MFTS: A Multi-Level Fault-Tolerant Archiving Storage with Optimized Maintenance Bandwidth

Jianzhong Huang, Xiao Qin, *Senior Member, IEEE*, Fenghao Zhang, Wei-Shinn Ku, *Senior Member, IEEE*, and Changsheng Xie, *Member, IEEE*

Abstract—In this paper, we propose a multi-level fault-tolerant storage cluster called MFTS, which provides flexible reliability for a wide variety of applications. MFTS makes use of a reliability upper-bound (i.e., Parameter *r*) to guide the process of adjusting fault-tolerance levels, i.e., i-erasure(s) and $i \in \{1, 2, \ldots, r\}$. In particular, MFTS can map an appropriate coding scheme to an application with individual reliability requirements. MFTS is capable of partitioning multi-level reliable storage using a virtual storage space, thereby adapting to any changing reliability demands of applications. We present the implementation of the MFTS system, which adopts an intersecting zigzag sets code (IZS code) rather than replication or general-purpose erasure codes. Our MFTS has three salient features: partial updates, fast reconstructions, and minimal overhead of fault-tolerance level transitions. To quantify performance optimization in our storage cluster, we compare IZS-enabled MFTS with two storage clusters equipped with the Vandermonde- and Cauchy-Reed-Solomon codes. The experimental results show that: 1) three schemes have comparable user-response-time performance in both the operational and degraded modes; 2) MFTS outperforms the other two alternatives by up to 26.1 percent in the offline reconstruction case; 3) MFTS speeds up the online reconstruction by up to 23.7 percent over the other two schemes with marginal increase in user response time.

Index Terms—Erasure-coded storage cluster, multi-level fault tolerance, optimized maintenance bandwidth, TCP-Incast

1 INTRODUCTION

In large data centers, data loss is unacceptable for many applications like banking systems, E-mail servers, and weather forecasting systems. Three-way replication is one of the most popular fault-tolerant techniques used in modern distributed storage systems (e.g., the Google file system [1] and the Hadoop system [2]). Although the replication is of high-performance, it inevitably leads to low storage utilization. Extra space for replicas can cost additional millions of dollars for a large-scale data center. To construct inexpensive large-scale storage systems, one can rely on erasure codes to fully utilize capacity of storage devices [3], [4]. Erasure codes are deployed in real-world storage systems, such as OceanStore [5], FAB [6], Pergamum [7], and Azure [8], etc. From the perspective of applications, it is desirable to provide an appropriate fault-tolerance level for each application's data. Therefore, we design a *M*ulti-level *F*ault-*T*olerant *S*torage cluster (MFTS) to dynamically offer multiple redundancy levels at run time. To achieve high performance, we incorporate an erasure code with optimized maintenance into the MFTS system.

- *J. Huang, F. Zhang, and C. Xie are with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology (HUST), Wuhan, Hubei 430074, P.R. China.*
E-mail: {hjzh, cs_xie}@hust.edu.cn, feng7188@gmail.com.
- *X. Qin and W.-S. Ku are with the Department of Computer Science and Software Engineering, Shelby Center for Engineering Technology, Samuel Ginn College of Engineering, Auburn University, AL 36849.*
E-mail: {xqin, weishinn}@auburn.edu.

Manuscript received 9 Aug. 2013; revised 13 Dec. 2013; accepted 24 Jan. 2014. Date of publication 2 Feb. 2014; date of current version 12 Nov. 2014.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TDSC.2014.2304296

Data volume has continuously increased over the last decade [9]. The massive amount of data that need to be archived leads to large-scale clustered storage, and commodity-based storage clusters have continued growing in popularity because of their cost-effectiveness and scalability. Modern large-scale storage clusters inevitably and frequently experience failures; therefore, it is important to take storage reliability into account. These phenomena motivate us to improve availability for storage clusters by tolerating one or multiple node failures to deliver non-stop storage services to applications.

There are a variety of data-intensive applications like web-search, E-mail server and database system, each of which has specific reliability requirements. For instance, a Hadoop storage system stores each chunk in a triple-mirrored way to provide resiliency [2], and a mirror file system keeps an extra copy in a remote backup site to tolerate disasters [10]. Furthermore, the reliability requirements of an application may change during its execution. For example, to tolerate data loss caused by a single failure, an E-mail server may employ a RAID-1 or RAID-5 system according to performance and capacity requirements. Recently stored E-mail files are more likely to be accessed than those E-mails saved a couple of months ago. In this case, the E-mail server can keep an extra copy for recent E-mail files in RAID-1 for good I/O performance, and save old email files using RAID-5 to improve storage utilization. Therefore, to meet such dynamically changing reliability requirements of applications, it is necessary to develop storage systems offering multiple fault-tolerance levels.

It is straightforward to adopt multiple fault-tolerant schemes to realize a multi-level reliable storage system. For example, one may employ mirroring, RAID-6, and $(k+3, k)$

Reed-Solomon codes to tolerate one, two, and three failures, respectively. When a system administrator intends to upgrade the fault-tolerance level from level 2 (i.e., 2-fault-tolerant) to 3, it is time-consuming to recompute all redundant data from the original data. Even worse, rebuilding a failed node requires reading all of the data in k surviving nodes, leading to the problem of *maintenance bandwidth*.

The goal of this study is to develop a multi-level fault-tolerant storage cluster with minimal maintenance overhead caused by modifying data blocks, reconstructing failed nodes, or altering fault-tolerance level. This goal is achieved by proposing an implementable design for the *I*ntersecting *Z*igzag *S*ets codes (*IZS* codes) [11]. Our MFTS system has the following three optimization techniques: (1) Efficient modification via partial updates—only data fragments needed for modification are read or written. (2) Fast reconstruction via exact updates—only data fragments needed for reconstruction are read. (3) Minimal overhead of fault-tolerance transitions—enabling or disabling certain parity nodes to change fault-tolerance levels.

This paper's contributions are summarized as follows:

- We propose a framework of multi-level fault-tolerant storage system or MFTS, which provides flexible reliability for a wide variety of applications according to data features and access patterns. MFTS can map applications to appropriate coding schemes and support multi-level reliable storage space.
- We give an implementable design for the IZS code, which enables the MFTS system to have three salient advantages: (1) modify archived data through partial updates; (2) alter fault-tolerance level by constructing or disabling only parity nodes; and (3) reconstruct failed nodes via exact repairs. Thanks to the above functionalities, MFTS not only offers adjustable fault-tolerance levels to satisfy various application requirements, but also achieves optimized maintenance bandwidth.
- We implement the IZS($k+i$, k)-enabled MFTS in a commodity-based storage cluster, where $i \in \{1, 2, 3\}$. To quantify performance optimization, we compare MFTS with two storage clusters equipped with two Reed-Solomon codes. The experiments show that: (1) three schemes have comparable user-response-time performance in both the operational and degraded modes; (2) MFTS outperforms the other two alternatives by up to 26.1 and 23.7 percent under offline and online reconstructions, respectively.

The remainder of the paper is organized as follows. We review related work in Section 2. Section 3 presents the design issues of MFTS. Then, we develop an implementable design of IZS($k+r$, k) code for MFTS in Section 4. The availability analysis of MFTS can be found in Section 5. Section 6 provides the implementation details of MFTS. Section 7 presents functional tests and performance evaluation. Finally, Section 8 concludes the paper.

2 Related Work

There are several research projects adopting multiple coding schemes within a storage system, including AUTORAID [12], TSS [13], RAIF [14], DHIS system [15], Panasas File System (PanFS) [16], OceanStore [5], etc. However, each of these systems has individual research motivations. For example, AUTORAID applies RAID-1 to hot data, whereas RAID-5 to cold data; TSS accomplishes the task of data migration between two different redundancy schemes based on access pattern; RAIF supports flexible per-file storage policies using a stackable file system which stacks on top of the combination of several file systems; DHIS system uses application-level hints to select customized placement and caching policies for the data; PanFS employs RAID-1 pattern to store small files, and disperses large files across multiple storage nodes in RAID-5 mode.

TABLE 1
Comparison of Different Fault-Tolerance Solutions

System	Redundancy schemes	Prototype	I/O Mode
Autoraid [12]	RAID-1, RAID-5	hardware-level	in-band
TSS [13]	RAID-1, RAID-5, cRAID-5	device driver	in-band
RAIF [14]	all RAID levels	filesystem-level	in-band
DHIS [15]	RAID-0, RAID-1, RAID-5	pseudo-driver	in-band
PanFS [16]	RAID-1, RAID-5	filesystem-level	out-of-band
OceanStore [5]	Replica and RS codes	filesystem-level	Peer-to-Peer
MFTS(Ours)	*IZS with Optimized Update*	*filesystem-level*	*out-of-band*

Conceptually, MFTS provides the most appropriate fault-tolerant storage spaces for various applications with specified reliability requirements. MFTS achieves such a flexibility by placing important data to storage nodes with high reliability. Most fault-tolerant storage systems using erasure codes apply one of the following two families of codes, namely, XOR-based RAID codes and Reed-Solomon codes. Existing storage systems based on RAID codes are unable to dynamically add a new redundancy scheme to upgrade the redundancy level. For example, owing to the specific parity layout of RAID-5 and RAID-6, it is burdensome to replace the 1-fault-tolerant RAID-5 layout by the 2-fault-tolerant RAID-6 one. As a generalization of RAID, systematic Reed-Solomon codes (RS codes) offer a good tradeoff between reliability and storage utilization [17]. As such, we adopt a systematic RS-like coding to develop a multi-level reliable storage system, i.e., MFTS.

The above-mentioned storage solutions hold two or more redundancy schemes in a system, providing transparent and reliable storage services for upper applications. These solutions can be accomplished at the hardware-level [12], driver-level [13], [15] and file-system-level [14], [16] (see Table 1, third column). Compared to the hardware or driver-level approach, a file-system-level scheme can support a flexible placement policy using the higher-level information, and per-file storage policies can help to quickly locate the unrecoverable faults. Therefore, we develop the prototype of MFTS at the file-system level.

