BUTTERFLY

OPENING THE CHRYSALIS: ON THE REAL REPAIR PERFORMANCE OF MSR CODES

Presented by: Matan Liram

Table of Contents

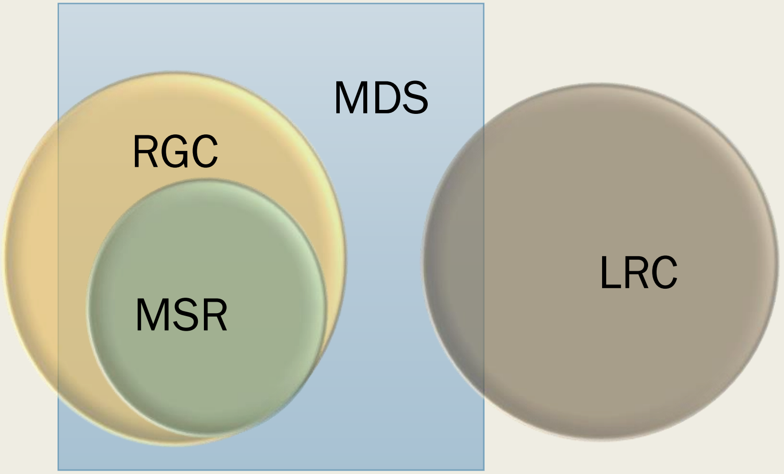
Motivation

- Large distributed storage systems use erasure codes to reliably store data
- Erasure codes reduce storage overhead
- But, for repairing a lost disk, common codes require **reading all data disks**, and **transferring** them through network
- In Facebook, **each day** ~20-100 nodes, 15TB each, fail

Motivation

- Theoretically, MSR (Minimum Storage Regenerating) codes optimally reduce this repair burden, as we have seen in Zig-Zag
- MSR codes have **not** been **implemented** in real-world distributed storage systems
- In the paper, they show how to vertically **integrate butterfly with the storage system**, resulting in good performance

Coding Classes



- MDS (Minimum Distance Seperable) codes are **storage optimal**, in order to repair r disk failures, we need r parity disks
- LRC (Locally Repairable Codes) **reduce** the number of **accessed nodes** during a repair, at the cost of optimal storage (MDS) property
- RGC (Regeneration codes) reduce amount of data transferred during a repair, by using more “helpers”, devices contacted during repair
- MSR (Minimum Storage Regeneration) codes minimize repair traffic without storage overhead.
- **Systematic MSR codes** also provide optimal disk I/O (Could you think why?)

Table of Contents

Erasure Codes Tradeoffs

- Erasure codes **decrease storage overhead** compared to replication, at a **higher repair cost** including: excessive computation, disk I/O, network bandwidth
- Increased repair costs **reduce** MTTDL and therefore **data durability**
- LRC offers an optimal trade-off between storage overhead, fault tolerance, number of nodes involved in repairs, but the codes class is not storage optimal
- LRC is widely used today, for example in Windows Azure, but is not storage optimal and doesn't achieve minimum repair traffic

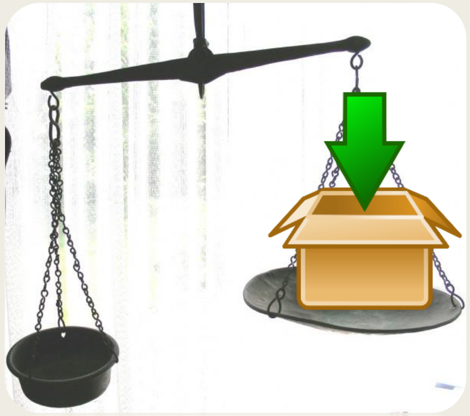
$\frac{k}{n} \leq \frac{1}{\prod_{j=1}^{t}\left(1+\frac{1}{jr}\right)}$

In LRC (n=?,k=10,r=5,t=2) with 2 erasures: $n \geq 10 \cdot 1.2 \cdot 1.1 = 14$

MSR Tradeoffs

- Most known MSR codes require storage overhead at least $\times 2$
  - *otherwise requiring an exponentially growing field or exponential number of code sub-elements, tradeoff with optimal rebuild*
- Fine grain read accesses allow locality in cache but cause disk inefficiencies due to small sub-elements (large k vs. small k)
- Computationally cheap (xor operations), but update complexity is high
- Large object takes a lot of computation time and DRAM, against a small object which consumes a lot of communication, tradeoff so they overlap

