INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING

ICPP/2020/EDMONTON/CANADA

AUGUST 17-20, 2020

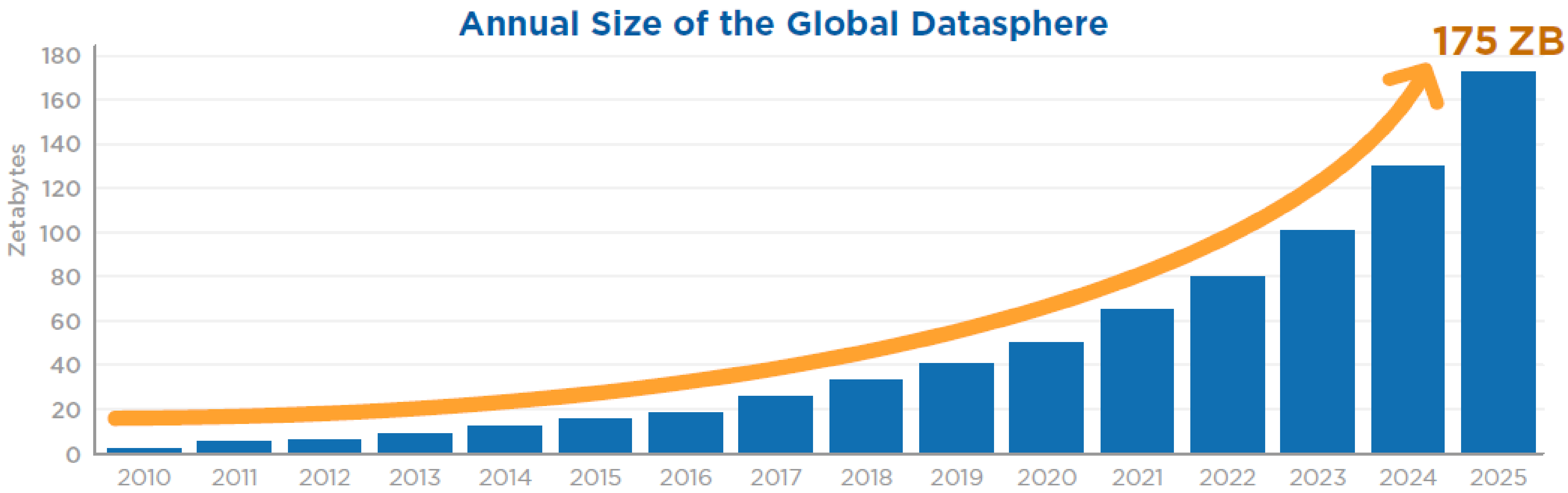
An Adaptive Erasure-Coded Storage Scheme with an Efficient Code-Switching Algorithm

Zizhong Wang, Haixia Wang, Airan Shao, and Dongsheng Wang

Tsinghua University

Really Big Data - Present and Future

Figure 1 – Annual Size of the Global Datasphere



1 ZB = 1,180,591,620,717,411,303,424 B

175 ZB = 206,603,533,625,546,978,099,200 B

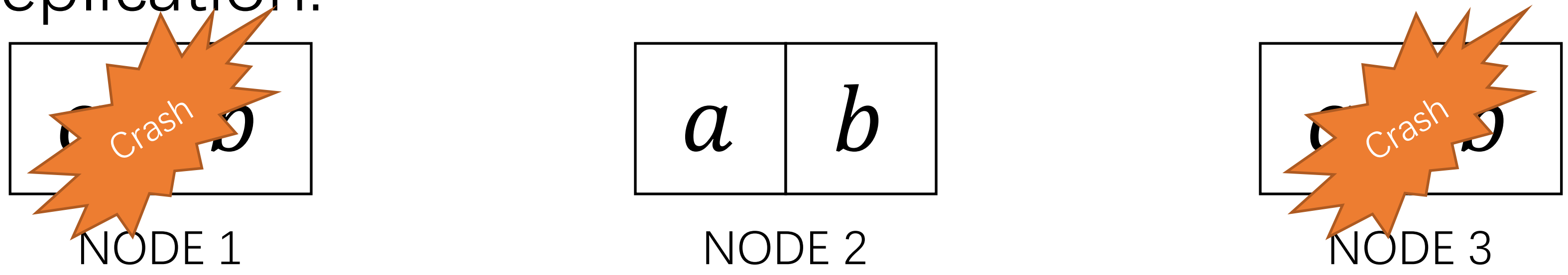
https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf

Distributed Storage Systems

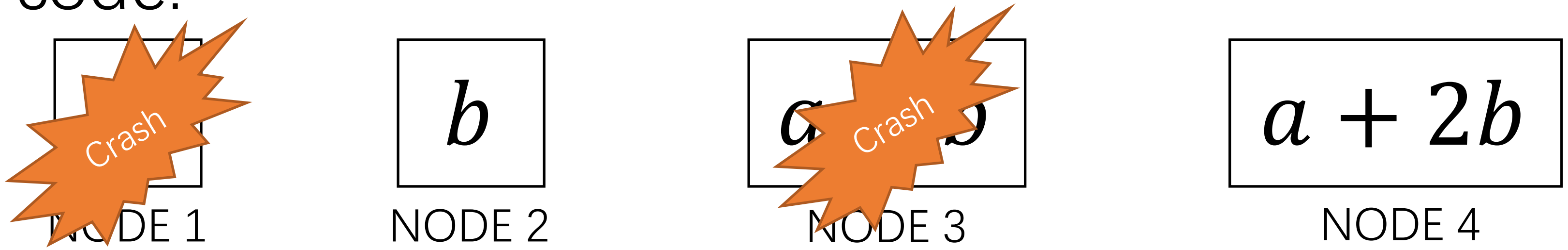
- How to guarantee reliability and availability?
- N-way replication
  - GFS (3-way)
  - N× storage cost to tolerate any (N-1) faults
    - Too **expensive**, especially when data amount grows fast
  - Simple, still the default setting in HDFS, Ceph
- Erasure coding
  - HDFS (since 3.0.0), Azure, Ceph
  - A (k,m) code can tolerate any m faults at a (1+m/k)× storage cost
    - Can save much storage space

An Example of Erasure Coding

- 3-way replication vs a (2,2) code, original data: | a | b |
- 3-way replication:



- a (2,2) code:



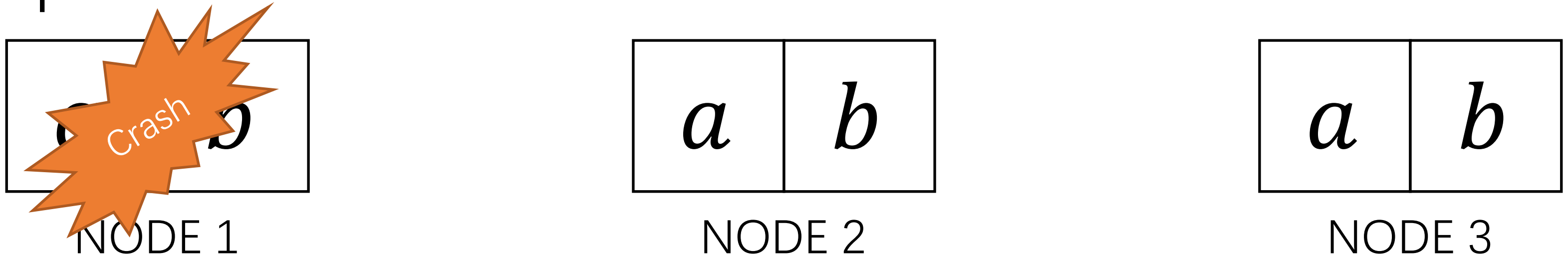
- They both can tolerate any 2 faults, but 3-way replication costs 3× storage space while the (2,2) code costs only 2×

Erasure Coding – What do We Concern?