Within storage virtualization, all data and I/O requests pass through virtualization devices in 'In-band' mode. Thus, AUTORAID, TSS, RAIF, and DHIS systems belong to 'In-band' mode (see Table 1, fourth column). For example, RAIF is mounted atop the combination of networked, disk-based and memory-based file systems, all requests and data must be transferred via the RAIF Module. 'Out-of-band' is necessary and suitable for distributed storage clusters focusing on system scalability. In 'Out-of-band' mode, the host can directly read/write data from/to storage nodes,

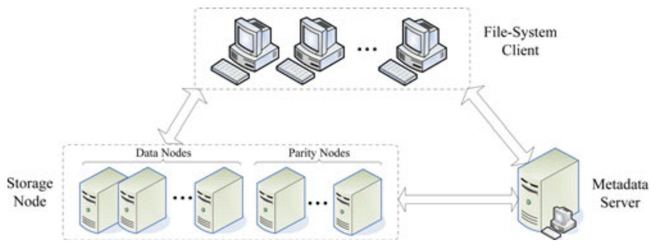


Fig. 1. Architecture of a (k + r, k) erasure-coded storage cluster, where there are k data nodes and r parity nodes.

ensuring the system scalability. Therefore, we implement our multi-level storage system in 'Out-of-band' mode (see Fig. 1).

In this paper, we implement a prototype of MFTS in 'Out-of-band' mode at the file-system level, providing appropriate reliable storage space based on the specific reliability requirement of the application. In addition, MFTS not only simultaneously supports multiple fault-tolerance levels in commodity-based storage clusters, but also provides highly efficient transitions of fault-tolerance levels via only updating redundancy data.

3 Framework of a Multi-Level Reliable Storage Cluster

Now we describe the framework of our multi-level reliable storage, which selects an appropriate coding scheme for an application imposing specific reliability requirements.

3.1 Architecture

From users' perspective, MFTS offers a file system interface just like NFS. Underneath MFTS is a distributed storage system. Fig. 1 depicts the architecture of MFTS, which belongs to a commodity-based storage cluster. The metadata server manages the control information of the whole system, including information about the distribution of data blocks. When users access a file, the file-system client looks up the file-metadata from the metadata server. The storage nodes can manage low-level storage allocation themselves, which adopt the commodity hardware owing to IT investment. These storage nodes are assumed to be distributed. As to a certain (k + r, k) erasure code, MFTS distributes the user data among the *k* systematic nodes (a.k.a., data nodes) and stores the redundancy data in the *r* parity nodes, achieving access parallelism. That is, data is typically stridden across multiple storage nodes to enable high aggregated I/O bandwidth. When the number of failed nodes is not greater than r (i.e., $\leq$r), MFTS can still provide storage service to front-end users uninterruptedly.

3.2 Mapping Applications to Coding Schemes

Data of different applications are likely to exhibit various characteristics such as essential attributes and access patterns. Essential attributes include data importance and inherited attributes (e.g., read-only, writable, append-only, etc.); access patterns refer to access frequencies and I/O types (e.g., read-intensive or write-intensive). Each redundancy scheme has individual features with respect to fault-tolerance, computation complexity, and storage utilization.

TABLE 2
Association between Coding Scheme and Data Characteristic

Data Importance	Access Frequency	Optional Placement Pattern
Not important	—	RS(k,k)
Important	Low frequency	RS(k+1,k)
Important	High frequency	Mirroring
Very important	Low frequency	RS(k+2,k)
Very important	Medium frequency	Mirroring+RS(k+1,k)
Very important	High frequency	Mirroring+Mirroring

Our MFTS relies on a mapping algorithm to offer appropriate mappings between coding schemes and applications. Algorithm 1 shows how to map an application requirement to an appropriate coding scheme. Three principles are followed as follows. First, to choose redundancy schemes that can meet applications' specified reliability goals. For example, a storage cluster stores only original data for 'not important' data, and keeps an extra copy for 'important' data, and tolerates two failures for 'very important' data (see lines 3, 6 and 13, respectively). Second, to allocate a coding scheme with low computation complexity (e.g., Mirroring) for frequently accessed data (see line 8). Last, to assign a coding scheme with high storage utilization (e.g., RS codes) to less-frequently accessed data (see line 10).

To demonstrate how Algorithm 1 works, let us consider a special case where application requirements include 'data importance' and 'access frequency', and candidate coding schemes include Mirroring, RS(n, k), 'Mirroring+RS(n,k)'. The RS(k, k) scheme partitions data into data blocks to be dispersed across multiple storage spaces without parity. Similar to RAID 1 + 5 [18], the 'Mirroring + RS(n, k)' scheme stores an extra copy of all the data blocks except parity blocks. The mapping outcome obtained by Algorithm 1 is summarized in Table 2. Note that the above special case (i.e., with factors of data importance and access frequency) can be extended to incorporate other factors, which may lead to more comprehensive and reasonable mappings.

Algorithm 1: Mapping an application to a coding scheme

```
   Input: Application Attributes,  e.g., data importance, access frequency
   Output: Coding Scheme
 1
 2 switch Data Importance do                          // 0-Fault-tolerance
 3     case Not Important
 4         Coding=RS(k,k); break;
 5     endsw
 6     case Important                                 // 1-Fault-tolerance
 7         if High access frequency then
 8             Coding=Mirroring; break;
 9         else
10             Coding=RS(k+1,k); break;
11         end
12     endsw
13     case Very Important                            // 2-Fault-tolerance
14         if High access frequency then
15             Coding=Mirroring+Mirroring; break;
16         else if Medium access frequency then
17             Coding=Mirroring+RS(k+1,k); break;
18         else
19             Coding=RS(k+2,k); break;
20         end
21     endsw
22 endsw
23 return Coding
```

In summary, as to infrequently accessed data, it is inclined to employ the coding scheme of high storage efficiency, e.g., RS(k+1, k) and RS(k + 2, k), and RS(k + 1, k)

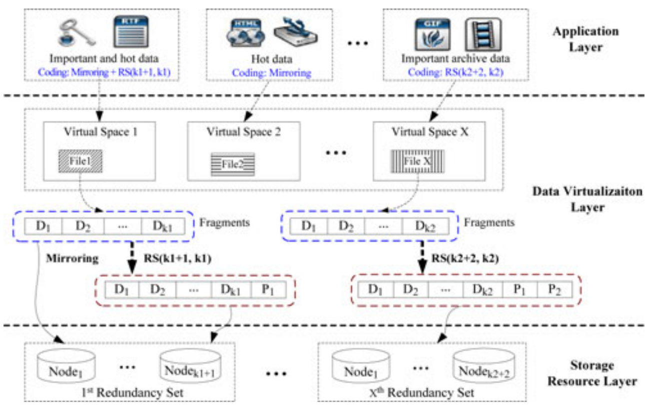


Fig. 2. Requirement-oriented reliable storage.

or RS(k + 2, k) is chosen according to the specific data importance. As to hot data which needs high performance, it is common to adopt the Mirroring scheme because of its no computation overhead and low response time. Especially, 'Mirroring', 'Mirroring + RS(k + 1, k)' or 'Mirroring + Mirroring' is then chosen depending on the data importance. Although both Mirroring and RS(k + 1, k) tolerate single failure, the former is good at access performance and the latter has higher storage efficiency.

3.3 Virtual Reliable Storage Space

From the viewpoint of management strategy, requirement-oriented data reliability is referred to provide an appropriate reliable scheme for each application. Each application data is logically stored in an independent storage space. Currently, storage systems usually map physical space to logical space by 'Volume', which is a key element of block-based storage virtualization. A use case of 'Volume' is RAID (e.g., RAID-1 can map two physical spaces to a logical space). Because networked storage service (e.g., cloud storage) usually offers a file-based access interface, it is natural to employ file-based data access to achieve a reliable storage system. That is, the block-level reliability strategy in disk array is shifted up to the file-level in cluster storage.

As shown in Fig. 2, the multi-level reliable storage system is partitioned into three layers: application layer, data virtualization layer and storage resource layer. The application layer takes charge of the decision-making of the reliability strategy, and the data virtualization layer is responsible for the layout and organization of stored data. The physical storage resource is transparent to upper-level applications because of the virtualization layer. An important application with high access frequency is mapped to '*Mirroring + RS(k + 1, k)*' scheme, meaning that two identical copies of the application data and a copy of the corresponding parity data are stored to underlying storage devices.

3.4 Put It All Together

In a storage cluster, we divide its storage nodes into several redundancy sets, which correspond to the storage groups shown in Fig. 6. Each redundancy set constitutes an individual virtual reliable storage space adopting a special coding scheme (see Fig. 2). According to the mapping method, an application is directed to an appropriate redundancy set, then the application data are dispersed across multiple nodes which belong to the redundancy set, thereby meeting its reliability requirement.

As to fault-tolerant storage, it is necessary to consider three metrics as follows: recovery capability, maintenance overhead, and storage utilization. This study is especially concentrated on the maintenance/updating overhead of the multi-fault-tolerance-level storage system. The following three cases may result in updating issues: data modification, rebuilding of failed nodes, and transition of fault-tolerance level. Take fault-tolerance level-transition as an example, when upgrading the fault-tolerance level, traditional erasure codes entirely regenerate all new redundancy data rather than selectively generate a fraction of redundancy data. Thus it is necessary to consider the updating issue of redundancy schemes.

4 Erasure Codes with Optimized Maintenance Bandwidth

4.1 Background

Replication and erasure coding are the two primary redundancy schemes for the reliable storage system. Although data replication can achieve low response time, it is desirable that the distributed storage system efficiently employs the storage space and networking bandwidth. Erasure codes can reduce bandwidth by an order of magnitude in comparison with replication and provide higher storage space utilization for a given reliability level [19].

It is well known that an (n = k + r, k) erasure code can be used to store information in a distributed storage system with n nodes. If the code is maximum distance separable (MDS), the storage system can recover the original information from any k surviving nodes. If the code is a systematic code, the information is stored exclusively in the first k nodes, and the redundancy data are stored exclusively in the last r nodes, where r is a parameter that can describe the reliability level. RS codes are not only the MDS codes, but also belong to systematic codes [20], [21], [22].