MSR System Design Tradeoffs



- Tradeoffs coming from the following choices will be analyzed in 2 butterfly implementations, Hadoop and Ceph:
  - *Online encoding or batch job encoding*
  - *Per-object encoding or object groups encoding*
  - *Open-interface or monolithic interface*

Table of Contents

Butterfly Construction

Code element

Code column

$$\overbrace{\begin{array}{cccc} D_4^3 & D_4^2 & D_4^1 & D_4^0 \\ \begin{pmatrix} \boxed{d_0} \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{pmatrix} & \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{pmatrix} & \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} & \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \end{array}}^{\text{Data chunks contains}} \qquad \overbrace{\begin{array}{cc} H & B \\ \begin{pmatrix} \oplus \\ \vdots \\ \\ \\ \\ \\ \vdots \\ \oplus \end{pmatrix} & \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \end{array}}^{\text{Redundancy chunks contains}}$$

- k α-dimensional data vectors
- $r(= 2)$: number of parity vectors/chunks
  - $n = k + r$
  - ***A chunk consists of*** *a $\alpha(= 8)$-dimensional data vector and is stored in a separate node*
- MDS code with $r = 2$, MSR (rebuilding ratio $\frac{1}{2}$)
- Over a small field $GF(2)$, thus requiring only XOR and AND operations

Butterfly Encoder

- Denote D_k a $2^{k-1} \times k$ boolean matrix for $k \geq 2$, which represents a data object to be encoded.

  $D_k = \begin{bmatrix} \boldsymbol{a} & A \\ \boldsymbol{b} & B \end{bmatrix}$ – *$\boldsymbol{a}$ and $\boldsymbol{b}$ are column vectors of length 2^{k-2}*

- Let D_k^j be the jth column of D_k, $j \in \{0,1,\ldots,k-1\}$
- For parities $H = \mathcal{H}(D_k)$, $B = \mathcal{B}(H_k)$ define:
  - $k = 2$: $\mathcal{H}\left(\begin{bmatrix} c & a \\ d & b \end{bmatrix}\right) = \begin{bmatrix} c \oplus a \\ d \oplus b \end{bmatrix}$ $\mathcal{B}\left(\begin{bmatrix} c & a \\ d & b \end{bmatrix}\right) = \begin{bmatrix} d \oplus a \\ c \oplus a \oplus b \end{bmatrix}$
  - $k > 2$: $\mathcal{H}(D_k) = \begin{bmatrix} \boldsymbol{a} \oplus \mathcal{H}(A) \\ P_{k-1}[P_{k-1}\boldsymbol{b} \oplus \mathcal{H}(P_{k-1}B)] \end{bmatrix}$

    $\mathcal{B}(D_k) = \begin{bmatrix} P_{k-1}\boldsymbol{b} \oplus \mathcal{B}(A) \\ P_{k-1}[\boldsymbol{a} \oplus \mathcal{H}(A) \oplus \mathcal{B}(P_{k-1}B)] \end{bmatrix}$

Butterfly Encoder

- Define:
  - $k = 2$: $\mathcal{H}\left(\begin{bmatrix} c & a \\ d & b \end{bmatrix}\right) = \begin{bmatrix} c \oplus a \\ d \oplus b \end{bmatrix}$ $\quad \mathcal{B}\left(\begin{bmatrix} c & a \\ d & b \end{bmatrix}\right) = \begin{bmatrix} d \oplus a \\ c \oplus a \oplus b \end{bmatrix}$
  - $k > 2$: $\mathcal{H}(D_k) = \begin{bmatrix} \boldsymbol{a} \oplus \mathcal{H}(A) \\ P_{k-1}[P_{k-1}\boldsymbol{b} \oplus \mathcal{H}(P_{k-1}B)] \end{bmatrix}$

The **result** will be a regular **xor parity.**

$$\mathcal{B}(D_k) = \begin{bmatrix} P_{k-1}\boldsymbol{b} \oplus \mathcal{B}(A) \\ P_{k-1}[\boldsymbol{a} \oplus \mathcal{H}(A) \oplus \mathcal{B}(P_{k-1}B)] \end{bmatrix}$$

Double vertical flip is used to simultaneously compute $\mathcal{H}$ and $\mathcal{B}$ over the same data.

