of q failed data blocks, which include all the $(q-1)$ nodes in $\mathbf{D}_1{}'$ and $\mathbf{d}_d$. This erasure pattern should be recoverable using all the $(q-1)$ redundant blocks in $\mathbf{C}_0$ together with $\mathbf{c}_1$. This is because there exists a full-size matching in the corresponding Tanner graph (a matching of size $(q-1)$ between $\mathbf{D}_1{}'$ and $\mathbf{C}_0$, together with an edge between $\mathbf{d}_d$ and $\mathbf{c}_1$). On the other hand, $\mathbf{c}_1$ can be computed from $\mathbf{D}_0$ and $\mathbf{C}_0$, and thus is *not* an effective redundant block (or $\mathbf{c}_1$ is linear dependent on $\mathbf{D}_0$ and $\mathbf{C}_0$). Hence, $\mathbf{c}_1$ should be removed. To this end, it is *impossible* to recover q failed data blocks from $(q-1)$ redundant blocks. This creates a contradiction. In summary, the proof is complete with the combination of the above two parts. ■

Based on Theorem 4, it is straightforward to design a decoding algorithm with the minimum recovery overhead. Given any erasure pattern, we choose subsets of redundant blocks, such that the size of each subset simply equals to the number of failed data blocks. If the recovery can succeed (again, the corresponding Tanner graph contains a full-size matching), the recovery (data + redundant) overhead is computed. After enumerating through all the redundant subsets, the minimum recovery overhead can be readily derived (details shown in Figure 12). In practice, the number of available redundant blocks in the Tanner graph will not be many more than the number of failed data blocks, so the complexity of the algorithm should *not* be high. For instance, the Tanner graph in Figure 9(b) contains 7 redundant blocks and 5 failed data blocks, thus there are merely $\binom{7}{5} = 21$ subsets to compute.

```
1: D^e := failed data nodes
2: C^a := all available redundant nodes
3: overhead := ∞
4: for C_s^a = subsets of C^a with size |D^e| do
5:    if ∃ full-size matching between C_s^a and D^e then
6:       o_l := |C_s^a| + available data blocks connected to C_s^a
7:       o_l := c_0 + overhead to recover failed redundant blocks
8:       overhead = min(overhead, o_l)
9: return
```

Fig. 12. Calculate minimum recovery overhead.

2) Minimum read overhead:

The recovery of a single data block in general requires smaller overhead than the recovery of all failed blocks, and their respective access paths could be rather different as well. An algorithm to find an access path with the minimum read overhead is described as follows.

Similar to the algorithm in Figure 12, we choose subsets of the available redundant blocks, whose size equals to the total number of failed data blocks. If the corresponding Tanner graph does *not* contain a full-size matching, this subset is simply skipped. (For simplicity, we only care cases where all the failed data blocks can be recovered. This assumption can be easily removed, should the recovery of a subset of blocks become interesting.) Otherwise, a breadth first search is carried out, starting from the target failed data block. If the search encounters a data node in the Tanner graph, it follows only the edge in the matching to the corresponding redundant node. If the search encounters a redundant node, it follows all edges in the Tanner graph to all the data nodes, which have *not* been visited before. Let $\mathbf{D}_v$ denote the set of data nodes already visited, $\mathbf{C}_v$ the set of redundant nodes already visited, and $\mathbf{D}_c$ the set of all data nodes connected to $\mathbf{C}_v$. The search stops when $\mathbf{C}_v$ becomes large enough to recover $\mathbf{D}_v$ (i.e., $|\mathbf{D}_v| \leq |\mathbf{C}_v|$ **and** $\mathbf{D}_c \subseteq \mathbf{D}_v$). Please refer to Figure 13 for details. After enumerating through all subsets, the minimum read overhead can be easily derived. Moreover, the complexity is comparable to the algorithm in Figure 12.