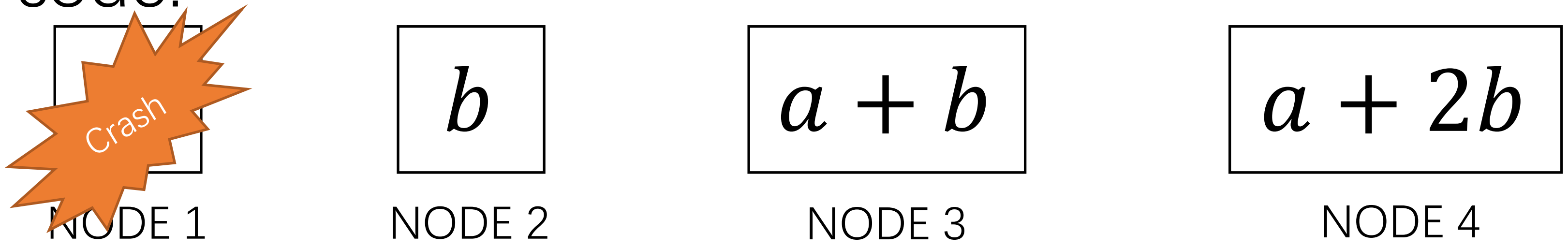
- Storage cost
  - In a (k,m) code: $(1+m/k)\times$
- Fault tolerance ability
  - In a (k,m) code: m
- Recovery cost
  - Discuss later
- Write performance
  - Correlated with storage cost
  - Hard-sell advertising: in asynchronous situation, can use CRaft ([FAST '20] Wang et al.)
- Update performance
- …

Major Concern: Recovery Cost

- 3-way replication:



- a (2,2) code:

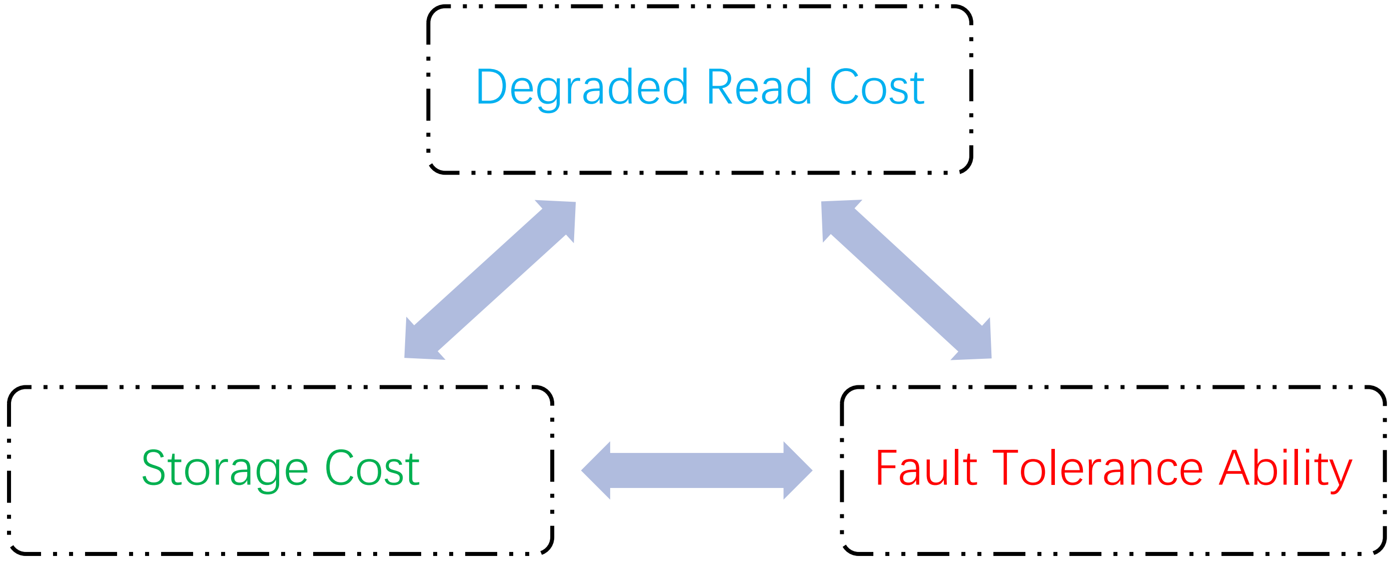


- Conclusion: k times recovery cost in (k,m) code

Degraded Read

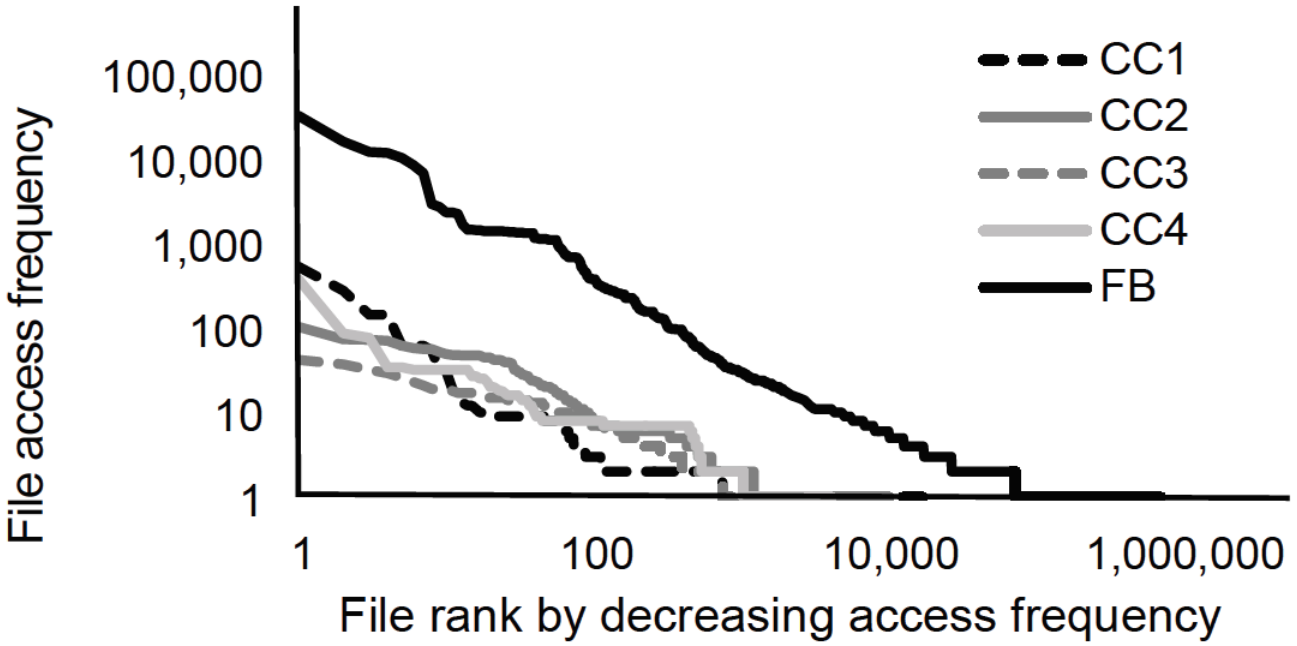
- > 90% data center errors are temporary errors ([OSDI '10] Ford et al.)
  - No data are permanently lost
  - Solved by degraded reads
    - Read from other nodes and then decode
- Our goal: reduce degraded read cost

Trade-Offs



- Different code families
  - MDS/non-MDS, locality, …
- Different parameters
  - small k + small m/k
    - low degraded read cost and storage cost, but low fault tolerance ability
  - small k + big m
    - low degraded read cost, high fault tolerance ability, but high storage cost
  - small m/k + big m
    - low storage cost, high fault tolerance ability, but high degrade read cost

Data Access Skew



Data access frequency is Zipf distribution

About 80% data accesses are applied in 10% data volume

[VLDB '12] Chen et al.