Within (n, k) erasure codes, if parameter n is different (e.g., n_1, n_2, with $n_1 < n_2$), then the fault-tolerance capability of (n_1, k) erasure code is weaker than that of (n_2, k) erasure code, and the storage utilization of (n_1, k) erasure code is higher than that of (n_2, k) erasure code (i.e., k/n_1 versus k/n_2). As described in Section 3.2, we associate the different coding schemes with different applications.

4.2 Issues of Maintenance Bandwidth

There exists an issue of *maintenance bandwidth* in erasure-coded storage—how much information is accessed for updating or rebuilding. Wu et al. [23] introduced a new class of erasure code (termed regenerating code) to enable efficient node repair. In [24], a novel 'Twin-code framework' was proposed to enable repair-bandwidth minimization. But they offer less flexibility because of the restriction of $n \geq 2k$.

There are three typical maintenance operations in the multi-level fault-tolerant storage: (1) modifying data, (2) rebuilding failed node(s), and (3) transition of fault-tolerance

TABLE 3
Symbols and Definitions

Symbols	Definition
block	A basic unit of data or parity; it is the building block in erasure coded storage cluster, analogous to strip in RAID
packet	A transferring unit through TCP/IP
element	A calculating unit in the IZS coding algorithm
k	Number of information columns in array A
r	Number of parity columns/nodes
p	Number of rows in array A, $p \geq k$, e.g., p=$2^{k\text{-}1}$
$a_{i,j}$	Element of array A, $0 \leq i < p$ and $0 \leq j < k$
C	Columns, C=$\{C_0, C_1, \ldots, C_{k\text{-}1}, C_k, \ldots, C_{k+r\text{-}1}\}$, where C_{k+i} is the i^{th} parity column, $0 \leq i < r$
e	Unit vectors, e=$\{e_0, e_1, \ldots, e_{k\text{-}1}\}$, and e_0 is a zero vector
u_i	Sum vector, $u_i = \sum_{l=0}^{i} e_l$, where $0 \leq i < k$
$f_i^g(x)$	Permutation f^g for g^{th} parity node, $f_i^g(x) = x{+}ge_i$
$a_{x,\overline{j}}$	Complement of $a_{x,j}$ in an information element subset, $a_{x,\overline{j}}$ = $\{a_{x,l} : 0 \leq l < k$ and $l \neq j\}$
r_i	The i^{th} entry in row parity column C_k
z_i^g	The i^{th} entry in parity column C_{k+g}
Z_x^g	The x^{th} generating set of column C_{k+g}
$\beta_{i,j}^g$	The coefficient of $a_{i,j}$ in parity column C_{k+g}

levels (i.e., storing the encoded data using another coding scheme). To address the maintenance issue, we introduce an MDS array code named *IZS code* [11] to the design of MFTS and propose an implementable design for the IZS code, thereby enabling MFTS to achieve low maintenance bandwidth: (1) when a data block is modified, the parity block(s) can be exactly updated without accessing any other data blocks; (2) when any of the parity nodes are erased, the parity node can be rebuilt efficiently because of its simple encoding procedure; (3) if any of the data nodes are erased, the information node can be rebuilt by accessing about $1/r$ of all the surviving data blocks; (4) to downgrade or upgrade the fault-tolerance level, MFTS simply discards or generates the parity node, respectively.

4.3 Intersecting Zigzag Sets Codes

The maintenance/updating operations in MFTS include data modification, reconstruction of failed nodes, and transition of the fault-tolerance level. As to data modification, it is expected that the erasure code may efficiently update the parity blocks. The decoding efficiency of a coding scheme may significantly influence the reconstruction time. As to level-transition, we restrict the modification to the redundant data, so MFTS can keep the same information data after changing the fault-tolerance level. To optimize the maintenance/updating performance, it is important to ensure the number of accessed data is minimal. Aiming at the design goal of minimizing maintenance bandwidth, we develop a set of coding algorithms, including construction, modification, level-transition, and reconstruction.

IZS codes were proposed by Tamo et al. [11]. The IZS codes provide a solid theoretical foundation for the design and implementation of our MFTS. In this paper, we present the *first* implementable design for the IZS codes in storage clusters. Our IZS implementation relies on the coding algorithm of the RS codes.

Let $A = (a_{i,j})$ be an array of size p $\times$ k. A column is also called a node, and each entry is an information element. In order to retain the MDS property and minimize the *maintenance bandwidth* in the network, each information element in A should appear exactly once in each parity column. In this section we give the general algorithm of construction and reconstruction for (k $+$ r, k) MDS code, show how to modify information data, and show how to upgrade or downgrade the fault-tolerance level in MFTS. Before presenting the algorithms in pseudo-code, we summarize the symbols used in the algorithms in Table 3.

The construction consists of two steps: first choosing the appropriate parity sets, and then determining the coefficients in the linear combinations in order to make sure that the constructed code is an MDS code. Thus, IZS is an RS-like code, where the parity is simply appended to the source data in a systematic mode, and the parity is the linear combination of source data elements. Particularly, the first parity column is the sum of each row of A. The construction process is given in Algorithm 2.

Algorithm 2: Construction of g^{th} parity node C_{k+g} in (k+r,k) MDS code, with $1 \leq g \leq r\text{-}1$

```
Input: $k, p, g$
Output: $Z_x^g, \beta^g$

 /* Determining the permutation($f_v^g$)                                  */
for $v \leftarrow 0$ to $k$-1 do
    for $x \leftarrow 0$ to $p$-1 do
        $f_v^g(x) = x{+}ge_v$;
    end
end
 /* Choosing elements and coefficients for set $Z_x^g$                   */
foreach element $a_{i,j}$, $0 \leq i \leq p\text{-}1$ and $0 \leq j \leq k\text{-}1$ do
    if $f_j^g(i) == x$ then
        Put $a_{i,j}$ to set $Z_x^g$;
        if $u_j \cdot i{==}1$ then
            $\beta_{i,j}^g$ = $2^g$;
        else
            $\beta_{i,j}^g$ = 1;
        end
    end
end
return $Z_x^g, \beta^g$
```

Algorithm 3 illustrates the procedure of data modification. It is observed that r $+$ 1 reads (i.e., reading r old parity elements and one old information element) and r $+$ 1 writes (i.e., writing r new parity elements and one new information element) are needed to modify a single information element $a_{i,j}$, without reading any other information data. Thus, such an updating is deterministic and partial.

Algorithm 3: Modifying an element in (k+r,k) MDS code

```
 /* Assume element $a_{i,j}$ is overwritten with $a'_{i,j}$              */

Get $a_{i,j}$;   // Reading original element
for $g \leftarrow 0$ to $r$-1 do
    Get $z_{f_j^g(i)}$;   // Reading parity element
    if $u_j \cdot i{==}1$ then $\beta_{i,j}^g$ = $2^g$; else $\beta_{i,j}^g$ = 1;
    $z_{f_j^g(i)}$ + = $(\beta_{i,j}^g a_{i,j} + \beta_{i,j}^g a'_{i,j})$;
    Put $z_{f_j^g(i)}$;   // Writing new parity element
end
Put $a'_{i,j}$;   // Writing new element
```

Algorithm 4 presents the pseudo-code of level-transition. When the fault-tolerance level decreases, MFTS will discard only some parity nodes (see line 7), which means that MFTS responds to the I/O requests using the coding scheme offering low fault-tolerance level after level degradation. On the other hand, MFTS must generate the extra parity nodes using the construction procedure to improve the reliability (see line 3). No matter which case, the elements in the systematic node are kept intact so they can achieve an excellent updating performance.

Algorithm 4: Transition of fault-tolerance level for (k+r,k) code

```
/* Assume $r_c$ and $r_t$ are the current and target fault
tolerance level, respectively, where $r_c \neq r_t$.          */

if $r_c < r_t$ then
    for $r \leftarrow r_c$+1 to $r_t$ do
        Generate the parity column $C_{k+r\text{-}1}$ using the construction procedure
        listed in Algorithm 2;
    end
else
    for $r \leftarrow r_t$+1 to $r_c$ do
        Discard the parity column $C_{k+r\text{-}1}$;
    end
end
```

For the jth failed information node C_j, one can reconstruct the node using the complement of $a_{i,j}$, i.e., $a_{i,\bar{j}}$. The number of complement elements approximates $1/r$ of surviving information elements. The reconstruction process is presented in Algorithm 5.

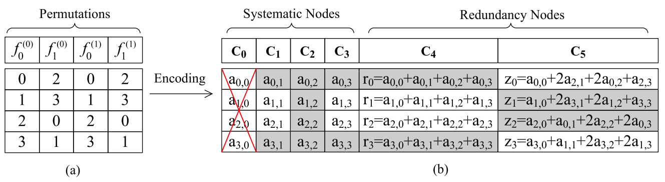
Algorithm 5: Reconstruction of j^{th} failed information node C_j in (k+r,k) Code, with $0 \leq j < k$

```
Input: $k, p, j$
Output: $a_{i,j}$,   $i \in \{0, 1, ..., p\text{-}1\}$

/* Determining the needed information elements               */
if j==0 then $e_0$=(1,1,...,1);
for $i \leftarrow 0$ to $p$-1 do
    if $i \cdot e_j$==0 then
        Get the surviving information elements $a_{i,\bar{j}}$;
    end
end
/* Calculating the failed information elements               */
for $i \leftarrow 0$ to $p$-1 do
    if $i \cdot e_j$==0 then
        Calculate $a_{i,j}$ using $z_{f_j^0(i)}$ and $a_{i,\bar{j}}$;
    else
        Calculate $a_{i,j}$ using $z_{f_j^1(i)}$ and $a_{i,\bar{j}}$;
    end
end
return $a_{0,j}, a_{1,j}, ..., a_{p\text{-}1,j}$
```

It is a conventional wisdom to transfer large packets to improve network bandwidth utilization; that is, it is inclined to adopt a small row number (i.e., parameter *p*). Therefore, we choose code-duplication mechanism [11] to increase the number of columns. Algorithm 6 gives the pseudo-code of a *2-duplication*-based (2k + 2, 2k) code.