Using the example shown in Figure 9(b), when a redundant subset is chosen with $\{\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3, \mathbf{c}_7, \mathbf{c}_8\}$, a full-size matching can be found. To access the failed data block $\mathbf{d}_7$, the breadth first search starts from $\mathbf{d}_7$, goes to $\mathbf{c}_7$, then $\mathbf{d}_3$, $\mathbf{c}_2$, $\mathbf{d}_4$ and stops at $\mathbf{c}_1$. It is straightforward to compute that the read overhead corresponding to this redundant subset is 5.

```
1: Q := null // queue used for the breadth first search
2: D_v := null, C_v := null, D_c := null
3: M := find a maximum matching
4: if |M| is less than failed data blocks then
5:    return
6: Q.enqueue(n_0) // n_0: the target failed data block
7: while |Q| > 0 do
8:    n := Q.dequeue
9:    if n is a data node then
10:      if n ∈ D_v then
11:         repeat
12:      D_v := D_v + {n}
13:      Q.enqueue(M[n]) // follow the edge in the matching
14:   else
15:      C_v := C_v + {n}
16:      // follow all edges to data nodes
17:      for n_d := data nodes connected to n do
18:         if n_d ∉ D_v then
19:            Q.enqueue(n_d)
20:         if n_d ∉ D_c then
21:            D_c := D_c + {n_d}
22:      if |D_v| ≤ |C_v| and D_c ⊆ D_v then
23:         last // found an access path for n_0
24: o_l := |C_v| + available data blocks connected to C_v
25: overhead := min(overhead, o_l)
26: return
```

Fig. 13. Calculate minimum read overhead (one redundant subset).

Very careful readers might challenge that given a redundant subset, there could exist more than one full-size matching in the Tanner graph (i.e., data and redundant nodes could be matched differently, while the sizes of matchings are the same). The breadth first search in Figure 13 only explores one of them, which might happen to be *not* minimum. Nevertheless, the following theorem states that the algorithm in Figure 13 can indeed find the minimum read overhead.

Theorem 5: Given an erasure pattern and a redundant subset, the algorithm in Figure 13 will always yield the same $\mathbf{C}_v$ even following different full-size matchings.

Proof: It is easy to show that $|\mathbf{D}_v| = |\mathbf{C}_v|$, when the algorithm terminates. Now, we prove the theorem by contradiction. Assume the algorithm yields with two different results (denoted as $\mathbf{D}_{v_0}$, $\mathbf{C}_{v_0}$ and $\mathbf{D}_{v_1}$, $\mathbf{C}_{v_1}$, respectively), when following two different matchings. It is clear that $\mathbf{D}_{v_0}$ and

$\mathbf{D}_{v_1}$ share at least the target data block. Then, $\mathbf{C}_{v_0}$ and $\mathbf{C}_{v_1}$ share at least one redundant block as well. Otherwise, failed data blocks in neither $\mathbf{D}_{v_0}$ nor $\mathbf{D}_{v_1}$ will *not* be recoverable, because they have to be decoded from redundant blocks not in $\mathbf{C}_{v_0}$ or $\mathbf{C}_{v_1}$, but there are less redundant blocks than failed data blocks. On the other hand, any data blocks, which are connected to the shared redundant block between $\mathbf{C}_{v_0}$ and $\mathbf{C}_{v_1}$, have to be shared by $\mathbf{D}_{v_0}$ and $\mathbf{D}_{v_1}$. Hence, following the same logic and using induction argument, we can show that $\mathbf{D}_{v_0}$ and $\mathbf{D}_{v_1}$ can *not* overlap. Then, one has to contain the other. Without loss of generality, assume $\mathbf{D}_{v_0}$ contains $\mathbf{D}_{v_1}$ (then $\mathbf{C}_{v_0}$ also contains $\mathbf{C}_{v_1}$). If that's the case, in the matching between $\mathbf{D}_{v_0}$ and $\mathbf{C}_{v_0}$, at least one node in both $\mathbf{D}_{v_0}$ and $\mathbf{D}_{v_1}$ should *not* connected to $\mathbf{C}_{v_1}$. Based on the existence of the full-size matching, at least one node in $\mathbf{C}_{v_1}$ should connect to a node in $\mathbf{D}_{v_0}$ while *not* in $\mathbf{D}_{v_1}$. This implies the algorithm would *not* have terminated with $\mathbf{D}_{v_1}$ and $\mathbf{C}_{v_1}$. Hence, it is neither possible for $\mathbf{D}_{v_0}$ to contain $\mathbf{D}_{v_1}$. In summary, this is a contradiction and the proof is complete. ■