Divide and Conquer

- Premise: guaranteed fault tolerance ability
- Hot data – degraded read cost is most important
- Cold data – storage cost is most important

- Data with different properties should be stored by different codes
  - A fast code for hot data
    - Low degraded read cost and high enough fault tolerance ability
    - High storage cost is acceptable
  - A compact code for cold data
    - Low storage cost and high enough fault tolerance ability
    - High degraded read cost is acceptable

Code-Switching Problem

- According to temporal locality, hot data will become cold
  - Cold data may become hot in some cases
- Problem: code-switching from one code to another code

a_1	a_2	a_3	a_4	a_5	a_6	$f_1(a)$	$f_2(a)$

a_1	a_2	a_3	a_4	a_5	a_6	$f_3(a)$	$f_4(a)$

- To compute $f_3(a)$ and $f_4(a)$, a should be collected first
  - Bandwidth-consuming

Alleviate the Problem

- HACFS ([FAST '15] Xia et al.)
  - Use two codes in the same code family with different parameters
  - Alleviate the code-switching problem by using the similarity in one code family
  - Cannot take advantage of the trade-off in different code families
  - Cannot get rid of the code family's inherent defects
    - Impossible to set an MDS compact code
- Our Scheme
  - We present an efficient code-switching algorithm

Our Scheme

- We choose Local Reconstruction Code (LRC) as fast code, Hitchhiker (HH) as compact code
  - (k,m-1,m)-LRC and (k,m)-HH
- Reasons

1. LRC has good fast code properties
   - Good locality
2. HH has good compact code properties
   - MDS
3. Common. Been implemented in HDFS or Ceph
4. They are similar. Both based on RS; data chunks be grouped

LRC

- Fast code

- An example of (6,2,3)-LRC

a_1	a_2	a_3	a_4	a_5	a_6
b_1	b_2	b_3	b_4	b_5	b_6

$f_1(a)$	$f_2(a)$	$f_3(a)$	$a_1 \oplus a_2 \oplus a_3$	$a_4 \oplus a_5 \oplus a_6$
$f_1(b)$	$f_2(b)$	$f_3(b)$	$b_1 \oplus b_2 \oplus b_3$	$b_4 \oplus b_5 \oplus b_6$

HH

- Compact code

- An example of (6,3)-HH

a_1	a_2	a_3	a_4	a_5	a_6
b_1	b_2	b_3	b_4	b_5	b_6

$f_1(a)$	$f_2(a)$	$f_3(a)$
$f_1(b)$	$f_2(b) \oplus a_1 \oplus a_2 \oplus a_3$	$f_3(b) \oplus a_4 \oplus a_5 \oplus a_6$

Scheme I

LRC → HH

a_1	a_2	a_3	a_4	a_5	a_6
b_1	b_2	b_3	b_4	b_5	b_6

$f_1(a)$	$f_2(a)$	$f_3(a)$	$a_1 \oplus a_2 \oplus a_3$	$a_4 \oplus a_5 \oplus a_6$
$f_1(b)$	$f_2(b)$	$f_3(b)$	$b_1 \oplus b_2 \oplus b_3$	$b_4 \oplus b_5 \oplus b_6$

a_1	a_2	a_3	a_4	a_5	a_6
b_1	b_2	b_3	b_4	b_5	b_6

$f_1(a)$	$f_2(a)$	$f_3(a)$
$f_1(b)$	$f_2(b) \oplus a_1 \oplus a_2 \oplus a_3$	$f_3(b) \oplus a_4 \oplus a_5 \oplus a_6$

a_1	a_2	a_3	$a_1 \oplus a_2 \oplus a_3$
b_1	b_2	b_3	$b_1 \oplus b_2 \oplus b_3$

a_4	a_5	a_6	$a_4 \oplus a_5 \oplus a_6$
b_4	b_5	b_6	$b_4 \oplus b_5 \oplus b_6$

$f_1(a)$	$f_2(a)$	$f_3(a)$
$f_1(b)$	$f_2(b) \oplus a_1 \oplus a_2 \oplus a_3$	$f_3(b) \oplus a_4 \oplus a_5 \oplus a_6$

a_1	a_2	a_3	a_4	a_5	a_6
b_1	b_2	b_3	b_4	b_5	b_6

Scheme I

HH → LRC

$f_1(a)$	$f_2(a)$	$f_3(a)$	$a_1 \oplus a_2 \oplus a_3$	$a_4 \oplus a_5 \oplus a_6$
$f_1(b)$	$f_2(b)$	$f_3(b)$	$b_1 \oplus b_2 \oplus b_3$	$b_4 \oplus b_5 \oplus b_6$