Algorithm 6: Construction of a 2-duplication-based (2k+2,2k) code

```
/* Z is the generating set of parity column $C_{2k+1}$, and $\beta_{i,j}$
is the coefficient of element $a_{i,j}$.                        */
Input: $k, p$
Output: $Z, \beta$,   Z={$Z_0, Z_1, ..., Z_{p\text{-}1}$}

/* Determining the permutation($f_i$)                          */
for $i \leftarrow 0$ to $k$-1 do
    for $x \leftarrow 0$ to $p$-1 do
        $f_i(x) = x$+$e_i$;
    end
end
/* Choosing elements and coefficients for set Z                */
for $l \leftarrow 0$ to $p$-1 do
    foreach element $a_{i,j}$, $0 \leq i \leq p$-1 and $0 \leq j \leq 2k$-1 do
        $t = j$ mod $k$;
        Put $a_{i,j}$ to set $Z_{f_t(i)}$;
        if ($j < k$ and $u_t \cdot i$==1) or ($j \geq k$ and $u_t \cdot i$≠1) then
            $\beta_{i,j}$ = 2;
        else
            $\beta_{i,j}$ = 1;
        end
    end
end
return $Z, \beta$
```



$f_0^{(0)}$	$f_1^{(0)}$	$f_0^{(1)}$	$f_1^{(1)}$
0	2	0	2
1	3	1	3
2	0	2	0
3	1	3	1

C_0	C_1	C_2	C_3	C_4	C_5
$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$r_0=a_{0,0}+a_{0,1}+a_{0,2}+a_{0,3}$	$z_0=a_{0,0}+2a_{2,1}+2a_{0,2}+a_{2,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$r_1=a_{1,0}+a_{1,1}+a_{1,2}+a_{1,3}$	$z_1=a_{1,0}+2a_{3,1}+2a_{1,2}+a_{3,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$r_2=a_{2,0}+a_{2,1}+a_{2,2}+a_{2,3}$	$z_2=a_{2,0}+a_{0,1}+2a_{2,2}+2a_{0,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$r_3=a_{3,0}+a_{3,1}+a_{3,2}+a_{3,3}$	$z_3=a_{3,0}+a_{1,1}+2a_{3,2}+2a_{1,3}$

Fig. 3. (a) The set of permutations for a 2-duplication (6, 4) code. (b) A 2-duplication (6, 4) MDS code generated by the left permutations. The first redundancy node C_4 is the row sum and the second redundancy node C_5 is generated by the construction Algorithm 6. The shaded elements are needed to reconstruct the failed node C_0.

Algorithm 7 lists the reconstruction procedure of the 2-*duplication*-based (2k + 2, 2k) code. If an information node fails, about a little more than 1/2 of the surviving elements are needed to reconstruct the failed information node.

Algorithm 7: Reconstruction of j^{th} failed information node in 2-duplication (2k+2,2k) MDS code

```
Input: $j$ ,   $0 \leq j < 2k$
Output: $a_{i,j}$,   $i \in \{0, 1, ..., p\text{-}1\}$

/* Determining the needed information elements               */
if j==0 then $e_0$=(1,1,...,1);
for $i \leftarrow 0$ to $p$-1 do
    if $i \cdot e_j$==0 then
        Get the surviving information elements $a_{i,\bar{j}}$;
        if $j < k$ then
            Get the information elements $a_{i,j+k}$;
        else
            Get the information elements $a_{i,j\text{-}k}$;
        end
    end
end
/* $a_{i,j\pm k}$ denotes $\{a_{i,j+k}$ when $j < k$, or $a_{i,j\text{-}k}$ when $j \geq k\}$   */
/* Calculating the failed information elements               */
for $i \leftarrow 0$ to $p$-1 do
    if $i \cdot e_j$==0 then
        Get the row parity element $r_i$;
        Calculate $a_{i,j}$ using $r_i$ and $a_{i,\bar{j}}$;
    else
        Get the zigzag parity element $z_{f_j(i)}$;
        Calculate $a_{i,j}$ using $z_{f_j(i)}$, $a_{i,\bar{j}}$ and $a_{i,j\pm k}$;
    end
end
return $a_{0,j}, a_{1,j}, ..., a_{p\text{-}1,j}$
```

Assume the size of array A is 4 × 4. We use 2-duplication scheme (see Algorithm 6) to construct a (6,4) MDS code with array size 4 × 6. As show in Fig. 3, the shaded elements should be accessed to reconstruct the failed systematic node C_0 according to Algorithm 7. We employ the 2-duplication IZS(6, 4) code in the comparative reconstruction experiment where the fault-tolerance level is 2-erasures.

4.4 Amount of Data Used for Reconstruction

Since a single erasure is more likely to happen than $r \geq 2$ concurrent erasures, we take single-node reconstruction as an example. Especially, we compare the number of elements needed to be accessed to rebuild a systematic node.

According to the reconstruction procedure listed in Algorithm 7, we can get the number of elements in 2-duplication (k + 2, k) MDS code:

$$N_{IZS} = p \times (k + r - 1)/r \; + \; p \times (r - 1)/r. \tag{1}$$

With the original property of MDS codes, a user can read any *k* nodes to reconstruct a failed node. That is:

$$N_{RS} = p \times k. \tag{2}$$

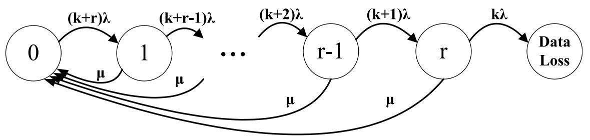


Fig. 4. Markov chain for the RS(k + r, k) code with concurrent rebuild.

The data-amount ratio R_{data} is:

$$R_{data} = \frac{N_{RS}}{N_{IZS}} = \frac{kr}{k+2r-2}. \qquad (3)$$

5 Availability Analysis

A handful of studies focus on node-oriented availability analysis for replicated systems using Markov models (for example, [25]), whereas many disk-oriented availability models are developed for RAIDs [26], [27]. Now we describe a node-oriented model for RS-coded storage. Our model extends the existing disk-oriented model by considering IZS coding in the context of storage clusters.

5.1 Availability Model

The MFTS system is built using a collection of storage nodes, thus MFTS achieves high reliability through redundant nodes. We assume a node consists of disk drives, a network interface controller, and a power supply. Similar assumptions can be found in [28]. Each component inside a node has no duplicated one, meaning that the node fails if any one of these components encounter failures. Each data strip is stored on one disk in a storage node.

The reliability of RS-coded storage can be estimated using a Markov model [29], [30]. Let μ and λ denote the repair rate and failure rate of a storage node, respectively. An MDS-coded storage system can be recovered to the fully operational state from any failure state excluding the *Data Loss* state (i.e., *Reconstruction-in-Parallel* or *Concurrent Rebuild*) [31]; the state transition probability is μ. Fig. 4 shows the Markov chain model for the RS(k + r, k) code. In this model, we assume that systems do not encounter any hard error during a rebuilding process of *r* failed nodes. State '0' represents a state in which all storage nodes are operational. State '1' and 'r' denote one and r failures, respectively. The probability of state transition from '0' to '1' is $(k+r)\lambda$, because there must be at least (k + r) failed nodes before the system transits into the first failure state.

Since our MFTS can accomplish node recovery in a *reconstruction-in-parallel* way, the mean time to repair (MTTR) a storage node relies on its node capacity. Usually, MTTR is much shorter than the mean time between failures or MTBF (see, for example, Table 4). That is, we have $\lambda \ll \mu$. Markov models are widely applied to analyze the mean time to data loss or MTTDL of storage systems. Using the canonical Markov model, we obtain the MTTDL value of an RS(k + 1, k)-coded redundancy set (see Eq. (4)). Similarly, MTTDL of the redundancy set using RS(k + 2, k) or RS(k + 3, k) can be expressed by Eqs. (5) and (6), respectively:

$$MTTDL_{RS(k+1,k)} \approx \frac{\mu}{k(k+1)\lambda^2}, \qquad (4)$$

TABLE 4
Parameters in an RS(n, k)-Coded Storage System

Parameter	Value
Z(Total data in the system)	4TB
γ(Node throughput for rebuild)	200GB/hrs
S(Data in a storage node)	1TB
$MTTF_{disk}$	$\approx 100,000$ hrs
$MTTF_{node}$	$\approx 20,000$ hrs
$MTTR_{node}$	S/γ (=5hrs)
λ(Failure rate of node)	$1/MTTF_{node}$ ($\approx 5\times 10^{-5}$)
μ(Repair rate of node)	$1/MTTR_{node}$ (=0.2)
k(Number of data nodes)	Z/S (i.e., k=4)
r(Number of redundancy nodes)	Varies. e.g., r=2 for RS(6,4) code

$$MTTDL_{RS(k+2,k)} \approx \frac{\mu^2}{k(k+1)(k+2)\lambda^3}, \qquad (5)$$

$$MTTDL_{RS(k+3,k)} \approx \frac{\mu^3}{k(k+1)(k+2)(k+3)\lambda^4}. \qquad (6)$$

5.2 Analysis of the Coding Schemes

Let us assume that both MTTF and MTTR of a storage node in MFTS follow the exponential distribution (i.e., $\lambda = 1/MTTF_{node}$, $\mu = 1/MTTR_{node}$). $MTTF_{disk}$ of the Seagate SV35.5 Series SATA drive specified in the product datasheet is approximately 1,000,000 hours [32]. For disks that are less than five years old, the replacement rate is larger than the vendor-specified MTTF by a factor of 2 ~ 10 [33]. Therefore, we let $MTTF_{disk}$ be 100,000 hours to consider the worst case. A study [34] reveals that the annual failure rate (AFR) of disks is about 20 percent of that of a low-end storage system (i.e., $MTTF_{node} \approx 0.2 \times MTTF_{disk} = 20{,}000$ hours).