F. Comparisons with basic Pyramid Codes

This subsection compares the generalized Pyramid Codes with the basic Pyramid Codes. We first use the same configuration (shown in Figure 5). Hence, both codes are $(18, 12)$ codes and guarantee the recovery of arbitrary 4 failures. Figure 14 compares their recoverability beyond 4 failures, as well as the recovery and read overhead. It is quite obvious that the generalized Pyramid Codes has higher recoverability when the number of failures exceeds 4. Moreover, this improvement of recoverability comes at the cost of increased read overhead (when there are 6 failures).

Next, we modify the configuration slightly and create a new generalized Pyramid Code, where the global redundant blocks are removed and replaced by $\mathbf{c}_3 : \{\mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_7, \mathbf{d}_8, \mathbf{d}_9\}$ and $\mathbf{c}_4 : \{\mathbf{d}_4, \mathbf{d}_5, \mathbf{d}_6, \mathbf{d}_{10}, \mathbf{d}_{11}, \mathbf{d}_{12}\}$. (Note that this configuration is *not* valid for a basic Pyramid Code, as there are group overlaps.) The performance of this code in also shown in Figure 14. We observe that its recoverability is slightly reduced, as it no longer guarantees the recovery of arbitrary 4 erasures. On the other hand, both its recovery and read overhead are reduced as well. Again, combined with failure probability models, it is possible to choose right configurations such that the access overhead is minimized while the reliability requirement is always satisfied.

Beyond this simple comparison, there are several other major differences between the basic and the generalized Pyramid Codes. The basic Pyramid Codes have less flexibility in code configurations, as they require all groups to be nested, i.e., the data blocks of one subgroup always form a subset of another higher hierarchy group, and two groups do not intersect with each other. The generalized Pyramid Codes, however, do *not* impose such a constraint, and two groups may overlap. On the other hand, the generalized Pyramid Codes may need a larger finite field and thus require higher computation complexity in encoding and decoding. As they are simply derived from existing codes, the basic Pyramid Codes can be constructed from well-known codes, e.g. Reed-Solomon codes, which often use rather small finite fields. Moreover, all the techniques used to speed up encoding and decoding (e.g. XOR-based array codes [7], [15], etc.) can be directly applied to the basic Pyramid Codes.

G. Additional notes

It is worth briefly comparing the generalized Pyramid Codes to some other ERC codes under the same configuration. Two examples are shown here.

In the first example, the configuration in Figure 7 is revisited, which turns out to be a simple form of *product codes* [17]. In this product code, 4 individual $(2, 1)$ codes are applied independently to the rows and the columns. *Iterative decoding* is often used to recover failures. For a particular erasure pattern, where all 4 data blocks are failed, the iterative decoding cannot succeed and the product code is declared unrecoverable. However, if the code were a generalized Pyramid Code, then, the erasure pattern is in fact recoverable (since the corresponding Tanner graph contains a full-size matching). Indeed, if we complete the construction of the earlier example, the following generator matrix is obtained:

$$\mathbf{G} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 142 & 0 & 0 \\ 0 & 0 & 244 & 71 \\ 41 & 0 & 190 & 0 \\ 0 & 108 & 0 & 174 \end{bmatrix}. \tag{8}$$

It is straightforward to verify that the generator submatrix formed by the last 4 rows of $\mathbf{G}$ is invertible, i.e., the 4 data blocks can be recovered from the 4 redundant blocks. Of course, the generalized Pyramid Codes require finite field operations, while the product code might only use XOR in each row/column. Nevertheless, the generalized Pyramid Codes *do* show higher recoverability under the same configuration.