A New Scheme

- When HH uses XOR sum of data chunks as the first parity chunk, a global parity chunk of LRC can be saved
  - (k,m-1,m-1)-LRC and (k,m)-HH

a_1	a_2	a_3	a_4	a_5	a_6
b_1	b_2	b_3	b_4	b_5	b_6

$f_2(a)$	$f_3(a)$	$a_1 \oplus a_2 \oplus a_3$	$a_4 \oplus a_5 \oplus a_6$
$f_2(b)$	$f_3(b)$	$b_1 \oplus b_2 \oplus b_3$	$b_4 \oplus b_5 \oplus b_6$

(6,2,2)-LRC

a_1	a_2	a_3	a_4	a_5	a_6	$a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6$
b_1	b_2	b_3	b_4	b_5	b_6	$b_1 \oplus b_2 \oplus b_3 \oplus b_4 \oplus b_5 \oplus b_6$

$f_2(a)$	$f_3(a)$
$f_2(b) \oplus a_1 \oplus a_2 \oplus a_3$	$f_3(b) \oplus a_4 \oplus a_5 \oplus a_6$

(6.3)-HH

Scheme II
LRC → HH

a_1	a_2	a_3	a_4	a_5	a_6
b_1	b_2	b_3	b_4	b_5	b_6

$f_2(a)$	$f_3(a)$	$a_1 \oplus a_2 \oplus a_3$	$\oplus a_4 \oplus a_5 \oplus a_6$
$f_2(b)$	$f_3(b)$	$b_1 \oplus b_2 \oplus b_3$	$\oplus b_4 \oplus b_5 \oplus b_6$

a_1	a_2	a_3	a_4	a_5	a_6	$a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6$
b_1	b_2	b_3	b_4	b_5	b_6	$b_1 \oplus b_2 \oplus b_3 \oplus b_4 \oplus b_5 \oplus b_6$

$f_2(a)$	$f_3(a)$
$f_2(b) \oplus a_1 \oplus a_2 \oplus a_3$	$f_3(b) \oplus a_4 \oplus a_5 \oplus a_6$

a_1	a_2	a_3	$a_1 \oplus a_2 \oplus a_3$
b_1	b_2	b_3	$b_1 \oplus b_2 \oplus b_3$

a_4	a_5	a_6	$a_4 \oplus a_5 \oplus a_6$
b_4	b_5	b_6	$b_4 \oplus b_5 \oplus b_6$

$f_2(a)$	$f_3(a)$
$f_2(b) \oplus a_1 \oplus a_2 \oplus a_3$	$f_3(b) \oplus a_4 \oplus a_5 \oplus a_6$

a_1	a_2	a_3	a_4	a_5	a_6
b_1	b_2	b_3	b_4	b_5	b_6

Scheme II

HH → LRC

$f_2(a)$	$f_3(a)$	$a_1 \oplus a_2 \oplus a_3$	$a_4 \oplus a_5 \oplus a_6$
$f_2(b)$	$f_3(b)$	$b_1 \oplus b_2 \oplus b_3$	$b_4 \oplus b_5 \oplus b_6$

Performance Analysis

Table 3: Different Schemes' Degraded Read Cost and MTTF

Scheme	Degraded read cost	MTTF (years)
Scheme I	4.44	$\mathbf{1.0 \times 10^{14}}$
Scheme II	**4.3**	$\mathbf{1.0 \times 10^{14}}$
(10, 4)-RS code	10	6.7×10^{13}
(10, 4)-HH code	6.7	9.3×10^{13}
(12, 2, 3)-LRC	6	6.3×10^{13}
3-replication		3.5×10^{9}
4-replication		7.7×10^{13}

