HTSC and FH_HTSC: XOR-based Codes to Reduce Access Latency in Distributed Storage Systems

Qiqi Shuai, *Student Member, IEEE* and Victor O.K. Li, *Fellow, IEEE*

Abstract: **A massive distributed storage system is the foundation for big data operations. Access latency performance is a key metric in distributed storage systems since it greatly impacts user experience while existing codes mainly focus on improving performance such as storage overhead and repair cost. By generating parity nodes from parity nodes, in this paper we design new XOR-based erasure codes HTSC and FH_HTSC to reduce access latency in distributed storage systems. By comparing with other popular and representative codes, we show that, under the same repair cost, HTSC and FH_HTSC codes can reduce access latency while maintaining favorable performance in other metrics. In particular, under the same repair cost, FH_HTSC can achieve lower access latency, higher or equal failure tolerance and lower computation cost compared with the representative codes while enjoying similar storage overhead. Accordingly, FH_HTSC is a superior choice for applications requiring low access latency and outstanding failure tolerance capability at the same time.**

Index Terms: **Erasure codes, access latency, storage overhead, repair cost, failure tolerance, computation cost.**

I. INTRODUCTION

IN 2003, Google described for the first time the Google file system (GFS) [1]. In GFS, three copies of each file are stored in different places to deal with potential failures. This simple replication strategy works well since it can easily support Google's frequent read requirements. Though it is easy to simply store replicated data to combat data losses, GFS suffers from extremely low storage efficiency. To solve this problem, erasure coding technique has been adopted to reduce storage overhead while maintaining high reliability in distributed storage systems. Actually, including storage overhead, there are at least five metrics to measure the performance in distributed storage systems [2], [3].

Storage Overhead. It measures the storage cost of some specific code in distributed storage systems, and is equal to the value of actual storage size over the original file size.

Failure Tolerance Capability. It measures the reliability of distributed storage systems and is usually equal to the maximum number of simultaneous failed storage nodes the system can tolerate. Note that in this paper, a node means a fragment in a codeword and actually fragments from different codeword can be stored in one storage device.

Repair Cost. It is usually defined as the number of nodes required to repair a failed node [4], [5] and can be further divided into repair I/O and repair bandwidth. Repair I/O is the number of storage nodes required to be accessed when repairing a node. Repair bandwidth is the volume of data transferred when repairing a storage node. For most codes, repair I/O and bandwidth increase or decrease at the same time, while some codes such as Regenerating Code (RC) [6] can reduce repair bandwidth by increasing repair I/O.

Computation Cost. This measures resources used in the encoding and decoding processes. Although it is arduous to use some close-form expression to measure it, it clearly has positive correlation with the complexity of the encoding and decoding processes and may have great impact on the performance of distributed storage systems. Generally, although it is almost impossible to get the exact computation cost for each code, we can compare the computation cost of different codes based on their Galois Field and the number of data nodes in a codeword. The more the number of elements in the Galois Field, the higher the computation cost [7]. We will introduce Galois Field in details later in this paper.

Access Latency. It measures the availability of storage nodes and may be reflected by the average time taken to read data from storage nodes. In this paper, we consider a typical append-only distributed storage system such as Hadoop distributed file system (HDFS) [8] and Windows Azure Storage system (WAS) [4]. In such systems, when you save something, the dispatcher can usually quickly distribute it to some applicable idle storage nodes. Actually, in an append-only system, when modifying any portion of a file that is already stored, the system will rewrite the whole file and store it in other nodes within a new codeword and then we can choose whether to delete the original version or not according to practical needs. Or we can just save the updated part to another node in another codeword. In distributed storage systems, we usually will not directly update some stored part in the original node, thus saving the trouble of updating the corresponding parity nodes. Therefore, update operations can just be regarded as write requests in such systems. In addition, users often do not care much regarding the latency of their write requests as long as they can be done within a reasonable period. But things are totally different for read requests. The access latency of read requests can greatly impact user experience. As an example, Google found that users performed up to 0.74% fewer searches after a 400 millisecond additional delay has been implemented for 4 to 6 weeks [9]. Besides, since there are only limited nodes in the system storing the data desired by certain read request, higher frequency of read requests will inevitably increase the access latency under the same conditions. When some storage node fails, the system will send a repair request, and to ensure no data loss, repair requests probably enjoy higher priority compared with read and write requests. That is, repair requests will be served before read requests and may greatly im-

Manuscript received April 30, 2015; approved for publication by Prof. Dan Wang, Guest Editor, August 12, 2015.

The authors are with the Department of Electrical and Electronic Engineering, The University of Hong Kong, Hong Kong (e-mail: {qqshuai,vli}@eee.hku.hk).

pact their access latencies. Accordingly, in this paper, we focus on the access latency of read requests and the impact of repair requests on the access latency of read requests.

For erasure codes with maximum-distance-separable (MDS) property, like the Reed-Solomon (RS) code [10], n coded symbols consist of a codeword and any k out of n coded symbols are sufficient to recover all the original file in the codeword. Compared with the three-replica strategy, it has superior storage overhead and failure tolerance capability, but worse repair cost and computation cost. Codes used in practice are usually systematic codes, namely, there is one copy of the data existing in uncoded form in a codeword and this can facilitate applications such as keyword searching [11]. The codes proposed in this paper will be systematic code in line with almost all the practical codes.

Some recent focus is on reducing repair cost. Dimakis $et\ al.$ [6] demonstrated that RC code can use network coding to lower repair bandwidth. This work spawned many papers [12]-[16]. But RC will increase repair I/O and cannot realize complete exact repair. In addition, the randomized construction of RC suffers from a high computation cost, especially when parameters are not chosen properly [17].

Duminuco $et\ al.$ [18] came up with the local repair degree concept and Hierarchical Code (HC) to reduce repair cost. But the repair cost of this code changes from 2 to k and the failure tolerance capability is hard to obtain even though all the parameters of the code are available. Huang $et\ al.$ [4] proposed the Local Reconstruction Code (LRC) and applied it to WAS. The key idea is to divide the storage nodes into different groups and generate local parity node in each group to reduce repair cost and also generate global parity node to improve failure tolerance capability. But when the global parity node fails, repair cost remains high. Sathiamoorthy $et\ al.$ [5] proposed Locally Repairable Codes (LRCs) to solve this problem, at the cost of higher storage overhead and using a deterministic algorithm with exponential complexity in the construction of code coefficients.

Flat XOR-based erasure codes usually enjoy low computation cost compared with MDS code, but many such codes, such as EVENODD [19], X-Code [20], RDP Code [21] and STAR [22], possess low failure tolerance capability (of not more than 3). Some flat XOR-based erasure codes such as WEAVER Codes [23] have outstanding failure tolerance capability but high storage overhead (of not less than 2).

Recently, based on traditional erasure codes, quite a few schemes have been proposed to reduce access latency in distributed storage systems. Huang $et\ al.$ [24] proposed the BoS algorithm and first manifested analytically that codes can reduce queueing delay. And a lot of work demonstrates that we can reduce access latency by sending redundant requests. Some of the work [25]-[27] is based on theoretical analysis, while some [28] also performs tests in trace-driven simulations. A code designed to reduce the access latency while retaining excellent performance of other metrics is still lacking. In this paper, we design HTSC and FH_HTSC which focus on reducing access latency while ensuring good performance in other metrics.