We analyze the reliability of an RS-coded storage cluster where k equals to 4 (the system reliability can be easily quantified in the case of $k \neq 4$). Table 4 summarizes all the parameters for the availability model of an RS(n, k)-coded storage system. We apply Eqs. (4), (5), and (6) to calculate the MTTDL values (see Table 5). Apart from MTTDL, the table also lists the corresponding storage utilization.

Fig. 5 shows the reliability levels of six reliable storage systems. For example, AUTORAID [12], TSS [13], DHIS system [15], and PanFS [16] tolerate a single node failure. RAIF [14] can tolerate two failed nodes. Theoretically, the (k + r, k) erasure codes can tolerate $r = 4$ failures. However, there is no incentive to adopt RS(k + 4,k) because RS(6,4) and RS (7,4) in a 4-TB storage cluster have an MTTDL value of 3.04 $\times\ 10^5$ and 1.74×10^8 years, respectively (see Table 5). Furthermore, a handful of well-known storage systems like Hadoop [2] and GFS [1] maintain three copies for each file, tolerating two failures. Therefore, it is sufficient to employ the 3-fault-tolerant coding schemes codes for reliable storage systems. We implement the MFTS system using the IZS

TABLE 5
MTTDL and Utilization of an RS(n, 4)-Coded Storage Cluster

Coding Scheme		RS(4,4)	RS(5,4)	RS(6,4)	RS(7,4)
MTTDL	(hours)	5×10^3	4×10^6	2.7×10^9	1.5×10^{12}
	(years)	0.57	4.56×10^2	3.04×10^5	1.74×10^8
Storage Utilization		100%	80%	66.6%	57.1%

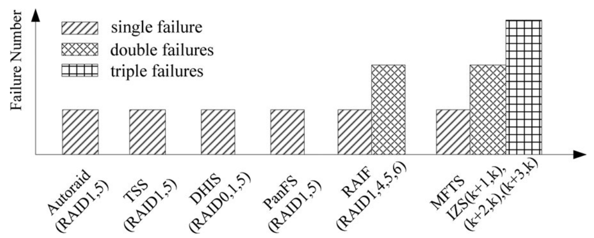


Fig. 5. Reliability levels of six storage systems.

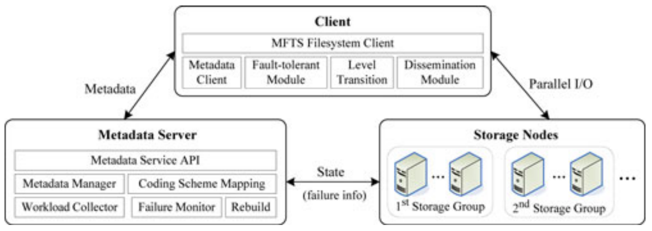


Fig. 6. An architecture overview of MFTS.

(k + 3, k) coding scheme, which includes the IZS(k + 1, k), IZS(k + 2, k), and IZS(k + 3, k) codes. Hence, MFTS can dynamically adjust fault-tolerance levels (i.e., fault-tolerance level $r \in \{1, 2, 3\}$).

6 Prototype of MFTS

We implemented an MFTS prototype using FUSE [35], a framework for building file systems in user-space. Fig. 6 shows an overview of MFTS's architecture. In our design, MFTS is a stackable file system which is located above local file systems; thus the global page cache can automatically cache both systematic data and redundancy data. The MFTS client takes charge of fault-tolerant operations (e.g., encoding, decoding, and level-transition) and dissemination of coded data, and provides a different reliable storage space to the upper applications. The metadata server is responsible for mapping coding schemes, rebuilding failed nodes, and collecting workloads besides the usual metadata management. Storage nodes are divided into different groups; each group offers appropriate fault-tolerant storage space to individual applications.

6.1 Interactive API

The MFTS metadata server provides storage access API to the file-system client. The API supports several file operations (e.g., create, read, write, delete, and repair, etc.) and *i*-node operations (e.g., lookup, update, etc.). Storage nodes respond to the I/O requests from the metadata server and file-system client. The clients use *mfts_create* to create new files, specifying a target location which has a specific fault-tolerance level. Operation *mfts_read* returns file data by reading data from storage nodes and decoding erasure-coded data when needed. Using the *mfts_write* operation allows the storing of new or modified file data to storage nodes and updating of the file-metadata on metadata server. Operation *mfts_rewrite* carries out the data modification by Algorithm 3. Operation *mfts_rebuild* reconstructs the failed systematic node by Algorithm 7 or the failed parity node by Algorithm 6. In multi-level fault-tolerant system, it is necessary to upgrade/downgrade the fault-tolerance level of a storage group to achieve the appropriate reliability capability. Operation *mfts_level* accomplishes it.

6.2 Fault-Tolerant Module

Fault-tolerant Module carries out erasure encoding on the data passed from the MFTS client or decoding on the encoded data gathered from the storage nodes via *Dissemination Module*. The encoding and decoding operations refer to the write and read requests, respectively. Particularly, the module can deal with the following request types: *mfts_create*, *mfts_read*, *mfts_write*, and *mfts_level*. For example, operation *mfts_level* generates the parity column C_5 using the construction procedure listed in Algorithm 6 when the fault-tolerance level *r* increases from 1 to 2, i.e., IZS(5, 4) $\rightarrow$ IZS(6, 4). On the contrary, *mfts_level* simply discards the parity column C_5 when the coding scheme IZS(6, 4) is replaced by IZS(5, 4). Thus, IZS(k + 2, k) can be seamlessly downgraded to IZS(k + 1, k) with minimum maintenance overhead.

6.3 Rebuilding Module

When the metadata server discovers a failed node through *Failure Monitor Module*, the rebuilding operation will be invoked. Take IZS(k + 2, k) as an example. If a systematic node fails, a replacement node is rebuilt by a reconstruction process in Algorithm 5. If the parity node C_k fails, a replacement node is the row sum of all the systematic nodes. If parity node C_{k+1} fails, a replacement node is rebuilt using the construction procedure in Algorithm 2. In the case of 2-erasures in IZS(k + 2, k), the rebuilding is sightly more complex. It is required to read all data blocks in *k* surviving nodes $\{C_1, C_2, \ldots, C_k\}$ when both systematic node C_0 and parity node C_{k+1} are erased. Node C_0 can be reconstructed from both surviving systematic nodes $\{C_1, C_2, \ldots, C_{k-1}\}$ and parity node C_k leveraging the rule that the first parity column is the sum of each row of A. Then, parity node C_{k+1} can be recovered using Algorithm 2. The rebuilding process is transparent to the foreground applications, so MFTS clients can provide an online storage service to the applications during the reconstruction.

6.4 Coding Scheme Mapping Module

This module completes the function of mapping from an application to an appropriate coding scheme (see Section 3.2). At the heart of the coding scheme mapping module is a workload collector, which intently collects workload information when MFTS client issues I/O requests. The workload information is used to quantify the access frequency of application data. Data importance is designated manually by system administrators. According to the access frequency and data importance, the module decides an appropriate coding scheme for the application data. The resulting coding scheme will activate *Fault-tolerant Module* to fulfill the task of encoding or decoding.

In the module implementation, access frequency is measured as the number of I/O accesses within a specified time period. For example, let us consider an array H of a file's access frequency; each record in H is logged once every 6 hours. Suppose we have H = {150, 120, 120, 80, 20, 10, 20, 10}, the access-frequency array indicates that the file has a high access frequency on the first day (see the first four records in H) and its access frequency is fairly low on the second day (see the last four records in H).

7 Evaluation

In this section, we first test MFTS's functionalities. Then, we evaluate its performance using workload of real-world applications (e.g., web search).

7.1 Experimental Environment

In our testbed, a metadata server and an MFTS client are built from two server-level machines. The metadata server is a Dell server (PowerEdge R710) with two Intel Xeon E5620 @2.40 GHz (four cores) CPUs, and 8 GB DDR3 RAM. The MFTS client is a SuperMicro server with two Intel Xeon X5560 @2.67 GHz (six cores) CPUs, 12 GB DDR3 RAM, and its memcpy() and XOR speeds are 4.45 and 1.68 GBps, respectively. The operating system running on both the Dell and SuperMicro servers is Fedora 12 $X86\_64$ (Kernel 2.6.32).

There are 12 PC-level machines serving as the storage nodes of MFTS. Each storage node is composed of a 3.2 GHz Intel Core Duo processor (E5800), 2 GB DDR3 memory, and Intel G41 Chipset Mainboard with on board integrated Gigabit Ethernet interface. All disks attached in the storage nodes are WD1002FBYS SATA2. The operating system running on the storage nodes is Linux Ubuntu 10.04 $X86\_64$ (Kernel 2.6.32). All the machines (i.e., storage nodes, a server, and a client) are connected by a WS-C3750G-24TS-S Cisco(R) switch with 24 Ethernet ports.

7.2 Evaluation Methodology

Both MFTS's functionalities and performance are evaluated in our experiments. The functionality tests include reading files in the presence of failed nodes (see Section 7.3.2), rebuilding a failed node (see Section 7.3.3), and upgrading fault-tolerance levels. For example, we follow four steps to test accessing files in degraded mode. First, we write a media file (*abc.avi*) using the *mfts_write* API. Second, we make a systematic node out of operation. Third, we use the *mfts_read* API to read the remaining encoded files and create a new file (*abc_decoded.avi*). Finally, we run the *diff* command to verify that '*abc_decoded.avi*' is identical to the original '*abc.avi*'. We apply the procedures similar to the above four steps to test the functionalities of node rebuilding and fault-tolerance-level transition.

We compare IZS-based MFTS with two storage clusters equipped with the Vandermonde Reed-Solomon (VRS) [20] and the Cauchy Reed-Solomon codes (CRS) [36]; the reason is two-fold. First, the implementation of IZS code relies on the coding algorithm of RS codes; Second, it is unwise to adopt XOR-based RAID codes or three-way replication to implement the MFTS system—as to RAID codes, it is of high I/O overhead to upgrade the 1-fault-tolerant RAID-5 layout to the two-fault-tolerant RAID-6 one due to the specific parity layout of RAID-5 and RAID-6; for the latter, three-way replication has low storage utilization. We implement the *VRS* and *CRS* codes using the *Zfec* [37] and Jerasure [38] libraries, respectively. It is worth noting that Zfec and Jerasure are currently the fastest implementation of VRS and CRS, respectively [39]. It is suggested that a configuration of 'r = 3' can achieve sufficient MTTDL for the archival storage [7]; both four-way replication and (k + 3, k) erasure coding have similar reliability [6]. Thus, we set parameter r to be 3 in the following tests.