Code-Switching Efficiency

- **Ratio I:**

the amount of data transferred during code-switching

to

the amount of data transferred during encoding

Table 4: Ratio I and Ratio II in Different Schemes

Different schemes	Ratio I	Ratio II
(12, 3, 4)-LRC + (12, 4)-HH code with the re-encoding algorithm	0.790	2.125
(12, 3, 3)-LRC + (12, 4)-HH code with the re-encoding algorithm	0.889	2.063
3-replication + (12, 4)-HH code	0.361	3.063
HACFS-PC	0.333	1.714
HACFS-LRC	0.300	1.625
Scheme I	**0.079**	**1.281**
Scheme II	0.194	1.344

Code-Switching Efficiency

- **Ratio II:**

the total amount of data transferred during encoding to hot data form and switching into cold data form

to

the amount of data transferred when directly encoding into cold data form

Table 4: Ratio I and Ratio II in Different Schemes

Different schemes	Ratio I	Ratio II
(12, 3, 4)-LRC + (12, 4)-HH code with the re-encoding algorithm	0.790	2.125
(12, 3, 3)-LRC + (12, 4)-HH code with the re-encoding algorithm	0.889	2.063
3-replication + (12, 4)-HH code	0.361	3.063
HACFS-PC	0.333	1.714
HACFS-LRC	0.300	1.625
Scheme I	**0.079**	**1.281**
Scheme II	0.194	1.344

Experiment Setup

- (k,m)=(12,4)
  - (12,3,4)-LRC and (12,4)-HH (Scheme I)
  - (12,3,3)-LRC and (12,4)-HH (Scheme II)
- Storage overhead set to 1.4×
- Schemes implemented upon Ceph
- Workload generated randomly, data access frequency set to be Zipf distributed