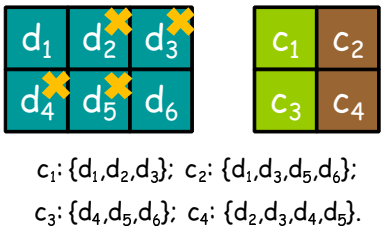


Fig. 15. Pyramid Codes vs. EVENODD Codes.

In the second example, the generalized Pyramid Codes are compared with EVENODD codes [2]. A particular configuration is shown in Figure 15, as well as an erasure pattern. It is easy to see that the erasure pattern is *not* recoverable by an EVENODD code, while in fact recoverable by a generalized Pyramid Code. Of course, the EVENODD codes were meant to protect failures of entire storage nodes (complete columns in here). Hence, the erasure pattern in Figure 15 was not considered in the original design of EVENODD codes. However, as the capacity of individual storage nodes increases, it

# of failed blocks		0	1	2	3	4	5	6
basic Pyramid Code (configuration shown in Figure 5)	recoverability (%)	100	100	100	100	100	94.12	59.32
	avg. recovery overhead	0	6.67	9.80	12	12	12	12
	avg. read overhead	1.0	1.28	1.56	1.99	2.59	3.29	3.83
generalized Pyramid Code (configuration shown in Figure 5)	recoverability (%)	100	100	100	100	100	94.19	76.44
	avg. recovery overhead	0	6.67	9.80	12	12	12	12
	avg. read overhead	1.0	1.28	1.56	1.99	2.59	3.29	4.12
generalized Pyramid Code (global redundant blocks removed)	recoverability (%)	100	100	100	100	97.94	88.57	65.63
	avg. recovery overhead	0	6.0	7.99	9.95	12	12	12
	avg. read overhead	1.0	1.28	1.56	1.87	2.32	2.93	3.85

Fig. 14. Comparisons with the basic Pyramid Codes. (The 2^{nd} generalized Pyramid Code has a different configuration from Figure 5, where the global redundant blocks are removed and replaced by $\mathbf{c}_3 : \{\mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_7, \mathbf{d}_8, \mathbf{d}_9\}$ and $\mathbf{c}_4 : \{\mathbf{d}_4, \mathbf{d}_5, \mathbf{d}_6, \mathbf{d}_{10}, \mathbf{d}_{11}, \mathbf{d}_{12}\}$.)

gradually becomes desirable to consider partial failures within a node, as suggested in [10]. In these scenarios, the generalized Pyramid Codes also show higher recoverability than existing two dimensional ERC schemes, such as [12], [14], etc.

IV. Additional related work

There are a few work, which bear a similar concept of trading storage space for access efficiency. For example, [11] can improve the read overhead by using twice as much storage spaces as the data collection itself. [21] uses slightly more storage spaces than MDS codes to improve access efficiency in wide area storage networks. Compared to these schemes, Pyramid Codes are much more flexible and can explore a much wider range of the trade-offs. Moreover, the generalized Pyramid Codes have optimal recovery performance, while none of the other existing schemes does.

There are also significant efforts in trying to improve the encoding/decoding performance of ERC schemes. In particular, lots of them advocate using pure XOR operations, such as EVENODD [2], X-Code [27], B-Code [28], RDP [5], codes based on CPM [7], [8], etc. As mentioned before, if these codes are used to derive the basic Pyramid Codes, then all optimizations direct apply. As for the generalized Pyramid Codes, some generic optimization concepts, such as [22], are also applicable.

V. Concluding remarks and open issues

In this paper, we describe two classes of Pyramid Codes, where the basic Pyramid Codes are simply derived from existing codes, while the generalized Pyramid Codes are radically advanced new codes. We also define a necessary condition of recoverability and show the generalized Pyramid Codes are optimal under the condition. Beyond presenting the main results, another intention of this paper is to inspire more research efforts in issues that still remain open. Below, we list a number of these challenges.