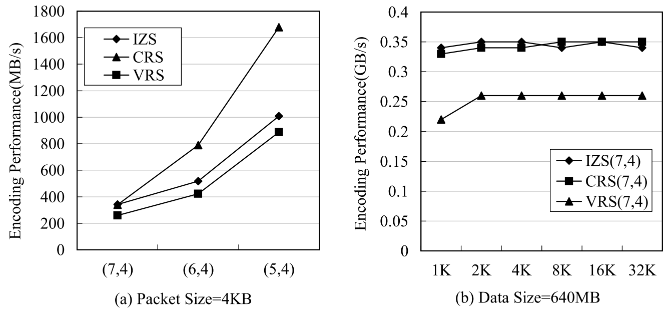


Fig. 7. Data encoding performance.

To evaluate the I/O performance of MFTS, a trace replayer is run on the SuperMicro server, which serves as multiple concurrent users and issues the I/O requests to the *Fault-tolerant Module* according the time-stamp recorded in the workload trace file. The *Dissemination Module* disperses/gathers coded data to/from storage nodes through TCP connections. We use the real-world trace—WebSearch2.spc or Web-2—to drive the trace replayer. Web-2 trace was collected from a popular search engine [40]. Its write ratio is 0.02 percent, IOPS is 298, and average request size is 15.07 KB. We evaluate the online reconstruction performance of MFTS using the Web-2 trace.

There exists an access mode called *synchronized reads* within the node reconstruction. This access mode may result in a throughput collapse problem, which motivates us to address the *TCP-Incast* issue of cluster-based storage systems in general and MFTS in particular. We denote the request unit issued by a spare storage node as *S*torage *R*equest *U*nit (SRU), which contains multiple data blocks. We use average response time as the I/O performance metric, and both reconstruction time and average user response time to evaluate the online reconstruction performance.

7.3 Performance Results

In this study, we mainly focus on both encoding and decoding performances of IZS-based MFTS system, because both encoding and decoding operations dominate all the read/write procedures (e.g., R/W in fail-free case, R/W in degraded mode, and R/W under reconstruction).

7.3.1 Data Encoding Performance

When the MFTS system is about to upgrade the fault-tolerance level, a new parity node should be created as soon as possible. The time spent in creating the parity node is determined by three primary factors: encoding performance, network latency, and storage I/O bandwidth. To evaluate the impact of encoding on the performance, we conduct a set of comparative tests on the SuperMicro server using coding schemes IZS(4 + i, 4), CRS(4 + i, 4) and VRS(4 + i, 4), where i ∈ {1, 2, 3}. Fig. 7 gives the encoding performance. We vary block size to study the sensitivity of the coding schemes to data block size, with $(n, k) = (7, 4)$. Fig. 7b illustrates the

TABLE 6
Encoding Operations Involved in the Three Coding Schemes

Coding	Step-1 (Coefficient Generation)	Step-2 (Linear Combination)
IZS	Algorithm 2 or 6	XOR for parity node C_k; Multiplication for other nodes
CRS	Cauchy matrix	XOR-based Operations
VRS	Vandermonde matrix	Multiplications

encoding performance of three schemes when data block size is varied from 1 to 32 KB.

As shown in Fig. 7a, the encoding performance of IZS anywhere is between those of VRS and CRS when $r \leq 2$. CRS is better than both IZS and VRS because CRS converts expensive multiplications into efficient XOR operations (see Table 6 for operations involved in the three coding schemes). From Fig. 7b, two observations are apparent: First, both IZS(7, 4) and CRS(7, 4) are noticeably better than VRS(7, 4); Second, the block size has sight impact on the encoding performance for a specific (k + 3, k) erasure code.

To measure performance effect of network and storage bandwidth, we use the API *mfts_write* to issue write requests of 640 MB to the storage nodes powered by the various coding schemes. The writing procedure mainly consists of four steps: (1) reading original file data from the Super-Micro server; (2) encoding the data; (3) transferring the encoded data to the storage nodes; and (4) writing the encoded data into the nodes. Fig. 8 shows the results of the writing process powered by the three coding schemes.

Fig. 8 shows that the write performance of IZS is very close to those of CRS and VRS. Time spent in writing data consists of four components: (1) reading, (2) encoding, (3) transferring over the network, and (4) writing to the storage nodes. The encoding time only accounts for a small portion of the total time within the networked storage. For example, in the case of IZS(6, 4), CRS(6, 4), and VRS(6, 4), the encoding time accounts about $5.4 \sim 8.7$ percent. Although CRS (6, 4) has faster encoding performance than that of IZS(6, 4) by a factor of 1.51 (see Fig. 7a), the write performance of CRS(6, 4) is almost the same as that of IZS(6, 4) (i.e., only improved by 2 percent). This observation implies that both network latency and storage bandwidth play a dominated role in affecting the I/O performance of the MFTS system.

7.3.2 User Response Time in Degraded Mode

To support uninterrupted storage services, the MFTS system must be able to decode data stored on a failed node even in the presence of any failed storage node, that is, it

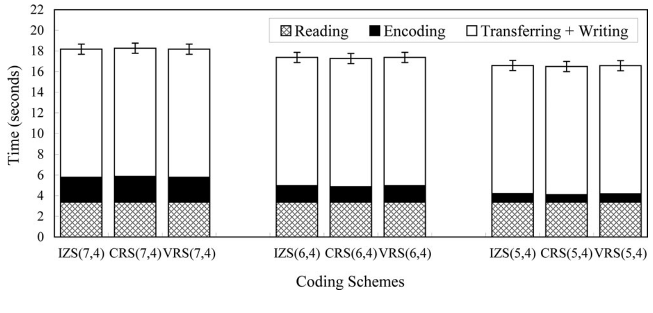


Fig. 8. Write performance of the three coding schemes.

TABLE 7
Average Response Time Under Different Data Block Sizes

Average User Response Time (ms)	Data block size				
	16KB	24KB	32KB	48KB	64KB
IZS(6,4)	5.75	5.75	5.75	5.65	5.58
CRS(6,4)	5.67	5.67	5.66	5.66	5.65
VRS(6,4)	5.68	5.68	5.68	5.68	5.68

can respond to I/O requests before having the failed node reconstructed. We pay attention to the impact of data block size on decoding performance and; thus, we measure average user response time when data block size S_b is set to 16, 24, 32, 48, and 64 KB, respectively. Table 7 shows the average user response times of IZS(6, 4), CRS(6, 4), and VRS (6, 4) in degraded mode when the Web-2 trace is replayed.

Two observations are evident from Table 7. First, the average user response times of the three coding schemes in the degraded mode are very close to each other. Second, the three coding schemes are insensitive to data block size in the networked storage environment (see Section 7.3.1 for the reason).

With respect to decoding, IZS reads fewer elements from surviving storage nodes than CRS or VRS when a request covers more than three elements. The probability distribution of workload indicates that IZS should achieve better performance than CRS and VRS in average user response time. However, our empirical results are inconsistent with the statistic analysis. The inconsistency lies in the fact that read requests directed to a surviving node are not contiguous in IZS, which causes extra disk access latency.

7.3.3 Reconstruction Performance

The reconstruction procedure is closely related to system reliability. Reconstruction schemes fall into two categories—offline and online reconstruction schemes. Under offline reconstruction, storage systems devote all resource to reconstruct failed node(s). In the case of online reconstruction, storage systems can respond to I/O requests issued by foreground applications during the reconstruction process. In this group of experiments, we limit the capacity of each storage node to 10 GB, which is large enough to cover the footprint of the workloads.

We conduct experiments to evaluate both the offline and online reconstruction cases of IZS(6, 4), CRS(6, 4), and VRS (6, 4), with the data block size of 16 KB. The online reconstruction is driven by the Web-2 trace, which has been widely used in the evaluation of online reconstruction techniques [41], [42]. The Web-2 trace represents the read-intensive application. When read requests are located on a failed data node, the required data block will be reconstructed by retrieving $k = 4$ surviving blocks.

Table 8 details average user response time, reconstruction time, and transferred data amount of the coding schemes during the reconstruction process. In this set of experiments, k, r, and RTO are set to 4, 2, and 200 μs, respectively. Here RTO denotes the retransmission timeout of TCP network. The block size and SRU are 16 and 256 KB, respectively. It is shown that IZS outperforms CRS and VRS under offline reconstruction by a factor of 1.261 and 1.271, respectively; and IZS improves the performance of CRS and VRS in the online reconstruction

TABLE 8
Performance of MFTS During the Reconstruction Process, with k = 4, r = 2, RTO = 200 μs, S_b = 16 KB, and SRU = 256 KB

Metric	IZS		CRS		VRS		Speedup (IZS vs. CRS)		Speedup (IZS vs. VRS)	
	offline	online	offline	online	offline	online	offline	online	offline	online
Reconstruction Time (s)	291	409	367	506	370	512	1.261	1.237	1.271	1.252
Average User Response Time (ms)	–	16.8	–	15.4	–	15.9	–	0.917	–	0.946
Total Data Amount (bytes)	30×2^9		40×2^9		40×2^9		1.333		1.333	

case by a factor of 1.237 and 1.252, respectively. The online reconstruction is slower than the offline counterpart (e.g., the reconstruction times of IZS in offline and online cases are 291 and 409 seconds, respectively), because an online reconstruction process must share CPU and I/O bandwidth with applications.

Table 8 also shows that IZS merely increases the response time of CRS and VRS by 9.1 and 5.7 percent, respectively. Such marginal increases are induced by a set of non-contiguous disk reads in IZS, and non-contiguous disk reads give rise to extra disk access latency [43].