Our Contributions: In this paper, to the best of our knowledge, we are the first to propose the idea of generating parity nodes from parity nodes to increase the availability of data nodes. Based on this idea, we propose the tree-structured XOR-based erasure codes, HTSC and FH_HTSC. Under the same repair cost, HTSC not only achieves better access latency, but also lowers storage overhead and computation cost compared with other popular and representative codes. FH_HTSC is based on HTSC and compared with HTSC, under the same repair cost, it slightly increases storage overhead while maintaining the same effectiveness on reducing access latency while greatly increasing the failure tolerance capability. Compared with other representative codes, FH_HTSC is a superior choice for applications requiring high data reliability and low access latency at the same time.

The remainder of this paper is organised as follows. In Section II we propose the system model for HTSC and FH_HTSC codes and analyze their performance. In Section III we build a model to measure access latency and discuss the relationship of access latency between HTSC and FH_HTSC codes. In Section IV we compare the performance of different codes and illustrate the advantages of HTSC and FH_HTSC codes. Finally, in Section V, we conclude and discuss some open questions.

II. SYSTEM MODEL

A. Generating Parity Nodes from Parity Nodes

Existing erasure codes mainly make use of data nodes to generate the corresponding parity nodes. This can increase failure tolerance capability and the more parity nodes, the more choices we have to recover the original files. However, increasing the number of parity nodes will not only increase the storage overhead, but also increase the repair burden on data nodes since data nodes are probably accessed to help repair parity node failures. For example, in LRC [4], any global parity node failure needs all the data nodes for repair. The increased repair burden on data nodes will increase the access latency of read requests for data nodes. This is especially undesirable in those applications with frequent data retrievals, such as in Google search.

If we not only generate parity nodes with data nodes, but also generate parity nodes with parity nodes, we can transfer part of the repair burden from data nodes to parity nodes, thus reducing access latency. That is, when any parity node fails, we can just repair it with other parity nodes and thus increasing the availability of data nodes. However, generating parity nodes with parity nodes may seem to increase storage overhead. But in the latter subsections, we will manifest that, with some reasonable design, generating parity nodes with parity nodes will help reduce access latency while not increasing or even lowering the storage overhead.

Here we will propose Hierarchical Tree Structure Code (HTSC) and High Failure-tolerant Hierarchical Tree Structure Code (FH_HTSC) and study their performance on different metrics.

B. HTSC Storage System

A Galois Field [29] is an algebraic structure that contains a finite number of elements. Galois Field has the property that the results of all the operations applied to its elements still belong to this field. The XOR operation can be regarded as a linear combination of the original data nodes in a Galois Field of size 2 and can be denoted as GF(2). Generally speaking, in a Galois Field GF(2^q), elements can be expressed with q-bit words.

Proposition 1: In a storage system with linear codes, suppose $d_i,\ i = 1, 2, 3, ...$ are the original data nodes. Then the parity nodes generated are $p_i = \sum_{j=1}^{n} c_{i,j} d_j,\ where\ c_{i,j} \in GF(2^q)$.

It is obvious that the more number of elements in Galois Field, i.e., the more possible different values of c_{ij} in Proposition 1, the higher the computation cost of the code.

To facilitate understanding, we first give a simple example to illustrate our HTSC storage system. In Fig. 1, the HTSC system is constructed by a full binary tree structure with height 3. Leaf nodes at level 3 are all data nodes and level 2 contains the parity nodes generated by two data nodes at level 3. Similarly, parity nodes at level 1 are generated by parity nodes at level 2 and the parity node at level 0 is generated by parity nodes at level 1. Only blue leaf nodes are data nodes and all other brown nodes are parity nodes.

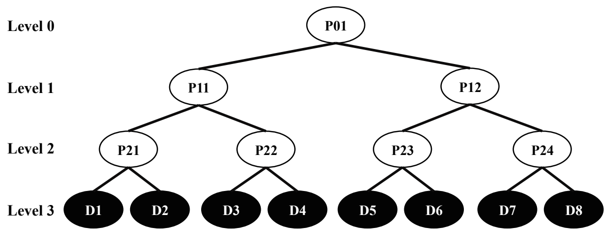

Fig. 1. A simple example: HTSC(2,3) Code.

Note that whenever a parity node fails, we can use other parity nodes instead of using data nodes to repair it, thus increasing the data nodes availability in the storage system. To construct and repair storage nodes in this system, simple XOR operation is sufficient, that means, in Proposition 1, $c_{ij} = 1\ \forall\ i, j = 1, 2, 3, ...$

In general, we use (D, h) to describe an HTSC storage system, where h stands for the height of the tree, and in the full tree, each node other than leaves has D child nodes.

There are two crucial assumptions for our HTSC(D, h) storage system: 1) in line with most of the previous work, such as [4]-[6], we suppose the failures on different nodes in the storage system are independent; 2) the tree structure is a full tree, in which any node other than leaves owns the same number of child nodes. The reason for the full tree assumption is that the full tree structure has elegant properties and more importantly, this structure can geometrically decrease the number of parity nodes, thus greatly reducing storage overhead.

When using HTSC(D, h) to store files, we usually combine different files into a fixed size M and divide them into K parts to be stored in the K data storage nodes. The general HTSC(D, h) is illustrated in Fig. 2.

Proposition 2: In HTSC(D, h), we first divide the fixed size M into $K = D^h$ pieces, each with size $\frac{M}{K}$, where K is the number of leaf nodes and the number of all nodes is $N = \frac{D^{h+1}-1}{D-1}$. At level h, the storage nodes are all original data nodes $d_i,\ i = 1, 2, 3, ..., D^h$, and at level $h-1$, the storage nodes are all parity nodes generated by original data nodes of level h. They are

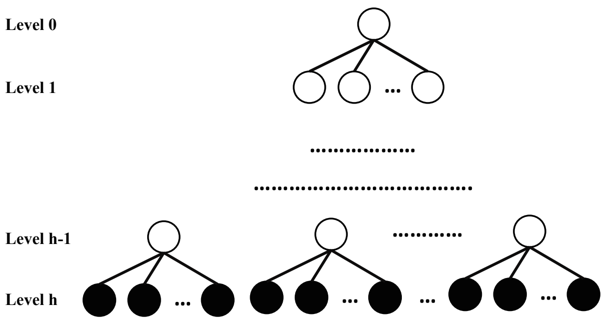

Fig. 2. A general model of HTSC(D, h) Codes.

$p_{h-1,i} = \sum_{j=1}^{D} d_{D(i-1)+j}, i = 1, 2, 3, ..., D^{h-1}$. At other levels less than $h-1$, we can similarly get the parity nodes $p_{h-l,i} = \sum_{j=1}^{D} p_{h-l+1,\ D(i-1)+j},\ \ i = 1, 2, 3, ..., D^{h-l},\ \ l = 2, 3, ..., h.$

Theorem 1: In an HTSC(D, h) storage system, the storage overhead $S = \frac{N}{K} = \frac{D^{h+1}-1}{D^h(D-1)}$, then $1 < S < \frac{D}{D-1}$, and the value of the storage overhead has negative correlation with D and positive correlation with h.

Proof: Let the storage overhead be $S(D, h) = \frac{D^{h+1}-1}{D^h(D-1)}$, then $\frac{\partial S(D,h)}{\partial h} = \frac{D^h(D-1)lnD}{D^{2h}(D-1)^2} > 0$, since $D \geq 2$, $S(D, h)$ has positive correlation with h.

Similarly, we have $\frac{\partial S(D,h)}{\partial D} = \frac{D^{h-1}(hD+D-h-D^{h+1})}{D^{2h}(D-1)^2}$, let $S_1(D, h) = hD + D - h - D^{h+1}$, then we have $\frac{\partial S_1(D,h)}{\partial D} = (h + 1)(1 - D^h) < 0$. Since $D \geq 2$, $h \geq 1$, $S_1(D, h)_{Max} = h + 1 - 2^{h+1} < 0$, therefore we have $\frac{\partial S(D,h)}{\partial D} < 0$, and storage overhead has a negative correlation with D.