Encoding D_k by encoding A and $P_{k-1}B$

- $D_k = \begin{bmatrix} \boldsymbol{a} & A \\ \boldsymbol{b} & B \end{bmatrix}$
- P_k is a $2^{k-1} \times 2^{k-1}$ vertical flip transform: $\begin{bmatrix} & & 1 \\ & ⋰ & \\ 1 & & \end{bmatrix}$

$$P_k \cdot \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \longrightarrow \begin{pmatrix} d \\ c \\ b \\ a \end{pmatrix}$$

Encoding Example

$$\mathcal{H}(D_k) = \begin{bmatrix} \boldsymbol{a} \oplus \mathcal{H}(A) \\ P_{k-1}[P_{k-1}\boldsymbol{b} \oplus \mathcal{H}(P_{k-1}B)] \end{bmatrix}$$

$$\mathcal{B}(D_k) = \begin{bmatrix} P_{k-1}\boldsymbol{b} \oplus \mathcal{B}(A) \\ P_{k-1}[\boldsymbol{a} \oplus \mathcal{H}(A) \oplus \mathcal{B}(P_{k-1}B)] \end{bmatrix}$$

D_4^3	D_4^2	D_4^1	D_4^0	H			B	
d_0	c_0	b_0	a_0	$d_0 \oplus$	$c_0 \oplus b_0 \oplus a_0$	$d_7 \oplus$	$c_3 \oplus$	$b_1 \oplus a_0$
d_1	c_1	b_1	a_1	$d_1 \oplus$	$c_1 \oplus b_1 \oplus a_1$	$d_6 \oplus$	$c_2 \oplus$	$b_0 \oplus a_0 \oplus a_1$
d_2	c_2	b_2	a_2	$d_2 \oplus$	$c_2 \oplus b_2 \oplus a_2$	$d_5 \oplus$	$c_1 \oplus b_1 \oplus a_1 \oplus$	$b_3 \oplus a_3 \oplus a_2$
d_3	c_3	b_3	a_3	$d_3 \oplus$	$c_3 \oplus b_3 \oplus a_3$	$d_4 \oplus$	$c_0 \oplus b_0 \oplus a_0 \oplus$	$b_2 \oplus a_3$
d_4	c_4	b_4	a_4	$d_4 \oplus$	$c_4 \oplus b_4 \oplus a_4$	$d_3 \oplus c_3 \oplus b_3 \oplus a_3 \oplus$		
d_5	c_5	b_5	a_5	$d_5 \oplus$	$c_5 \oplus b_5 \oplus a_5$	$d_2 \oplus c_2 \oplus b_2 \oplus a_2 \oplus$	$\mathcal{B}(P_{k-1}B)$	
d_6	c_6	b_6	a_6	$d_6 \oplus$	$c_6 \oplus b_6 \oplus a_6$	$d_1 \oplus c_1 \oplus b_1 \oplus a_1 \oplus$		
d_7	c_7	b_7	a_7	$d_7 \oplus$	$c_7 \oplus b_7 \oplus a_7$	$d_0 \oplus c_0 \oplus b_0 \oplus a_0 \oplus$		

$$D_k = \begin{bmatrix} \boldsymbol{a} & A \\ \boldsymbol{b} & B \end{bmatrix} \qquad \mathcal{H}\left(\begin{bmatrix} c & a \\ d & b \end{bmatrix}\right) = \begin{bmatrix} c \oplus a \\ d \oplus b \end{bmatrix} \qquad \mathcal{B}\left(\begin{bmatrix} c & a \\ d & b \end{bmatrix}\right) = \begin{bmatrix} d \oplus a \\ c \oplus a \oplus b \end{bmatrix}$$

Butterfly Decoder

$$\mathcal{H}\left(\begin{bmatrix} c & a \\ d & b \end{bmatrix}\right) = \begin{bmatrix} c \oplus a \\ d \oplus b \end{bmatrix} \quad \mathcal{B}\left(\begin{bmatrix} c & a \\ d & b \end{bmatrix}\right) = \begin{bmatrix} d \oplus a \\ c \oplus a \oplus b \end{bmatrix}$$