7.3.4 Sensitiveness to SRU Size

To evaluate the impact of block size on reconstruction performance, we conduct two sets of experiments to measure reconstruction time and average user response time respectively using fixed SRU (i.e., SRU = 256 KB) and varied SRU (i.e., SRU = $4 \times p \times S_b$). Figs. 9 and 10 plot the results of tests driven by the Web-2 trace with fixed and varied SRU, respectively.

From Table 7 and Fig. 9a, it is observed that the average response times in the degraded mode is less than those in the online reconstruction mode, because the storage and network bandwidth are shared by both the user and reconstruction I/O requests in the online case.

Fig. 9 shows that if SRU is fixed, reconstruction time and average user response time are sightly sensitive to data block size. For example, the average response time of CRS (6, 4) varies from 15.4 to 16.0 ms; the reconstruction time varies from 490 to 506 seconds. In contrast, Fig. 10 shows that if SRU is changing, the reconstruction time and average response time are rather sensitive to the data block size. For instance, when the block size goes up, the reconstruction time decreases, whereas the response time increases. The average response time increases because the reconstruction process consumes significant network and storage bandwidth when SRU is large.

Figs. 9a and 10a indicate that IZS has slightly longer average user response time than those of CRS and VRS. As mentioned in Section 7.3.3, the reason is that the IZS coding leads to some non-contiguous disk reads.

Figs. 9b and 10b show that IZS improves the reconstruction performance over CRS by anywhere between 23.7 and 30.6 percent, and IZS improves the reconstruction performance over VRS by anywhere between 25.2 and 32.6 percent. The reason why IZS can reduce reconstruction time is that IZS merely needs to read 30 GB data from surviving storage nodes. Note that 30 GB is $3/4$ of the data volume accessed by CRS or VRS (see Eq. (3)).

7.3.5 Impact of TCP-Incast

Recall that MFTS is a clustered storage system, where storage nodes are connected via Ethernet/IP switches. In this experiment, we study the *TCP Incast congestion* issue. When rebuilding a failed storage node in MFTS, data packets (i.e., SRU) are fetched in parallel from surviving nodes. The subsequent rebuilding-read request will not be issued until the rebuilding node has received all the data packets for the current request. Phanishayee et al. discovered that such *synchronized reads* over TCP/IP network may result in the *Incast* problem [44].

We measure the goodput performance (i.e., throughput observed by applications) of MFTS when the number of storage nodes is varied from 1 to 12, and the block size is set to 64 KB; the SRU size is the product of block size and number of rows (e.g., 256 KB = 64 KB $\times$ 4). Fig. 11 shows that when RTO is 200 ms, there is a throughput collapse where the goodput sharply drops when the number of storage node is 6. However, the *Incast* impact can be mitigated by adopting high-resolution retransmission (e.g., RTO = 200 μ) [45].

Table 9 illustrates the impact of parameter k on reconstruction speed of IZS, CRS, and VRS. The reconstruction speed is calculated as the total amount of reconstructed data divided by reconstruction time. In this experiment, k is set to 4, 6, and 8, respectively; with parameters $r = 2$, RTO = 200 ms, and IOPS = 150. We observe that the *Incast*

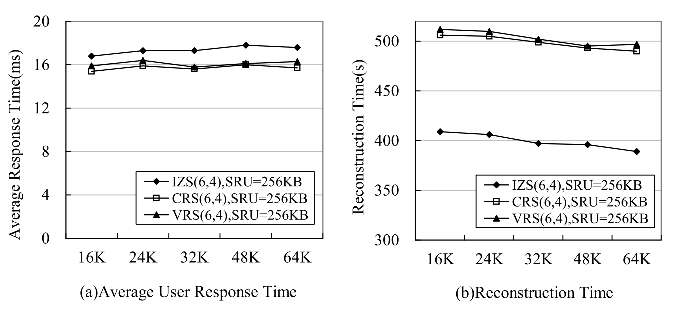


Fig. 9. Impact of block size (S_b) on reconstruction times and average response times, with a fixed SRU (SRU = 256 KB).

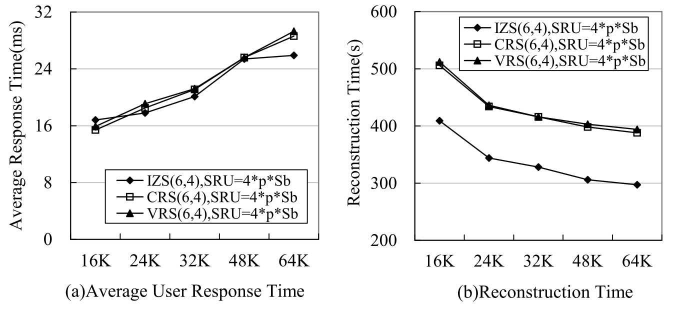


Fig. 10. Impact of packet size (S_b) on reconstruction times and average response times, with varied SRUs ($\mathrm{SRU} = 4 \times p \times S_b$).

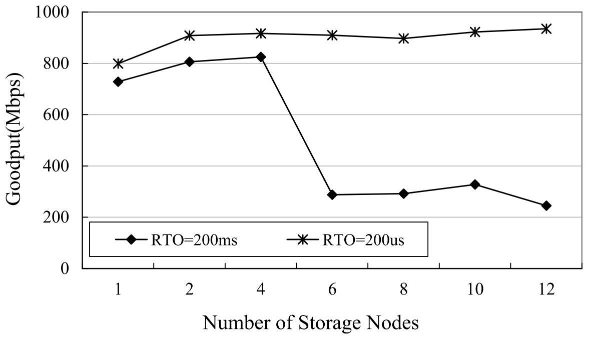


Fig. 11. Impact of the storage node number on goodput, with RTO = 200 ms and 200 μs, and IOPS = 150.

problem can deteriorate the reconstruction performance of all the three schemes. The parameter k significantly affects reconstruction speed in the *Incast* scenario. For example, the reconstruction speeds of VRS(6, 4), VRS(8, 6) and VRS(10, 8) are 839.0, 385.8, and 308.4 Mbps, respectively.

The reconstruction-performance improvements of IZS over CRS and VRS become more prominent as k increases. For instance, IZS speeds up the reconstruction time over CRS by a factor of 122.6, 202.5, and 224.0 percent when k is 4, 6, and 8, respectively. Here 122.6% = 0.92 × (4/3), 4/3 is the data-amount ratio when k = 4 and r = 2 (see Eq. (3)). The *Incast* problem is caused by insufficient buffer capacity in switches to handle large influx of packets in many-to-one communications. Compared with the VRS and CRS codes, IZS reduces the number of accessed data blocks, thereby weakening the requirement of buffer space in switches and optimizing the *synchronized reads* potentially.

7.4 A Summary of Observations

Important observations drawn from the above experimental results are summarized as follows:

- The write performance of IZS is close to those of CRS and VRS; both network latency and storage bandwidth play a dominated role in write performance for networked reliable storage systems;
- The average response times of the three coding schemes (i.e., CRS, VRS, and IZS) in the degraded mode are very close to each other;
- IZS outperforms CRS and VRS in the offline reconstruction case by a factor of 1.261 and 1.271, respectively; and IZS improves the performance of CRS and VRS in the online reconstruction case by a factor of 1.237 and 1.252, respectively;
- The reconstruction-performance improvements of IZS over CRS and VRS become more prominent in the TCP-*Incast* scenario than in the non-*Incast* scenario.

8 Conclusions and Future Work

To meet various reliability requirements of applications, we design a multi-level fault-tolerant storage system called MFTS to offer multiple fault-tolerance levels at run time. The MFTS system employs the Reed-Solomon-like codes (IZS codes) with optimized maintenance bandwidth. MFTS

TABLE 9
Reconstruction Speed Under Different Data Node Numbers, with RTO = 200 ms and IOPS = 150

Reconstruction Speed (Mbps)	Number of information nodes (i.e., k)		
	4	6	8
IZS(k+2,k)	783.6	571.4	495.6
CRS(k+2,k)	852.3	423.5	354.9
VRS(k+2,k)	839.0	385.8	308.4
Speedup (IZS vs. CRS)	0.92	1.35	1.40
Speedup (IZS vs. VRS)	0.93	1.48	1.61

has three salient features. First, it enables mapping of the most appropriate fault-tolerant schemes to applications with individual reliability requirements. Second, it dynamically adjusts fault-tolerance levels if an application's reliability requirement is changing. Fault-tolerance levels are upgraded or downgraded in MFTS by simply constructing or discarding parity nodes. Third, it improves the maintenance performance, which reduces time spent in reconstructing any failed storage node, modifying the archived data, and altering fault-tolerance levels.

MFTS achieves good maintenance performance owing to three techniques. First, the archived data are modified through partial updates, which makes the updating overhead in MFTS very small. Second, the node reconstruction is performed through exact repairs, which enable the reconstruction traffic to be minimized. Third, the fault-tolerance level is altered by constructing or disabling only parity nodes. The experimental results also validate the performance optimization—MFTS can significantly improve the reconstruction performance in both offline and online reconstruction cases, and the improvement becomes more prominent in the TCP-*Incast* scenario.

We implemented the MFTS system in a small-scale storage cluster. We also addressed potential challenges in applying MFTS to large-scale networked storage systems. For example, *synchronized reads* may lead to the *Incast* problem; both network and storage bandwidths rather than encoding/decoding speed dominate the I/O performance of large-scale clusters. In the future, we will evaluate the MFTS system in a large-scale storage cluster, where the number k of data nodes is greater than 16 [6], [7].

Acknowledgments

The authors would like to thank to the anonymous reviewers for their suggestions on the earlier version of this manuscript. This work was supported in part by the National Basic Research Program of China under Grant No.2011CB302303, and National High Technology Research Program of China under Grant No. 2013AA013203. This work was also supported by Key Laboratory of Information Storage System, Ministry of Education. Xiao Qin and Wei-Shinn Ku's research was supported by the US National Science Foundation under Grants CCF-0845257(CAREER), CNS-0917137 (CSR), and CCF-0742187(CPA). Changsheng Xie is the corresponding author of this paper.