Besides, we can get $\lim_{D\to\infty} S(D, h) = \lim_{D\to\infty} \frac{D^{h+1}-1}{D^h(D-1)} = \lim_{D\to\infty} \frac{1-\frac{1}{D^{h+1}}}{1-\frac{1}{D}} = 1$, as well as $\lim_{h\to\infty} S(D, h) = \lim_{h\to\infty} \frac{D^{h+1}-1}{D^h(D-1)} = \lim_{h\to\infty} \frac{1-\frac{1}{D^{h+1}}}{1-\frac{1}{D}} = \frac{D}{D-1}$, and consequently $1 < S < \frac{D}{D-1}$. □

Theorem 1 indicates that under the full tree structure and with appropriate placements of data nodes and parity nodes, the storage overhead is controlled within a reasonable range. We will compare it with other codes later.

In accordance with [4], [5], we calculate repair cost as the number of nodes required to repair a failed node.

Proposition 3: In an HTSC(D, h) storage system, for any one failure, the repair cost is $R = D$.

Actually the repair cost here stands for both repair I/O and repair bandwidth. From Proposition 3 we can see that, in the HTSC system, the repair cost for any one failure is a constant. Since HTSC incorporates the idea of LRC that divides the data nodes into small groups, the repair cost is usually much smaller than $K = D^h$. Some other hierarchical codes such as HC [18] also contains the idea of local reconstruction, but its repair cost varies from D to K. This is a big range and it is exceedingly hard to estimate the average repair cost. Accordingly, the repair cost in HTSC(D, h) is both small and constant.

In the HTSC storage system, we use parity nodes to repair the failed parity node, but use data nodes to repair only as a last resort. We achieve constant repair cost D that is much smaller than K, the number of original data nodes.

Definition 1: In an HTSC system, data loss happens whenever there is some failure in the system we cannot repair.

As illustrated in Fig. 1, if at level 3, two paired nodes (paired nodes refer to those nodes that generate parity nodes together) fail simultaneously, there will be data loss in the storage system, i.e., data failure that cannot be repaired. To realize other performance improvement, HTSC sacrifices some failure tolerance capability compared with MDS codes. In practice, the probability of one failure is much higher than the probability of more than one failure [4], [30]. Therefore, a little sacrifice of failure tolerance capability in HTSC is acceptable as long as we can control the data loss probability within a reasonable range acceptable to specific applications.

Next, we will discuss the data loss probability of the HTSC storage system in detail.

Theorem 2: In an HTSC(D, h) storage system, let p be the failure probability of a storage node, when n failures happen, the probability of data loss, $P_n(loss) = \frac{K(D-1)C_{N-2}^{n-2}}{2}p^n(1-p)^{N-n}$, where K is the number of leaves, $K = D^h$, C_N^n is the number of feasible combinations of choosing n numbers from N numbers, $n \geq 2$. $P_n(loss)$ has a positive correlation with n and a negative correlation with D and h.

Proof: In an HTSC(D, h) storage system, when any two of n failed nodes belong to the same D-data node group, there will be data loss for the system. That is, when n storage nodes fail, to study the the conditional probability of data loss, $P(loss|n)$, we can just focus on two failed nodes which belong to the same D-data node group. It is easy to get that there are D^{n-1} D-data node groups at level h. Since each D-data node group has only D nodes, the number of possibilities that the two failed nodes belong to any such group is $C_D^2 = \frac{D(D-1)}{2}$ and at the same time, the number of possibilities for the other $n-2$ failed nodes from the other $N-2$ nodes is C_{N-2}^{n-2}, so the total number of possibilities for data loss when n storage nodes fail is $D^{n-1}C_D^2C_{N-2}^{n-2}$. Since $K = D^h$, we can express it as $\frac{K(D-1)C_{N-2}^{n-2}}{2}$. It is obvious that when any n from N nodes fail, the total number of possibilities is C_N^n. Therefore, when n storage nodes fail, the conditional probability of data loss, $P(loss|n) = \frac{K(D-1)C_{N-2}^{n-2}}{2C_N^n}$. When p is the failure probability of a node in the storage system, the probability of n failures is $P(n) = C_N^np^n(1-p)^{N-n}$, so the data loss probability when n failures happen equals $P_n(loss) = P(loss|n)P(n) = \frac{K(D-1)C_{N-2}^{n-2}}{2}p^n(1-p)^{N-n}$. Although there may be some difference between the access frequency of data and parity nodes, some recent studies such as the one from Google [31] points out that temperature and activity levels are not correlated with storage node failures as previously thought and in line with almost all previous work on different kinds of codes, we also assume the same failure rate p for data and parity nodes. With mathematical induction, it is easy to prove that $P_n(loss)$ has a positive correlation with n and a negative correlation with D and h. $\square$

For instance, consider an HTSC(4,3) code. Suppose the independent failure probability of each node in the storage system p =0.1, then we can get the data loss probability of two failures as $P_2(loss) = 0.015\%$, three failures as $P_3(loss) = 0.14\%$ and four failures as $P_4(loss) = 0.64\%$. We can see that in this storage system, although we can only ensure 100% no loss when one failure happens, the probability of data loss when more than one failure happens is actually rather small. Actually, in distributed storage systems, as in the Facebook warehouse cluster, single failure recovery usually accounts for more than 98% storage repair [30]. That is, the data loss probability can be even lower in practice.

In HTSC(D, h), to reduce the probability of data loss, we need to use a structure with large D and h, while at the same time, we need to consider jointly the repair cost and storage overhead.

For the same h, when D increases, the storage overhead and the probability of data loss will decrease while the repair cost will increase, and vice versa. Similarly, for the same D, when h increases, the storage overhead will increase but the probability of data loss will decrease, and at the same time, the repair cost remains the same, and vice versa. That is, in this HTSC(D, h) storage system, we cannot get the optimal condition of storage overhead, data loss probability and repair cost at the same time by adjusting parameters D and h, but we can flexibly change D and h to achieve different tradeoffs to meet the performance requirements for different applications.

C. FH_HTSC Storage System

Although the probability of data loss is extremely small in HTSC storage system, for some applications, its failure tolerance capability may be insufficient. Hence, we modify HTSC to get higher failure-tolerance capability, and create an FH_HTSC system.

For general XOR-based erasure codes, to improve the failure tolerance capability, we have to increase the number of links between data nodes and parity nodes [32]. Theoretically, the number of arbitrary failures tolerable is decided by the Hamming distance d and exactly speaking, it equals $d-1$. That is, when any two data nodes do not own more than one shared parity node, the number of arbitrary failures it can tolerate depends on how many parity nodes a data node is linked with. But as discussed in [32], to increase the failure tolerance capability, the repair cost for each node will greatly increase. Considering the special structure of HTSC, we propose FH_HTSC, which can achieve higher failure tolerance capability compared with HTSC.

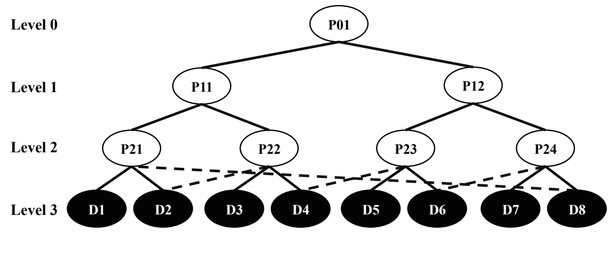

Fig. 3. A simple FH_HTSC(2,3) Code.