If we revisit the construction of the generalized Pyramid Codes, there is another interesting observation. When the simplest configuration is used (a flat configuration, where all the redundant blocks are computed from all the data blocks), then the generalized Pyramid Code essentially becomes an MDS code. Hence, the construction algorithm can also be used to find new MDS codes. On the other hand, the generalized Pyramid Codes might need larger finite fields. Indeed, our empirical experience shows that for the same block length, traditional MDS codes (e.g. Reed-Solomon codes) require much smaller finite fields. In our opinion, traditional MDS codes are constructed using more structured approaches, while the construction of the generalized Pyramid Codes carries certain random fashion. To this end, there is a very interesting and yet challenging question: is it ever possible to construct non-MDS generalized Pyramid Codes using more structured approaches, such that much smaller finite fields are required and yet the optimality is preserved?

Recent developments in applying network coding to storage applications [6] suggest the effectiveness of random linear codes in this area. From the perspective of the generalized Pyramid Codes, these work apparently move even further away from structured construction approaches and into complete randomness. Moving towards that extreme, it is also interesting to ask: given a code configuration, instead of following the generalized Pyramid Codes construction, what if we simply fill in the entries with random values? Of course, the code will *not* be optimal any more, but how much will it deviate from the optimality in terms of recoverability? Moreover, how is the access overhead affected? This appears a very interesting issue to be quantified both theoretically and empirically.

When the configuration is given, for both basic and generalized Pyramid Codes, we have described how to study the storage cost, the recoverability and the access overhead. Applying simple probability failure models, we can compare and choose among various configurations, such that certain aspects or a combination of them is optimized. We will present those results in a separate paper. However, models that capture the failures of practical large scale systems are in general far more sophisticated, especially when failures could happen correlated, as recent observations suggest [19]. Hence, the grand challenge is how to choice right configurations (or even better, adapt configurations) in practical large scale systems.

Acknowledgment

The authors would like to thanks Dr. Cha Zhang, Dr. Yunnan Wu and Dr. Philip A. Chou at Microsoft Research for very helpful and inspiring discussions on various parts during this work. In particular, Dr. Zhang helped on designing the technique to find a $\mathbf{g}_m$ given $\mathbf{U}$ in Section III. Dr. Wu helped on solving the problem to find the minimum recovery overhead of the generalized Pyramid Codes in Section III. Dr. Chou offered great insights on the open issues.

REFERENCES

[1] M. K. Aguilera, R. Janakiraman, and L. Xu, "Using erasure codes for storage in a distributed system", *International Conference on Dependable Systems and Networks (DSN 2005)*, Yokohama, Japan, June. 2005.

[2] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures," *IEEE Trans. on Computers*, 44(2), 192-202, Feb. 1995.

[3] M. Chen, C. Huang, and J. Li, "'On the Maximally Recoverable Property for Multi-Protection Group Codes", (to appear) *IEEE International Symposium on Information Theory (ISIT 2007)*, Nice, France, Jun. 2007.

[4] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "Raid – High-Performance, Reliable Secondary Storage", *ACM Computing Surveys*, 26(2), 145-185, 1994.

[5] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-Diagonal Parity for Double Disk Failure Correction", *the* 4^{th} *USENIX Conference on File and Storage Technolgoies (FAST 2005)*, San Francisco, CA, Dec. 2005.

[6] A. G. Dimakis, P. B. Godfrey, M. Wainwright, and K. Ramchandran, "Network Coding for Distributed Storage Systems", *the* 26^{th} *IEEE Conference on Computer Communications (INFOCOM 2007)*, Anchorage, AL, May. 2007.

[7] G.-L. Feng, R. H. Deng, F. Bao, and J.-C. Shen, "New Efficient MDS Array Codes for RAID Part I: Reed-Solomon-Like Codes for Tolerating Three Disk Failures", *IEEE Trans. on Computers*, 54(9), Sep. 2005.

[8] G.-L. Feng, R. H. Deng, F. Bao, and J.-C. Shen, "New Efficient MDS Array Codes for RAID Part II: Rabin-Like Codes for Tolerating Multiple (≥ 4) Disk Failures", *IEEE Trans. on Computers*, 54(12), Dec. 2005.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung,"The Google File System", *the* 19^{th} *ACM Symposium on Operating Systems Principles (SOSP 2003)*, Lake George, NY, October, 2003.