- *Theorem 1: (MDS) The Butterfly code can decode the original data matrix when* any *two columns are missing, hence it is an MDS code.*
- Proof by induction on k:
  - *For $k = 2$, can be easily verified*
  - *i.e. let $H = \begin{pmatrix} h_1 \\ h_2 \end{pmatrix}$, $B = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$ and data $\begin{pmatrix} c & a \\ d & b \end{pmatrix}$*
  - *Then, if both data nodes fail $b = h_1 \oplus b_2,\ a = h_1 \oplus h_2 \oplus b_1 \oplus b_2$ and etc.*

Butterfly Decoder

- *Theorem 1: (MDS) The Butterfly code can decode the original data matrix when **any** two columns are missing, hence it is an MDS code.*
- Proof by induction on k:
  - *Assuming it gives an MDS code for $k-1$, for $k>2$, we will prove that the construction with k columns is also MDS.*

1. The two parity nodes are lost, *re-encode*
2. One parity and one data column, decode data column, then re-encode. If H failed, if D_k^{k-1}, XORing. Otherwise, get $\mathcal{B}(P_{k-1}B)$ by induction (2 failures), then get $\mathcal{B}(A)$, $\mathcal{H}(A)$ and by induction (1 failure)
3. Two non leftmost data columns are lost:
   - $\boldsymbol{a} \oplus h_1 = \mathcal{H}(A)$; $P_{k-1}\boldsymbol{b} \oplus b_1 = \mathcal{B}(A)$ - *inductively recover upper half*
   - *Similarly, generate by XORing $\mathcal{H}(P_{k-1}B)$ and $\mathcal{B}(P_{k-1}B)$*

$$\mathcal{H}(D_k) = \begin{bmatrix} \boldsymbol{a} \oplus \mathcal{H}(A) \\ P_{k-1}[P_{k-1}\boldsymbol{b} \oplus \mathcal{H}(P_{k-1}B)] \end{bmatrix} \qquad \mathcal{B}(D_k) = \begin{bmatrix} P_{k-1}\boldsymbol{b} \oplus \mathcal{B}(A) \\ P_{k-1}[\boldsymbol{a} \oplus \mathcal{H}(A) \oplus \mathcal{B}(P_{k-1}B)] \end{bmatrix}$$

Butterfly Decoder

- *Theorem 1: (MDS) The Butterfly code can decode the original data matrix when any two columns are missing, hence it is an MDS code.*
- Proof by induction on k:
  - *Assuming it gives an MDS code for $k-1$, for $k>2$, we will prove that the construction with k columns is also MDS.*

4. The leftmost column along with another data column D_k^j are lost.
   - *Notice that $\mathcal{B}(P_{k-1}B) = h_1 \oplus P_{k-1}b_2$. We can easily get $\mathcal{H}(P_{k-1}B)$ from H and decode the bottom half of D_k^j.*
   - *From h_2 we can decode $\boldsymbol{b}$.*
   - *From b_1 we get $\mathcal{B}(A)$ and in the same manner get upper half of D_k^j and $\boldsymbol{a}$.*

$$\mathcal{H}(D_k) = \begin{bmatrix} \boldsymbol{a} \oplus \mathcal{H}(A) \\ P_{k-1}[P_{k-1}\boldsymbol{b} \oplus \mathcal{H}(P_{k-1}B)] \end{bmatrix} \qquad \mathcal{B}(D_k) = \begin{bmatrix} P_{k-1}\boldsymbol{b} \oplus \mathcal{B}(A) \\ P_{k-1}[\boldsymbol{a} \oplus \mathcal{H}(A) \oplus \mathcal{B}(P_{k-1}B)] \end{bmatrix}$$

Single Column Regeneration

- *Theorem 2: (optimal regeneration) In the case of one failure, the lost column can be regenerated by communicating an amount of data equal to $^1/_2$ of the remaining data.*
  - *If the lost column is **not the butterfly parity,** the amount of communicated data is equal to the amount read from surviving disks (optimal I/O).*
- At first I'll present the algebraic expressions for choosing the elements to send from each disk, then give an intuition

1. One column $\mathrm{D_k^j} \in \{D_k^1, \dots, D_k^{k-1}\}$ is lost. Every remaining column will transfer elements in position i such that $\left\lfloor \frac{i}{2^{j-1}} \right\rfloor \equiv_4 0 \vee 3$

Single Column Regeneration

Notice that the indices we read correspond to butterfly locations that do not require the additional elements

- *Theorem 2: (optimal regeneration)*

1. One column $\mathrm{D}_{\mathrm{k}}^{\mathrm{j}} \in \{D_k^1, \dots, D_k^{k-1}\}$ is lost. Every remaining col transfer elements in position i such that $\left\lfloor \frac{i}{2^{j-1}} \right\rfloor \equiv_4 0 \vee 3$

$h = \mathcal{H}(D_{k-1}) \oplus H_{k-1}$
$b = \mathcal{B}(D_{k-1}) \oplus B_{k-1}$
Data lost from D_k^j contained in h, b.