FH_HTSC(2,3) shown in Fig. 3 can tolerate any two failures, and it is created by just adding the four dotted links to HTSC(2,3) in Fig. 1. It is obvious that there is no change in the storage overhead. The repair cost for the whole system can be computed as two cases. For data nodes at level 3, the repair cost increases from 2 to 3 while the repair cost of parity nodes remains unchanged at 2. For Stepped Combination codes [32], to guarantee the tolerance of any two failures, each data node need to connect at least two parity nodes as illustrated in Fig. 4.

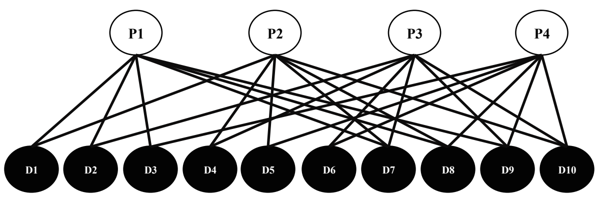

Fig. 4. A two-node failure-tolerant Stepped Combination codes.

We can see that, in Fig. 4, since each data node has to be connected with at least two different parity nodes, the links for each parity node will greatly increase by more than 100%, and thus the repair cost for each data node will also greatly increase.

While for FH_HTSC(2,3) in Fig. 3, at level 3, for each two-node group, the first data node only connects one parity node, thus reducing the number of links for each parity node at level 2. If each data node has to connect two different parity nodes, the repair cost for each data node will be 4 rather than 3. To tolerate any two failures, FH_HTSC only increase the repair cost by 50%, thus indicating the advantage of FH_HTSC system compared with Stepped Combination codes [32], in terms of repair cost.

To construct FH_HTSC(D, h) with the capability to tolerate more than two failures simultaneously, we have at least two choices. The first way is to increase the number of links of the data nodes that have already connected with two parity nodes to three or more. This is similar to the way in [32], but [32] will greatly increase the repair cost for each data node. The other way is to exploit the special structure for FH_HTSC(D, h), and to connect the first data node in each D-data node group with two, rather than one, different parity nodes. This is almost the same structure as the two-failure tolerant Stepped Combination codes [32], but we will prove that in this structure, we can tolerate any four rather than two failures. Theoretically, there are many ways to construct the four-failure tolerant FH_HTSC(D, h) storage system that contains different links between the data nodes and parity nodes at level $h-1$. To facilitate the understanding of the logical structure, we give a simple way to construct it in Proposition 4, and the result is illustrated in Fig. 5.

Proposition 4: To construct an FH_HTSC(D, h) storage system which can tolerate any four-node failures, we first divide the fixed size M into $K = D^h$ pieces, each with size $\frac{M}{K}$, where K is the number of leaf nodes and the number of all nodes is $N = \frac{D^{h+1}-1}{D-1}$. At level h, the storage nodes are all original data nodes d_i, $i = 1, 2, 3, ..., D^h$, and at level $h-1$, the storage nodes are all parity nodes generated by original data nodes of level h. They are $p_{h-1,i} = \sum\limits_{j=1}^{D} d_{D(i-1)+j} + d_{D(i-1)-j(D-1)+D^h\{1-u[D(i-1)-j(D-1)]\}}, i = 1, 2, 3, ..., D^{h-1}$, where $u(t) = 1$, when $t > 0$, and $u(t) = 0$, when $t \le 0$. At other levels less than $h-1$, we can similarly get the parity nodes $p_{h-l,i} = \sum\limits_{j=1}^{D} p_{h-l+1,\, D(i-1)+j}$, $i = 1, 2, 3, ..., D^{h-l}$, $l = 2, 3, ..., h$.

Theorem 3: The necessary and sufficient condition for four-failure tolerance FH_HTSC(D, h) constructed as in Proposition 4 is $h = 3, D \ge 3$, or $h \ge 4, D \ge 2$.

Proof: For the construction in Proposition 4, to make sure each data node can connect with two different parity nodes and any two data nodes share no more than one common parity node, we need to meet the condition $C^2_{D^{h-1}} \ge D^h$, that is, we have $D^{h-1} - 2D - 1 \ge 0$. If $h \le 2$, $D^{h-1} - 2D - 1 \le D - 2D - 1 = -D - 1 < 0$. Similarly, $D^{h-1} - 2D - 1 = D^2 - 2D - 1$ when $h = 3$, so $D^2 - 2D - 1 = (D-1)^2 - 2 \ge 0$=$(D-1)^2 - 2 \ge 0$ when $D \ge 3$. Then if $h \ge 4$, we can get $D^{h-1} - 2D - 1 \ge DD^2 - 2D - 1 \ge 2D^2 - 2D - 1 = (D-1)^2 + D^2 - 2 \ge 0$, when $D \ge 2$.

For the proof that $h = 3$, when $D \ge 3$, or $h \ge 4$, when $D \ge 2$ is a sufficient condition for the four-failure tolerance system, we can refer to the proof of Theorem 5. □

The repair cost of four-failure tolerant FH_HTSC storage system when any one failure happens in the system are stated in Theorem 4.

Theorem 4: For four-failure tolerant FH_HTSC(D, h) storage system constructed as in Proposition 4, any parity node repair cost is D, and the repair cost for any data node is $2D$.

Proof: In an FH_HTSC(D, h) storage system, repair cost for each parity node is the same as the HTSC(D, h) storage system, that is, D. Repair cost for any data node depends on the number of links each parity node at level $h-1$ has, connecting to different data nodes.

When a four-failure tolerance FH_HTSC(D, h) storage system constructed as in Proposition 4 meeting the condition in Theorem 3, there will be $2D^h$ links from data nodes to all the D^{h-1} parity nodes at level $h-1$. Accordingly, each parity node owns $\frac{2D^h}{D^{h-1}} = 2D$ links to the data nodes. So, the repair cost for each data node is $2D$. □

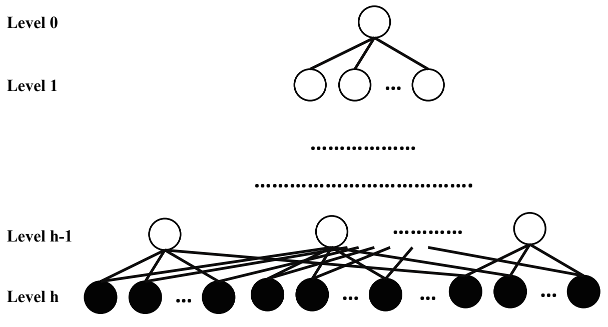

Fig. 5. General four-node failure-tolerant FH_HTSC(D, h) Code.

Theorem 5: An FH_HTSC(D, h) storage system constructed as in Proposition 4 and meeting the condition of Theorem 3 can tolerate any four-node failures simultaneously.