[10] J. L. Hafner, V. Deenadhayalan, KK Rao, and J. A. Tomlin, "Matrix Methods for Loss Data Reconstruction in Erasure Codes", *the* 4^{th} *USENIX Conference on File and Storage Technolgoies (FAST 2005)*, San Francisco, CA, Dec. 2005.

[11] J. L. Hafner, "WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems", *the* 4^{th} *USENIX Conference on File and Storage Technolgoies (FAST 2005)*, San Francisco, CA, Dec. 2005.

[12] J. L. Hafner, "HoVer Erasure Codes for Disk Arrays", *International Conference on Dependable Systems and Networks (DSN 2006)*, Philadelphia, PA, Jun. 2006.

[13] J. Hamilton, "An Architecture for Modular Data Centers", *Conference on Innovative Data Systems Research (CIDR 2007)*, Jan. 2007

[14] L. Hellerstein, G. A. Gibson, R. M. Karp, R. H. Katz, and D. A. Patterson, "Coding Techniques for Handling Failures in Large Disk Arrays", *Algorithmica*, vol. 12, no. 2-3, Aug. 1994

[15] C. Huang, and L. Xu, "STAR: an Efficient Coding Scheme for Correcting Triple Storage Node Failures", *the* 4^{th} *USENIX Conference on File and Storage Technolgoies (FAST 2005)*, San Francisco, CA, Dec. 2005.

[16] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: an Architecture for Global-Scale Persistent Storage", *the* 9^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Cambridge, MA, Nov., 2000.

[17] S. Lin, and D. J. Costello, "Error Control Coding, Fundamentals and Applications", Prentice Hall Press, 2004.

[18] F. J. MacWilliams, and N. J. A. Sloane, "The Theory of Error Correcting Codes, Amsterdam: North-Holland", 1977.

[19] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure Trends in a Large Disk Drive Population", *the* 5^{th} *USENIX Conference on File and Storage Technolgoies (FAST 2007)*, San Francisco, CA, Feb. 2007.

[20] J. S. Plank, "A tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems", *Software – Practice & Experience*, 27(9), 995-1012, Sep. 1997.

[21] J. S. Plank, and M. G. Thomason, "A practical analysis of low-density parity-check erasure codes for wide-area storage applications", *the International Conference on Dependable Systems and Networks (DSN 2004)*, Florence, Italy, Jun. 2004.

[22] J. S. Plank, and L. Xu, "Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications," *the* 5^{th} *IEEE International Symposium on Network Computing and Applications (NCA 2006)*, Cambridge, MA, Jul., 2006.

[23] I. S. Reed, and G. Solomon, "Polynomial Codes over Certain Finite Fields", *J. Soc. Indust. Appl. Math.*, 8(10), 300-304, 1960.

[24] A. Schrijver, "Combinatorial Optimization, Polyhedra and Efficiency", *Algorithms and Combinatorics*, Springer, vol. A, 2003.

[25] B. Schroeder, and G. A. Gibson, "Disk Failures in the Real World: What does an MTTF of 1,000,000 Hours Mean to You?", *the* 5^{th} *USENIX Conference on File and Storage Technolgoies (FAST 2007)*, San Francisco, CA, Feb. 2007.

[26] J. Stribling, E. Sit, M. F. Kaashoek, J. Li, and R. Morris, "Don't Give Up on Distributed File Systems", *International Workshop on Peer-to-Peer Systems (IPTPS 2007)*, Bellevue, WA, Feb. 2007.

[27] L. Xu, and J. Bruck, "X-code: MDS array codes with optimal encoding", *IEEE Trans. on Information Theory*, vol. 45, no. 1, 1999.

[28] L. Xu, V. Bohossian, J. Bruck, and D.Wagner, "Low Density MDS Codes and Factors of Complete Graphs", *IEEE Trans. on Information Theory*, 45(1), 1817-1826, Nov. 1999.