Recovery Cost

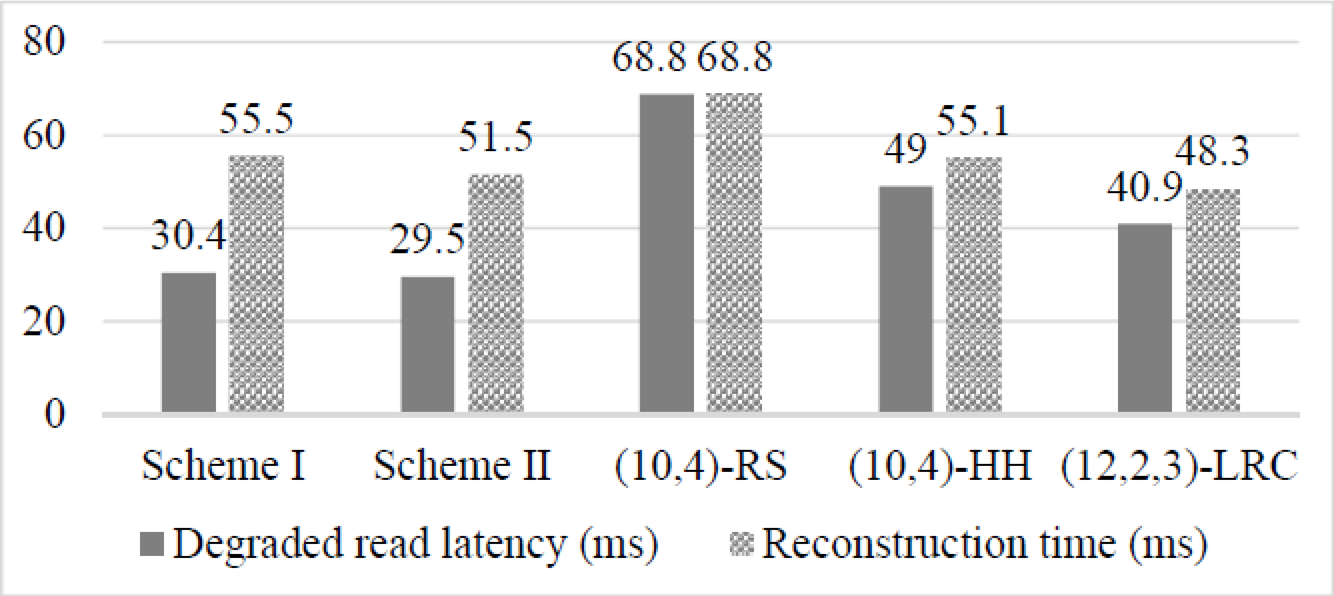


Figure 10: Time usage for different schemes to recover 1 MB data.

Code-Switching Time

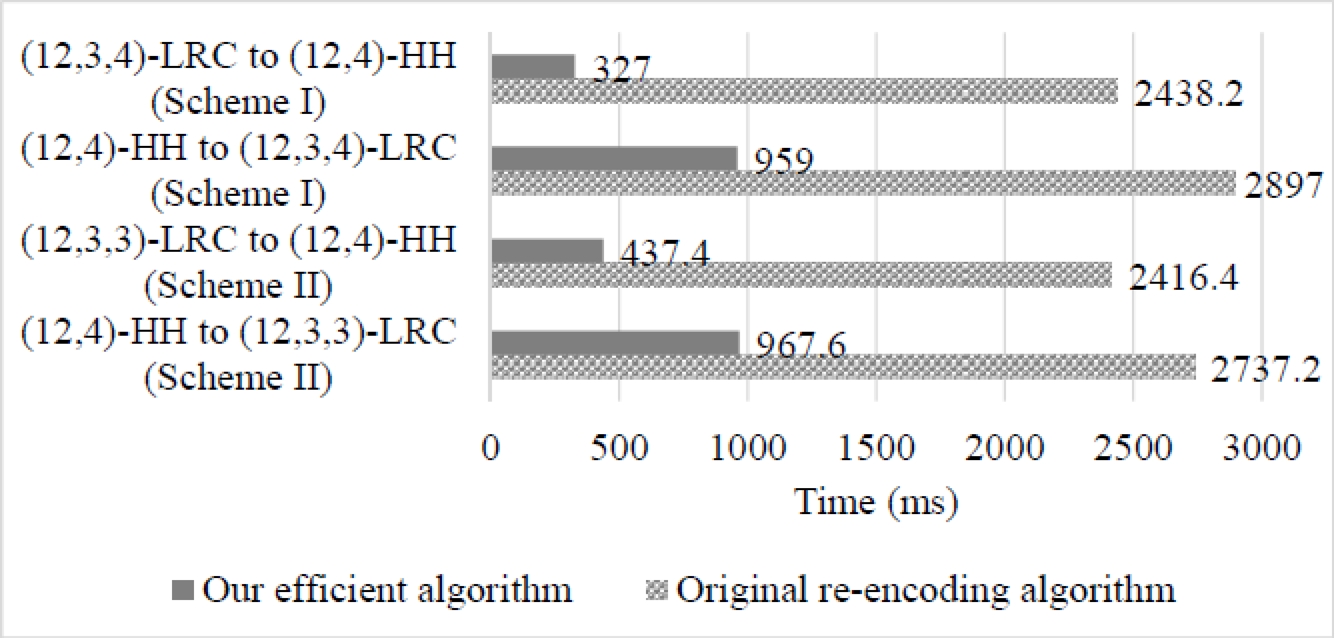


Figure 11: Time usage for switching codes by different algorithms.

Future Works

- More detailed evaluations
  - Actual traces
  - Implemented in Ceph
- More parameter choices
  - Combining our scheme with HACFS-LRC
- More code family choices
  - MSR and MBR?

An Adaptive Erasure-Coded Storage Scheme with an Efficient Code-Switching Algorithm

Zizhong Wang, Haixia Wang, Airan Shao, and Dongsheng Wang
Tsinghua University

Thank you!

wds@tsinghua.edu.cn
wangzizhong13@tsinghua.org.cn