Proof: We construct the equations as below,

$$M=\begin{pmatrix} 1 & 1 & ... & 1 & 0 & 0 & ... & 0 & ...... & 1 & 0 & ... & 0 \\ 1 & 0 & ... & 0 & 1 & 1 & ... & 1 & ...... & 0 & 1 & ... & 0 \\ \vdots & \vdots & ... & \vdots & \vdots & \vdots & ... & \vdots & ...... & \vdots & \vdots & ... & \vdots \\ 0 & 0 & ... & 0 & 0 & 0 & ... & 0 & ...... & 1 & 1 & ... & 1 \end{pmatrix}$$

$$d=\begin{pmatrix} d_{11} \\ d_{12} \\ \vdots \\ d_{1D} \\ d_{21} \\ d_{22} \\ \vdots \\ d_{2D} \\ \vdots \\ d_{D^{h-1}1} \\ d_{D^{h-1}2} \\ \vdots \\ d_{D^{h-1}D} \end{pmatrix} \quad P=\begin{pmatrix} P_1 \\ P_2 \\ \vdots \\ P_D \\ \vdots \\ P_{D^{h-1}} \end{pmatrix}$$

Then we have

$$Md = P$$

where M is a $D^{h-1} \times D^h$ matrix, d is $D^h \times 1$ and P is $D^{h-1} \times 1$. As in Proposition 4, each data node connected with two different parity nodes and any two data nodes cannot share more than one common parity node, and hence in matrix M, each column has exactly two ones and each column vector must be independent of each other.

We first consider four failures all occurring on data nodes.

When any D^{h-1} data nodes fail at the same time, we can rewrite these failed storage nodes in the first D^{h-1} rows in vector d, and also adjust the relevant column vectors in M to the first D^{h-1} columns, then we can rewrite the equation as below.

$$\begin{pmatrix} M_a & M_b \end{pmatrix} \begin{pmatrix} d_a \\ d_b \end{pmatrix} = P$$

The D^{h-1} failed storage nodes are all in d_a and the relevant coefficients in M for the failed storage nodes are adjusted to M_a. So M_a is $D^{h-1} \times D^{h-1}$, M_b is $D^{h-1} \times D^{h-1}(D-1)$, d_a is $D^{h-1} \times 1$ and d_b is $D^{h-1}(D-1) \times 1$. If M_a is full rank and its rank equals D^{h-1}, then the inverse of M_a exists, we call it ${M_a}^{-1}$. Consequently we have $d_a = M_a^{-1}(P - M_b d_b)$. This equation has unique solution and we can repair all the D^{h-1} failed storage nodes with it.

But in fact, although in the original M, each column has exactly two ones, the row rank of M_a is not necessarily equal to D^{h-1}. We know that, if we can repair n failed data nodes, there must be at least n corresponding parity nodes that can help. That is, n failed data nodes correspond to n columns in the matrix M, and only when the row rank of the n columns is equal to or greater than n, can we repair the n failed nodes. Since $C_3^2 = 3$, when $n \le 4$, the row rank of the n columns must be equal to or greater than n, which ensures that we can repair the n failed data nodes. But when $n > 4$, the row rank of the n columns may be less than n, and we cannot guarantee the repair of all the n failed nodes.

Next, we consider the case in which some of the four failed nodes are parity nodes. Since in the FH_HTSC(D, h) and HTSC(D, h) structure, parity nodes can be repaired not only by data nodes but also by other parity nodes, it is easy to check that under this condition, we can also repair all the four failed nodes no matter they are parity nodes or data nodes. □

For the structure in Proposition 4, we just connect each data node with two different parity nodes at level $h - 1$. We can further consider the failure tolerance capability and repair cost if we connect each data node with more than two parity data nodes.

Theorem 6: In a general FH_HTSC(D, h) storage system, if we connect each data node at level h with n different parity nodes and any two data nodes have at least one different parity node to connect with them respectively, then the system can tolerate any $n + 2$ node failures simultaneously and the repair cost for each data node is nD and the repair cost for each parity node is D.

Proof: Similar to the proof of Theorem 5, we can construct the corresponding matrix equation $Md = P$. The difference is that each column of M has n ones instead of two ones. Similarly, when $C_N^n = N$, we can get $N = n + 1$ ($n = 1$ here is an invalid solution and we ignore it). Therefore, the system can tolerate any $n + 2$ data node failures simultaneously. Similar to the proof of Theorem 5, we can easily get that the system can tolerate any $n + 2$ node failures simultaneously.

For each data node, since there are n links to parity nodes at level $h - 1$, there will be nD^h links between data nodes at level h and parity nodes at level $h - 1$. That is, for each parity node at level $h - 1$, there will be $\frac{nD^h}{D^{h-1}} = nD$ links to data node at level h. Accordingly the repair cost for each data node at level h is nD. □

Of course, not all FH_HTSC(D, h) storage systems is capable of tolerating any $n + 2$ node failures. Some conditions similar to Theorem 3 must be met. Here in Theorem 6, we just give the expression of potential higher failure tolerance capability.

III. MODEL TO MEASURE ACCESS LATENCY

When using HTSC(D, h) or FH_HTSC(D, h) to store files, we usually combine different files into a fixed size M, say 1 to 3 GB and divide them into K parts to be stored in K data storage nodes. In practice, we can decide the fixed size M according to the capacity of each storage node and the parameters of HTSC(D, h) or FH_HTSC(D, h). It is obvious that the size of M is extremely big for most users, and most of the time, users only desire part of the file stored in one of the K nodes, say, the systematic part in uncoded form. This is a big difference from much of previous work since they usually assume that each read request desires all the data in a codeword, namely, all the data in the K nodes. However, this hardly captures the reality. As an example, in WAS, only when a file reaches a certain size (e.g., 3GB), will it be a candidate for erasure coding [4]. While clearly, 3GB is exceedingly big for most users and they only desire part of them most of the time. This is consistent with the design of HTSC(D, h). Accordingly, in this paper, we focus on read requests from users who only desire part of the data stored in one of the K data nodes.

As described in [11], other than write requests, there are three kind of requests for any storage node, namely, read requests, degraded reads and repair requests. Since degraded reads result from read requests on other storage nodes, we can merge degraded reads and read requests into general read requests.

In a storage system, suppose read requests arrival rate for any storage node is λ_1'. A fraction x of read requests are direct reads and $1-x$ of them are degraded reads. Any one storage node is connected to n nodes and p is the probability of that storage node joining the reconstruction of the n nodes. We can get the general read requests arrival rate for that node as $\lambda_1 = x\lambda_1' + (1-x)np\lambda_1'$ [11]. Choose the original repair requests arrival rate λ_2' of one code as unit value. Combining n and p, we can get the repair requests arrival rate λ_2 of each code to be compared [11]. μ_1 and μ_2 are the service rates of general read requests and repair requests, respectively.

We model the general read and repair requests as a head-of-the-line (HOL) priority queueing [33] system. Since we try our best to guarantee no data loss in distributed storage systems, repair requests should enjoy higher priority than general read requests. Therefore, repair requests invariably queue in front of read requests. But in their respective groups, requests follow the rule of first-come-first-served.

Theorem 7: Access latency of four-node failure-tolerant FH_HTSC(D,h) equals the access latency of HTSC$(2D,h)$, where $h=3,\, D\geq 3,$ or $h\geq 4,\, D\geq 2$.

Proof: From [11], for the same h, FH_HTSC(D,h) and HTSC$(2D,h)$ share the same repair cost $2D$ and parameter h is uncorrelated with repair cost. For an arbitrary storage node in HTSC$(2D,h)$, the number of nodes connected to it is $2D$ and in FH_HTSC(D,h), the number of nodes connected to such a node is $4D$. The general read requests arrival rate of HTSC$(2D,h)$ is $\lambda_{1\_HTSC(2D,h)} = x\lambda_1' + (1-x)2D\lambda_1'\frac{2D-1}{2D} = x\lambda_1' + (1-x)(2D-1)\lambda_1'$ and $\lambda_{1\_FH\_HTSC(D,h)} = x\lambda_1' + (1-x)4D\lambda_1'(\frac{4D-2}{4D}\frac{1}{2} + \frac{2}{4D}0) = x\lambda_1' + (1-x)(2D-1)\lambda_1'$. That is, $\lambda_{1\_HTSC(2D,h)} = \lambda_{1\_FH\_HTSC(D,h)}$.