D_4^3	D_4^2	D_4^1	D_4^0		H		B	
d_0	c_0	b_0	a_0	$d_0 \oplus$	$c_0 \oplus b_0 \oplus a_0$	$d_7 \oplus$	$c_3 \oplus$	$b_1 \oplus a_0$
d_1	c_1	b_1	a_1	$d_1 \oplus$	$c_1 \oplus b_1 \oplus a_1$	$d_6 \oplus$	$c_2 \oplus$	$b_0 \oplus a_0 \oplus a_1$
d_2	c_2	b_2	a_2	$d_2 \oplus$	$c_2 \oplus b_2 \oplus a_2$	$d_5 \oplus$	$c_1 \oplus b_1 \oplus a_1 \oplus$	$b_3 \oplus a_3 \oplus a_2$
d_3	c_3	b_3	a_3	$d_3 \oplus$	$c_3 \oplus b_3 \oplus a_3$	$d_4 \oplus$	$c_0 \oplus b_0 \oplus a_0 \oplus$	$b_2 \oplus a_3$
d_4	c_4	b_4	a_4	$d_4 \oplus$	$c_4 \oplus b_4 \oplus a_4$	$d_3 \oplus c_3 \oplus b_3 \oplus a_3 \oplus$	$c_7 \oplus b_7 \oplus a_7 \oplus$	$b_5 \oplus a_4$
d_5	c_5	b_5	a_5	$d_5 \oplus$	$c_5 \oplus b_5 \oplus a_5$	$d_2 \oplus c_2 \oplus b_2 \oplus a_2 \oplus$	$c_6 \oplus b_6 \oplus a_6 \oplus$	$b_4 \oplus a_4 \oplus a_5$
d_6	c_6	b_6	a_6	$d_6 \oplus$	$c_6 \oplus b_6 \oplus a_6$	$d_1 \oplus c_1 \oplus b_1 \oplus a_1 \oplus$	$c_5 \oplus$	$b_7 \oplus a_7 \oplus a_6$
d_7	c_7	b_7	a_7	$d_7 \oplus$	$c_7 \oplus b_7 \oplus a_7$	$d_0 \oplus c_0 \oplus b_0 \oplus a_0 \oplus$	$c_4 \oplus$	$b_6 \oplus a_7$

Single Column Regeneration

$$\mathcal{B}(D_{k-1}) = \begin{pmatrix} d_6 \oplus c_2 \oplus b_0 \\ d_4 \oplus c_0 \oplus b_0 \oplus b_2 \\ d_2 \oplus c_2 \oplus b_2 \oplus c_6 \oplus b_6 \oplus b_4 \\ d_0 \oplus c_0 \oplus b_0 \oplus c_4 \oplus b_6 \end{pmatrix}$$

- *Theorem 2: (optimal regeneration)*

2. Column $\mathrm{D}_{\mathrm{k}}^{0}$ is lost. The columns $D_k^1, \ldots, D_k^{k-1}, H$ will transfer even indexed elements and B will transfer odd indexed elements.