References

[1] S. Ghemawat, H. Gobioff, and S. Leung, "The google file system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.
[2] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Proj. Website* 2007.

[3] J. Plank, "Erasure codes for storage applications," presented at the 4th Usenix Conf. File and Storage Technologies–Tutorial Slides, San Jose, CA, USA, 2005.

[4] J. L. Hafner, "WEAVER codes: Highly fault tolerant erasure codes for storage systems," in *Proc. 4th USENIX Conf. File Storage Technol.*, 2005, pp. 211–224.

[5] 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," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 190–201, 2000.

[6] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch, "A decentralized algorithm for erasure-coded virtual disks," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2004, pp. 125–134.

[7] M. Storer, K. Greenan, E. Miller, and K. Voruganti, "Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, p. 1.

[8] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, Boston, MA, USA, 2012, p. 2.

[9] S. Gopisetty, S. Agarwala, E. Butler, D. Jadav, S. Jaquet, M. Korupolu, R. Routray, P. Sarkar, A. Singh, M. Sivan-Zimet, C.-H. Tan, S. Uttamchandani, D. Merbach, S. Padbidri, A. Dieberger, E. M. Haber, E. Kandogan, C. A. Kieliszewski, D. Agrawal, M. Devarakonda, K.-W. Lee, K. Magoutis, D. C. Verma, and N. G. Vogl, "Evolution of storage management: Transforming raw data into information," *IBM J. Res. Develop.*, vol. 52, no. 4, pp. 341–352, 2008.

[10] J. Wong, "Mirror file system (MFS). A multiple server file system," in *Proc. 6th USENIX Conf. File Storage Technol. (FAST '08)*, 2008.

[11] I. Tamo, Z. Wang, and J. Bruck, "MDS array codes with optimal rebuilding," in *Proc. IEEE Int. Symp. Inf. Theory*, 2011, pp. 1240–1244.

[12] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP auto-RAID hierarchical storage system," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 108–136, Feb. 1996.

[13] K. Gopinath, N. Muppalaneni, N. Kumar, and P. Risbood, "A 3-tier RAID storage system with RAID1, RAID5 and compressed RAID5 for linux," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, 2000, pp. 30–44.

[14] N. Joukov, A. Krishnakumar, C. Patti, A. Rai, S. Satnur, A. Traeger, and E. Zadok, "RAIF: Redundant array of independent filesystems," in *Proc. 24th IEEE Conf. Mass Storage Syst. Technol.*, 2007, pp. 199–214.

[15] C. Yalamanchili, K. Vijayasankar, E. Zadok, and G. Sivathanu, "DHIS: Discriminating hierarchical storage," in *Proc. SYSTOR 2009: The Israeli Exp. Syst. Conf.*, 2009, pp. 9–21.

[16] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 17–33.

[17] J. S. Plank, "A tutorial on reed-solomon coding for fault-tolerance in raid-like systems," *Softw. Pract. Exp.*, vol. 27, no. 9, pp. 995–1012, 1997.

[18] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "DiskReduce: RAID for data-intensive scalable computing," in *Proc. 4th Annu. Workshop Petascale Data Storage*, 2009, pp. 6–10.

[19] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Proc. 1st Int. Workshop Peer-to-Peer Syst.*, 2001, pp. 328–337.

[20] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.

[21] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 2, pp. 24–36, 1997.

[22] J. S. Plank and L. Xu, "Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications," in *Proc. 5th IEEE Int. Symp. Netw. Comput. Appl.*, 2006, pp. 173–180.

[23] Y. Wu, A. Dimakis, and K. Ramchandran, "Deterministic regenerating codes for distributed storage," presented at the 45th Annu. Allerton Conf. Control, Computing, and Communication, Urbana-Champaign, IL, USA, 2007.

[24] K. Rashmi, N. B. Shah, and P. V. Kumar, "Enabling node repair in any erasure code for distributed storage," in *Proc. IEEE Int. Symp. Inf. Theory*, 2011, pp. 1235–1239.

[25] S. Ramabhadran and J. Pasquale, "Analysis of long-running replicated systems," in *Proc. 25th IEEE Int. Conf. Comput. Commun.*, 2006, pp. 1–9.

[26] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson, "RAID: High-performance, reliable secondary storage," *ACM Comput. Surv.*, vol. 26, no. 2, pp. 145–185, 1994.

[27] J. Elerath and M. Pecht, "Enhanced reliability modeling of RAID storage systems," in *Proc. 37th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2007, pp. 175–184.

[28] K. Rao, J. Hafner, and R. Golding, "Reliability for networked storage nodes," *IEEE Trans. Dependable Secure Comput.*, vol. 8, no. 3, pp. 404–418, May 2011.

[29] T. Schwarz, "Reliability and performance of disk arrays," Ph.D. dissertation, Dept. Comput. Sci., UC San Diego, La Jolla, CA, 1994.

[30] S. Lin, C. Zhang, G. Wang, X. Liu, and J. Liu, "Reliability analysis for full-2 code," in *Proc. 10th Int. Symp. Pervasive Syst., Algorithms, Netw.*, 2009, pp. 454–459.

[31] J. Hafner and K. Rao, "Notes on reliability models for non-MDS erasure codes," IBM, Armonk, NY, USA, IBM Res. Rep. RJ10391, 2006.

[32] Seagate, "Seagate barracuda 7200.11 product overview," http://www.seagate.com/staticfiles/support/disc/manuals/ce/SV35 Series/ SV35.5~Series/100562053b.pdf, 2011.

[33] B. Schroeder and G. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" in *Proc. 5th USENIX Conf. File Storage Technol.*, 2007, pp. 1–16.

[34] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky, "Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics," *ACM Trans. Storage*, vol. 4, no. 3, p. 7, Nov. 2008.

[35] M. Szeredi. (2008). *Fuse: Filesystem in userspace* [Online]. Available: http://fuse.sourceforge.net.

[36] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An XOR-based erasure-resilient coding scheme," Int. Comput. Sci. Inst., Berkeley, CA, USA, Tech. Rep. ICSI TR-95-048, 1995.

[37] Z. Wilcox-O'Hearn. (2011). *Zfec 1.4.2. open source code distribution* [Online]. Available: http://pypi.python.org/pypi/zfec.

[38] J. Plank, S. Simmerman, and C. D. Schuman. (Aug. 2008). *Jerasure: A library in C/C++ facilitating erasure coding for storage applications – V1.2* [Online]. Available: http://www.cs.utk.edu/~plank/plank/papers/CS-08-627.html.

[39] J. Plank, J. Luo, C. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proc. 7th Conf. File Storage Technol.*, 2009, pp. 253–265.

[40] O. Application. (2007). *I/o and search engine i/o. umass trace repository* [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage.

[41] S. Wu, H. Jiang, D. Feng, L. Tian, and B. Mao, "Workout: I/o workload outsourcing for boosting raid reconstruction performance," in *Proc. 7th Conf. File Storage Technol.*, 2009, pp. 239–252.

[42] S. Wan, Q. Cao, J. Huang, S. Li, X. Li, S. Zhan, L. Yu, C. Xie, and X. He, "Victim disk first: An asymmetric cache to boost the performance of disk arrays under faulty conditions," in *Proc. USENIX Conf. USENIX Annu. Techn. Conf.*, 2011, p. 13.

[43] Y. Deng, "What is the future of disk drives, death or rebirth?" *ACM Comput. Surv.*, vol. 43, no. 3, pp. 1–27, 2011.

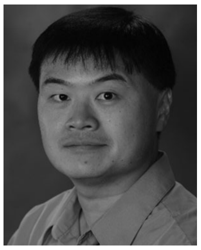
[44] A. Phanishayee, E. Krevat, V. Vasudevan, D. Andersen, G. Ganger, G. Gibson, and S. Seshan, "Measurement and analysis of TCP throughput collapse in cluster-based storage systems," in *Proc. Sixth USENIX Conf. File Storage Technol.*, 2008, p. 12.

[45] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 303–314, 2009.

Jianzhong Huang received the PhD degree in computer architecture in 2005 and completed the post doctoral research in information engineering in 2007 from the Huazhong University of Science and Technology(HUST), Wuhan, China. He is currently an associate professor in the Wuhan National Laboratory for Optoelectronics at HUST. His research interests include computer architecture and dependable storage systems. He received the National Science Foundation of China in Storage System Research Award in 2007. He is a member of the China Computer Federation (CCF).

Xiao Qin (S'00-M'04-SM'09) received the BS and MS degrees in computer science from HUST, China, and the PhD degree in computer science from the University of Nebraska-Lincoln, in 1992, 1999, and 2004, respectively. He is currently an associate professor with the Department of Computer Science and Software Engineering, Auburn University. His research interests include parallel and distributed systems, storage systems, fault tolerance, real-time systems, and performance evaluation. He received the US National Science Foundation (NSF) Computing Processes and Artifacts Award and the NSF Computer System Research Award in 2007 and the NSF CAREER Award in 2009. He is a senior member of the IEEE.

Fenghao Zhang received both the BS and MS degrees in computer science and technology from HUST in 2010 and 2013, respectively. His research interests include networked storage systems and file system.



Wei-Shinn Ku (S'02-M'07-SM'12) received the PhD degree in computer science from the University of Southern California (USC) in 2007. He received both the MS degree in computer science and the MS degree in electrical engineering from USC in 2003 and 2006, respectively. He is currently an associate professor with the Department of Computer Science and Software Engineering at Auburn University. His research interests include data management systems, data analytics, geographic information systems, mobile computing, and information security. He is a senior member of the IEEE.

Changsheng Xie received the BS and MS degrees in computer science both from HUST, Wuhan, China, in 1982 and 1988, respectively. He is currently a professor in the Department of Computer Engineering at HUST. He is also the director of the Data Storage Systems Laboratory of HUST and the deputy director of the Wuhan National Laboratory for Optoelectronics. His research interests include computer architecture, I /O system, and networked storage system. He is the vice chair of the expert committee of Storage Networking Industry Association (SNIA), China. He is a member of the IEEE.