It is obvious that the practical repair requests arrival rate for any storage node is in proportion to the number of nodes it is connected to. Choose HTSC$(2D,h)$'s original repair requests arrival rate λ_2' as unit value and we adjust the repair requests arrival rate of FH_HTSC(D,h) as $2\lambda_2'$ according to the number of nodes each node is connected to. For a data node in HTSC$(2D,h)$, it will potentially help repair $2D-1$ out of $2D$ nodes connected to it because one parity node will be repaired just by parity nodes. We adjust repair requests arrival rate of HTSC$(2D,h)$ to $\frac{2D-1}{2D}\lambda_2'$. For one data node in FH_HTSC(D,h), it is connected with $4D-2$ other data nodes but may potentially repair them with probability 0.5, and the other two parity nodes will just be repaired by parity nodes. Hence we adjust the repair requests arrival rate as $(\frac{4D-2}{4D}\cdot 0.5 + \frac{2}{4D}\cdot 0)\cdot 2\lambda_2' = \frac{2D-1}{2D}\lambda_2'$. That is, $\lambda_{2\_HTSC(2D,h)} = \lambda_{2\_FH\_HTSC(D,h)}$.

Since FH_HTSC(D,h) and HTSC$(2D,h)$ are both flat XOR-based erasure codes and possess the same coding complexity, they enjoy the same service rate. Therefore, they have the same access latency.

$h=3,\, D\geq 3,$ or $h\geq 4,\, D\geq 2$ is a sufficient and necessary condition for the four-failure tolerance of FH_HTSC(D,h), as demonstrated in Theorem 3. □

IV. COMPARISON RESULTS AND ANALYSIS

In this section, we model the read requests to each storage node as single-server $M/M/1$ queues, and part of them are transferred to other nodes as degraded read requests. There are also repair requests with highest priority in the system. Accordingly, with different parameters, we can get the access latency of different codes for users' read requests. Note that we focus on reducing access latency of read requests, and we also consider degraded reads and repair requests since they have great impact on the latency performance of read requests.

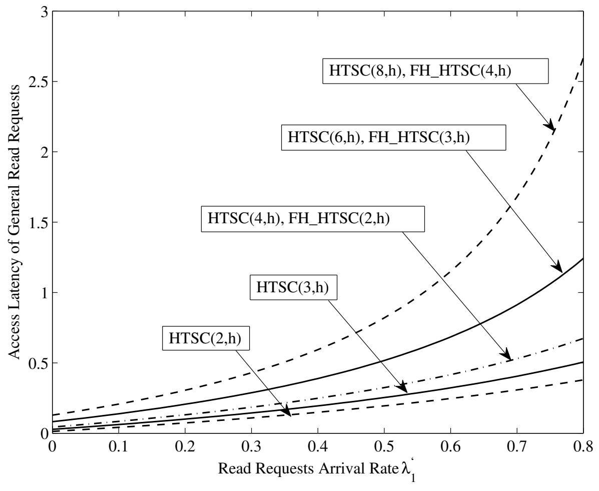

Fig. 6. Access latency comparison between HTSC(D,h) and FH_HTSC(D,h).

First, we compare the access latency of HTSC(D,h) and FH_HTSC(D,h) as illustrated in Fig. 6. We set $\mu_1 = 2$, $\mu_2 = 2$ and $x = 0.9$ for all the codes and set the original repair requests arrival rate of HTSC$(2,h)$, $\lambda_2' = 0.1$, as unit value. As in [11], we can work out the relative values of other codes. Fig. 6 indicates that when D is smaller, access latencies of both HTSC(D,h) and FH_HTSC(D,h) are lower. In addition, the access latencies of FH_HTSC$(2,h)$ and HTSC$(4,h)$, FH_HTSC$(3,h)$ and HTSC$(6,h)$, FH_HTSC$(4,h)$ and HTSC$(8,h)$ are, respectively, the same, which validates the results in Theorem 7.

Then, we compare the access latencies of HTSC and FH_HTSC codes with different erasure codes. Combined with other performance metrics, we will display the advantages of HTSC and FH_HTSC codes over other codes. Many erasure codes have been proposed for distributed storage systems and we choose some representative ones to compare. Till now, to the best of our knowledge, only RS, LRC and LRCs (LRC and LRCs are different codes) are widely applied such as by Facebook, Google, Microsoft and so on. RS can greatly reduce storage overhead and improve failure tolerance while suffering from relatively high repair cost. Compared with RS, LRC and LRCs can reduce repair cost while LRCs suffers relatively high computation cost. To put it simply, we believe RS, LRC and LRCs are representative codes since they have representative properties and they are all widely applied.

Theoretically, each code has an infinite number of parameter sets and it is impossible to compare each set of parameters between different codes. Besides, the fact that different sets of parameters for each code just achieve different tradeoffs of different performance metrics also suggest it is meaningless to compare all of them. So, we choose the sets of parameters that are used in practice by different companies for each code to compare with HTSC and FH_HTSC codes. Their wide applications in those big companies definitely show that those sets of parameters can achieve good or at least acceptable performance of different metrics and this will make sure that our comparisons are meaningful.

Hence in this section, we will compare the performance of LRC(12,2,2) (used in WAS [4]), RS(6,3) (used in GFS II [34], [35]), RS(10,4) (used in HDFS-RAID in Facebook [36]) and LRCs(10,6,5) (used in HDFS-Xorbas [5]) with that of HTSC and FH_HTSC for different metrics. Moreover, to make the results more meaningful, we divide the comparison results into three parts such that in each part, the repair cost of HTSC and FH_HTSC is the same or almost the same as the widely used codes.

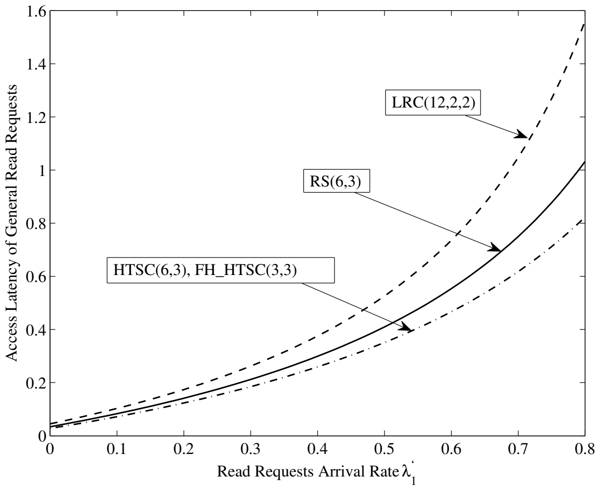

Fig. 7. Access latency comparison of different erasure codes with repair cost equal to 6.

For Fig. 7, 8 and 9, we set $\mu_1 = 2$, $\mu_2 = 2$ and $x = 0.9$ for all the codes. The access latency ranks in all tables are included to facilitate comparisons. Since the real latency value is related with the value of the read requests arrival rate, all latency values are shown in Fig. 7, 8 and 9.

Fig. 7 compares the access latency of LRC(12,2,2), RS(6,3), HTSC(6,3) and FH_HTSC(3,3) and we set the $\lambda_2' = 0.1$ of HTSC(6,3) as unit value. Table 1 compares the five performance metrics of all those codes. From Fig. 7 and Table 1, we can see that, under the same repair cost, HTSC(5,3) has the best access latency, storage overhead and computation cost, while suffering from a low failure tolerance capability; FH_HTSC(3,3) is superior to RS(6,3) in each performance metric and also outperforms LRC(12,2,2) in all performance metrics except for storage overhead.