D_4^3	D_4^2	D_4^1	D_4^0	H		B		
d_0	c_0	b_0	a_0	$d_0 \oplus$	$c_0 \oplus b_0 \oplus a_0$	$d_7 \oplus$	$c_3 \oplus$	$b_1 \oplus a_0$
d_1	c_1	b_1	a_1	$d_1 \oplus$	$c_1 \oplus b_1 \oplus a_1$	$d_6 \oplus$	$c_2 \oplus$	$b_0 \oplus a_0 \oplus a_1$
d_2	c_2	b_2	a_2	$d_2 \oplus$	$c_2 \oplus b_2 \oplus a_2$	$d_5 \oplus$	$c_1 \oplus b_1 \oplus a_1 \oplus$	$b_3 \oplus a_3 \oplus a_2$
d_3	c_3	b_3	a_3	$d_3 \oplus$	$c_3 \oplus b_3 \oplus a_3$	$d_4 \oplus$	$c_0 \oplus b_0 \oplus a_0 \oplus$	$b_2 \oplus a_3$
d_4	c_4	b_4	a_4	$d_4 \oplus$	$c_4 \oplus b_4 \oplus a_4$	$d_3 \oplus c_3 \oplus b_3 \oplus a_3 \oplus$	$c_7 \oplus b_7 \oplus a_7 \oplus$	$b_5 \oplus a_4$
d_5	c_5	b_5	a_5	$d_5 \oplus$	$c_5 \oplus b_5 \oplus a_5$	$d_2 \oplus c_2 \oplus b_2 \oplus a_2 \oplus$	$c_6 \oplus b_6 \oplus a_6 \oplus$	$b_4 \oplus a_4 \oplus a_5$
d_6	c_6	b_6	a_6	$d_6 \oplus$	$c_6 \oplus b_6 \oplus a_6$	$d_1 \oplus c_1 \oplus b_1 \oplus a_1 \oplus$	$c_5 \oplus$	$b_7 \oplus a_7 \oplus a_6$
d_7	c_7	b_7	a_7	$d_7 \oplus$	$c_7 \oplus b_7 \oplus a_7$	$d_0 \oplus c_0 \oplus b_0 \oplus a_0 \oplus$	$c_4 \oplus$	$b_6 \oplus a_7$

Single Column Regeneration

- *Theorem 2: (optimal regeneration)*

3. First parity column H is lost. All the remaining columns transfer their lower halves. Butterfly parity XORed with data from D_{k-1} will provide upper rows of H.

D_4^3	D_4^2	D_4^1	D_4^0	H			B	
d_0	c_0	b_0	a_0	$d_0 \oplus$	$c_0 \oplus b_0 \oplus a_0$	$d_7 \oplus$	$c_3 \oplus$	$b_1 \oplus a_0$
d_1	c_1	b_1	a_1	$d_1 \oplus$	$c_1 \oplus b_1 \oplus a_1$	$d_6 \oplus$	$c_2 \oplus$	$b_0 \oplus a_0 \oplus a_1$
d_2	c_2	b_2	a_2	$d_2 \oplus$	$c_2 \oplus b_2 \oplus a_2$	$d_5 \oplus$	$c_1 \oplus b_1 \oplus a_1 \oplus$	$b_3 \oplus a_3 \oplus a_2$
d_3	c_3	b_3	a_3	$d_3 \oplus$	$c_3 \oplus b_3 \oplus a_3$	$d_4 \oplus$	$c_0 \oplus b_0 \oplus a_0 \oplus$	$b_2 \oplus a_3$
d_4	c_4	b_4	a_4	$d_4 \oplus$	$c_4 \oplus b_4 \oplus a_4$	$d_3 \oplus c_3 \oplus b_3 \oplus a_3 \oplus$	$c_7 \oplus b_7 \oplus a_7 \oplus$	$b_5 \oplus a_4$
d_5	c_5	b_5	a_5	$d_5 \oplus$	$c_5 \oplus b_5 \oplus a_5$	$d_2 \oplus c_2 \oplus b_2 \oplus a_2 \oplus$	$c_6 \oplus b_6 \oplus a_6 \oplus$	$b_4 \oplus a_4 \oplus a_5$
d_6	c_6	b_6	a_6	$d_6 \oplus$	$c_6 \oplus b_6 \oplus a_6$	$d_1 \oplus c_1 \oplus b_1 \oplus a_1 \oplus$	$c_5 \oplus$	$b_7 \oplus a_7 \oplus a_6$
d_7	c_7	b_7	a_7	$d_7 \oplus$	$c_7 \oplus b_7 \oplus a_7$	$d_0 \oplus c_0 \oplus b_0 \oplus a_0 \oplus$	$c_4 \oplus$	$b_6 \oplus a_7$

Single Column Regeneration

- *Theorem 2: (optimal regeneration)*

4. Second parity column B is lost. D_k^{k-1} will transfer its top half, H will transfer its bottom half, $D_k^j, j \neq k-1$ will transfer their contribution to bottom part of B.

1. XORing data will give bottom part of B.
2. Butterfly of $D_k^j, j \neq k-1$ and XOR with bottom half of H will recover top part of B.