Since obviously the number of elements in Galois Field of LRC(12,2,2), RS(6,3) is more than that of HTSC(6,3) and FH_HTSC(3,3), so LRC(12,2,2), RS(6,3) have higher computation cost. Moreover, the global parity nodes of LRC(12,2,2) need to connect 12 data nodes rather than 6 data nodes in RS(6,3). Overall, it is obvious that LRC(12,2,2) suffers from higher computation cost compared with RS(6,3). Since HTSC(6,3) and FH_HTSC(3,3) have the same Galois Field and the same number of data nodes in a local codeword, they have the same computation cost. Similarly, we can get the computation cost ranks in the other two tables.

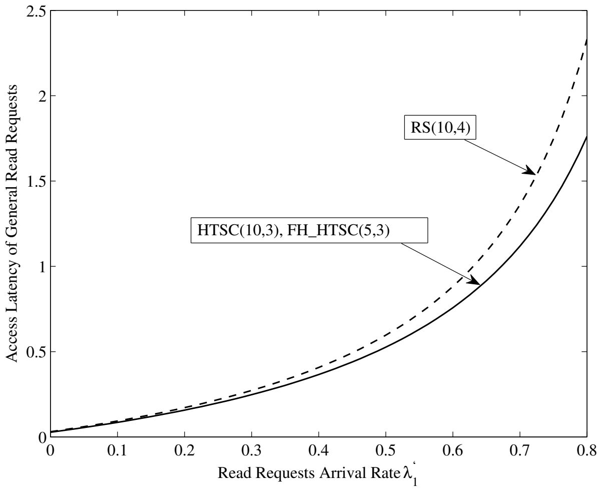

Fig. 8. Access latency comparison of different erasure codes with repair cost equal to 10.

Fig. 8 compares the access latency of RS(10,4), HTSC(10,3) and FH_HTSC(5,3) and we set $\lambda_2' = 0.1$ of HTSC(10,3) as unit value. Table 2 compares the five performance metrics of all those codes. From Fig. 8 and Table 2, similarly, under the same repair cost, HTSC(10,3) has the best access latency, storage overhead and computation cost performance, while suffering from a low failure tolerance capability; FH_HTSC(5,3) has the same failure tolerance capability while outperforming RS(10,4) in all other performance metrics.

Fig. 9 compares the access latency of LRCs(10,6,5), HTSC(5,2) and FH_HTSC(2,4) and we set $\lambda_2' = 0.1$ of FH_HTSC(2,4) as unit value. Table 3 compares the five performance metrics of all those codes. From Fig. 9 and Table 3, we can see that, with the same repair cost, HTSC(5,2) is superior to LRCs(10,6,5) in access latency, storage overhead and computation cost, while suffering from a lower failure tolerance capability. FH_HTSC(2,4) outperforms LRCs(10,6,5) in all performance metrics except for storage overhead. The main reason for the higher storage overhead of FH_HTSC(2,4) is that its repair cost is a little lower than the other two codes. From Table 1 we can find that when the repair cost of FH_HTSC is 6, it can realize much lower storage overhead.

Table 1. Performance Comparison of different erasure codes with repair cost equal to 6

Code\Metrics	Repair Cost	Access Latency Rank	Storage Overhead	Failure Tolerance	Computation Cost Rank
LRC(12,2,2)	6	3rd	1.33	Any 3	3rd
RS(6,3)	6	2nd	1.5	Any 3	2nd
HTSC(6,3)	6	1st	1.2	Any 1	1st
FH_HTSC(3,3)	6	1st	1.48	Any 4	1st

Table 2. Performance Comparison of different erasure codes with repair cost equal to 10

Code\Metrics	Repair Cost	Access Latency Rank	Storage Overhead	Failure Tolerance	Computation Cost Rank
RS(10,4)	10	2nd	1.4	Any 4	2nd
HTSC(10,3)	10	1st	1.11	Any 1	1st
FH_HTSC(5,3)	10	1st	1.25	Any 4	1st

Table 3. Performance Comparison of different erasure codes with repair cost almost equal to 5

Code\Metrics	Repair Cost	Access Latency Rank	Storage Overhead	Failure Tolerance	Computation Cost Rank
LRC(10,6,5)	5	2nd	1.6	Any 4	2nd
HTSC(5,2)	5	1st	1.24	Any 1	1st
FH_HTSC(2,4)	4	1st	1.94	Any 4	1st
Note: Since the special structure of FH_HTSC, its repair cost can only be even number.					

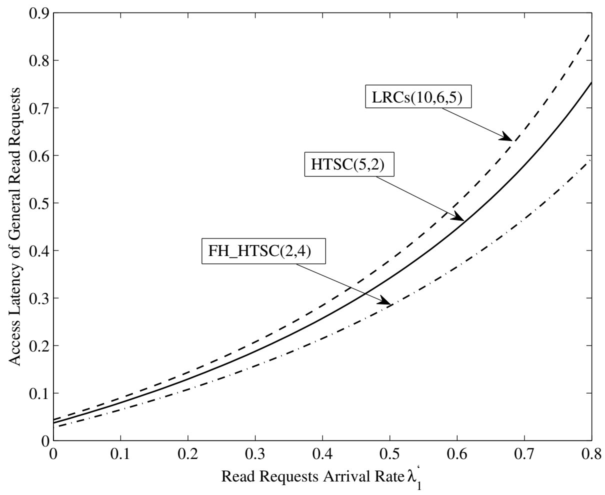

Fig. 9. Access latency comparison of different erasure codes with repair cost equal to 5.

V. CONCLUSION AND FUTURE WORK

In this paper, we design flat XOR-based erasure codes HTSC(D, h) and FH_HTSC(D, h) to reduce access latency while maintaining outstanding performance for other metrics in distributed storage systems. We compare them with other representative codes and find that other than reducing access latency, under the same repair cost, they also realize superior performance on other metrics, such as computation cost and storage overhead. In particular, FH_HTSC is a superior choice for applications requiring high data reliability and low access latency at the same time.

There are still some open questions. For instance, along with most of the previous work, models in this paper are based on the independent failure model and assume that storage nodes are located at different sites. As [35], [37] mentioned, in fact, it is hard to avoid dependent failures in practical applications and such dependencies may impact system reliability and availability. In the future, we will study the influence of storage node placement and also analyze our HTSC and FH_HTSC codes under the circumstances of dependent failures.

ACKNOWLEDGMENT

This research is supported in part by the Strategic Research Theme on Computation and Information, The University of Hong Kong.

REFERENCES

[1] S. Ghemawat, H. Gobioff, and S. T. Leung, “The Google file system,” *ACM SIGOPS Operating Systems Review*, vol. 37, pp. 29–43, 1997.
[2] J. Li and B. Li, “Erasure coding for cloud storage systems: A survey,” *Tsinghua Science and Technology*, vol. 18, pp. 259–272, 2013.
[3] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh, “A survey on network codes for distributed storage,” *Proceedings of the IEEE*, vol. 99, pp. 476–489, 2011.
[4] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, “Erasure Coding in Windows Azure Storage,” in *Proc. USENIX ATC*, (Boston, USA), 2012, pp. 15–26.
[5] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, “Xoring elephants: Novel erasure codes for big data,” in *Proceedings of the 39th international conference on Very Large Data Bases*, (Trento, Italy), 2013, pp. 325–336.
[6] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, “Network coding for distributed storage systems,” *IEEE Transactions on Information Theory*, vol. 56, pp. 4539–4551, 2010.
[7] A. Rudra, P. K. Dubey, C. S. Jutla, V. Kumar, J. R. Rao, and P. Rohatgi, “Efficient Rijndael encryption implementation with composite field arithmetic,” *Cryptographic Hardware and Embedded Systems-CHES, Springer*, pp. 171–184, 2001.
[8] M. Foley, “High availability HDFS,” in *28th IEEE Conference on Massive Data Storage*, (Asilomar Conference Grounds Pacific Grove, USA), vol. 12, 2012.
[9] J. Brutlag, “Speed matters for Google web search,” *Google*, June, 2009.
[10] S. B. Wicker and V. K. Bhargava, “Reed-Solomon codes and their applications,” *John Wiley & Sons*, 1999.
[11] Q. Shuai, V. O. K. Li, and Y. Zhu, “Performance models of access latency in cloud storage systems,” in *Fourth Workshop on Architectures and Systems for Big Data*, (Minneapolis, USA), June, 2014.
[12] K. V. Rashmi, N. B. Shah, and P. V. Kumar, “Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a productmatrix construction,” in *IEEE Transactions on Information Theory*, vol. 57, pp. 5227–5239, 2011.

[13] V. R. Cadambe, S. A. Jafar, H. Maleki, K. Ramchandran, and C. Suh, "Asymptotic interference alignment for optimal repair of MDS codes in distributed data storage," *IEEE Transactions on Information Theory*, vol. 59, pp. 2974–2987, 2013.

[14] N. B. Shah, K. Rashmi, P. V. Kumar, and K. Ramchandran, "Explicit codes minimizing repair bandwidth for distributed storage," *Information Theory Workshop*, (Cairo, Egypt), 2010, pp. 1–5.

[15] V. R. Cadambe, S. A. Jafar, and H. Maleki, "Distributed data storage with minimum storage regenerating codes-exact and functional repair are asymptotically equally efficient," in *arXiv preprint arXiv:1004.4299*, 2010.

[16] N. B. Shah, K. Rashmi, P. V. Kumar, and K. Ramchandran, "Interference alignment in regenerating codes for distributed storage: Necessity and code constructions," *IEEE Transactions on Information Theory*, vol. 58, pp. 2134–2158, 2012.

[17] A. Duminuco and E. Biersack, "A practical study of regenerating codes for peer-to-peer backup systems," in *29th IEEE International Conference on Distributed Computing Systems.*, (Montreal, Canada), 2009, pp. 376–384.

[18] A. Duminuco and E. W. Biersack, "Hierarchical codes: A flexible tradeoff for erasure codes in peer-to-peer storage systems," *Peer-to-peer Networking and Applications,*, vol. 3, pp. 52–66, 2010.

[19] M. Blaum, J. Brady, J. Bruck, and J. Menon, "Evenodd: An efficient scheme for tolerating double disk failures in raid architectures," *IEEE Transactions on Computers*, vol. 44, pp. 192–202, 1995.

[20] L. Xu and J. Bruck, "X-code: MDS array codes with optimal encoding," *IEEE Transactions on Computers*, vol. 45, pp. 272–276, 1999.

[21] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, (San Francisco, USA), 2014, pp. 1–14.

[22] C. Huang and L. Xu, "Star: An efficient coding scheme for correcting triple storage node failures," *IEEE Transactions on Computers*, vol. 57, pp. 889–901, 2008.

[23] J. L. Hafner, "Weaver codes: Highly fault tolerant erasure codes for storage systems," in *FAST*, (San Francisco, USA), 2005, pp. 16–16.

[24] L. Huang, S. Pawar, H. Zhang, and K. Ramchandran, "Codes can reduce queueing delay in data centers," in *IEEE International Symposium on Information Theory*, (Cambridge, USA), 2012, pp. 2766–2770.

[25] N. B. Shah, K. Lee, and K. Ramchandran, "The MDS queue: Analysing the latency performance of erasure codes," in *IEEE International Symposium on Information Theory*, (Honolulu, USA), 2014, pp. 861–865.

[26] G. Joshi, Y. Liu, and E. Soljanin, "On the delay-storage trade-off in content download from coded distributed storage systems," *IEEE Journal on Selected Areas in Communications*, vol. 32, pp. 989–997, 2014.

[27] N. B. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency?," in *Allerton Conf*, (Monticello, USA), 2013.

[28] G. Liang and U. C. Kozat, "Fast Cloud: Pushing the envelope on delay performance of cloud storage with coding," *IEEE/ACM Transactions on Networking*, vol. 22, pp. 2012–2025, 2014.

[29] L. E. Dickson, "Linear Groups: With an exposition of the Galois field theory," *Courier Dover Publications*, 2003.

[30] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster" *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, (San Jose, USA), 2013.

[31] E. Pinheiro, W. D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *FAST*, (San Jose, USA), 2007, pp. 17–23.

[32] K. M. Greenan, X. Li, and J. J. Wylie, "Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs," in *IIEEE 26th Symposium on Mass Storage Systems and Technologies*, (Incline Village, USA), 2010, pp. 1–14.

[33] L. Kleinrock, "Queueing Systems: Volume 2: Computer Applications," *John Wiley & Sons*, New York, 1976.

[34] A. Fikes, "Storage architecture and challenges," *Talk at the Faculty Summit*, 2010.

[35] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V. A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *OSDI*, (Vancouver, Canada), 2010, pp. 61–74.

[36] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling, "HDFS RAID," in *Hadoop User Group Meeting*, 2010.

[37] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan, "Subtleties in tolerating correlated failures in wide-area storage systems," in *NSDI*, (San Jose, USA), 2006, pp. 225–238.

Qiqi Shuai (S'14) received the B.Eng. degree in electronic engineering from Shanghai Jiao Tong University (SJTU), Shanghai, China, in 2012, and is currently working toward the Ph.D. degree at the University of Hong Kong (HKU), Hong Kong. His research interests include erasure coded distributed storage systems, including delay performance, bandwidth cost and storage overhead tradeoff, and locality and networking techniques used in storage systems.

Victor O.K. Li (S'80−M'81−F'92) received SB, SM, EE and ScD degrees in Electrical Engineering and Computer Science from MIT in 1977, 1979, 1980, and 1981, respectively. He is Chair Professor of Information Engineering and Head of the Department of Electrical and Electronic Engineering at the University of Hong Kong (HKU). He has also served as Assoc. Dean of Engineering and Managing Director of Versitech Ltd., the technology transfer and commercial arm of HKU. He served on the board of China.com Ltd., and now serves on the board of Sunevision Holdings Ltd. and Anxin-China Holdings Ltd., listed on the Hong Kong Stock Exchange. Previously, he was Professor of Electrical Engineering at the University of Southern California (USC), Los Angeles, California, USA, and Director of the USC Communication Sciences Institute. His research is in the technologies and applications of information technology, including clean energy and environment, social networks, wireless networks, and optimization techniques. Sought by government, industry, and academic organizations, he has lectured and consulted extensively around the world. He has received numerous awards, including the PRC Ministry of Education Changjiang Chair Professorship at Tsinghua University, the UK Royal Academy of Engineering Senior Visiting Fellowship in Communications, the Croucher Foundation Senior Research Fellowship, and the Order of the Bronze Bauhinia Star, Government of the Hong Kong Special Administrative Region, China. He is a Registered Professional Engineer and a Fellow of the Hong Kong Academy of Engineering Sciences, the IEEE, the IAE, and the HKIE.