D_4^3	D_4^2	D_4^1	D_4^0	H		B		
d_0	c_0	b_0	a_0	$d_0 \oplus$	$c_0 \oplus b_0 \oplus a_0$	$d_7 \oplus$	$c_3 \oplus$	$b_1 \oplus a_0$
d_1	c_1	b_1	a_1	$d_1 \oplus$	$c_1 \oplus b_1 \oplus a_1$	$d_6 \oplus$	$c_2 \oplus$	$b_0 \oplus a_0 \oplus a_1$
d_2	c_2	b_2	a_2	$d_2 \oplus$	$c_2 \oplus b_2 \oplus a_2$	$d_5 \oplus$	$c_1 \oplus b_1 \oplus a_1 \oplus$	$b_3 \oplus a_3 \oplus a_2$
d_3	c_3	b_3	a_3	$d_3 \oplus$	$c_3 \oplus b_3 \oplus a_3$	$d_4 \oplus$	$c_0 \oplus b_0 \oplus a_0 \oplus$	$b_2 \oplus a_3$
d_4	c_4	b_4	a_4	$d_4 \oplus$	$c_4 \oplus b_4 \oplus a_4$	$d_3 \oplus c_3 \oplus b_3 \oplus a_3 \oplus$	$c_7 \oplus b_7 \oplus a_7 \oplus$	$b_5 \oplus a_4$
d_5	c_5	b_5	a_5	$d_5 \oplus$	$c_5 \oplus b_5 \oplus a_5$	$d_2 \oplus c_2 \oplus b_2 \oplus a_2 \oplus$	$c_6 \oplus b_6 \oplus a_6 \oplus$	$b_4 \oplus a_4 \oplus a_5$
d_6	c_6	b_6	a_6	$d_6 \oplus$	$c_6 \oplus b_6 \oplus a_6$	$d_1 \oplus c_1 \oplus b_1 \oplus a_1 \oplus$	$c_5 \oplus$	$b_7 \oplus a_7 \oplus a_6$
d_7	c_7	b_7	a_7	$d_7 \oplus$	$c_7 \oplus b_7 \oplus a_7$	$d_0 \oplus c_0 \oplus b_0 \oplus a_0 \oplus$	$c_4 \oplus$	$b_6 \oplus a_7$

Table of Contents

Hadoop Filesystem

- Designed for managing large-scale computation/storage systems, suitable for **large** amounts of data
- One of the most widely used distributed storage systems in industry and academics
- *Namenode* – metadata and location, “centralized” architecture
  - *Limited metadata*
  - *Single point of failure*
- *Datanode* – actual files
- Facebook implemented Reed-Solomon, XORbas codes in a module named HDFS-RAID

Erasure Coding in HDFS

- HDFS-RAID does encoding as a batch-job
  - *Low write latency*
  - *Additional storage (think why)*
- Iterative version implemented to avoid Java recursion and non-explicit memory management
- Facebook HDFS as a starting point
  - *RaidNode encodes (creates parity files)*
  - *BlockFixer fixes corrupted data*
- Files are replicated, RaidNode schedules coding jobs, take k newly inserted chunks and generates r parity chunks, stored back in HDFS

Butterfly Implementation in HDFS

- Encoding and repair follow a 4-step protocol:
    1. *Determine the location of the data blocks:*
       To calculate file offset of the k-block message we use the ***position*** *of the symbol being built and its* ***index***
    2. *Fetch the data to the primary node - asynchronously*
    3. *Encoding/decoding computation is performed in the primary node*
    4. *Created data is committed back to HDFS*

Butterfly Implementation in HDFS

- Tuning code column size improves data locality for cache, and communication can be overlapped by computation.
- JNI can increase optimizations, but benefits shadowed by cost of data movements between Java and JNI modules.
- Loop unrolling and reordering didn’t increase performance as expected
- Memory management outweighs benefits of computation optimizations
- Parallelize in OpenMP fashion, avoid column collocating

Communication Protocol and Memory in HDFS

- If the data stream is broken, client assumes communication error and starts re-establishing connection with datanode
- To solve the problem, the Datanode packs the data contagiously into a buffer and sends to the client which extracts it
- In Reed-Solomon, $(k+2) \times 64MB$ DRAM required for decoding, buffered communication requires additional space
- Multiple sequential decode tasks require garbage-collecting, if frequent and not properly scheduled, they cause performance degradation
  - *They implement memory pool, performance benefits of up to 15%*
  - *Allocated during task setup, reused by computation threads*

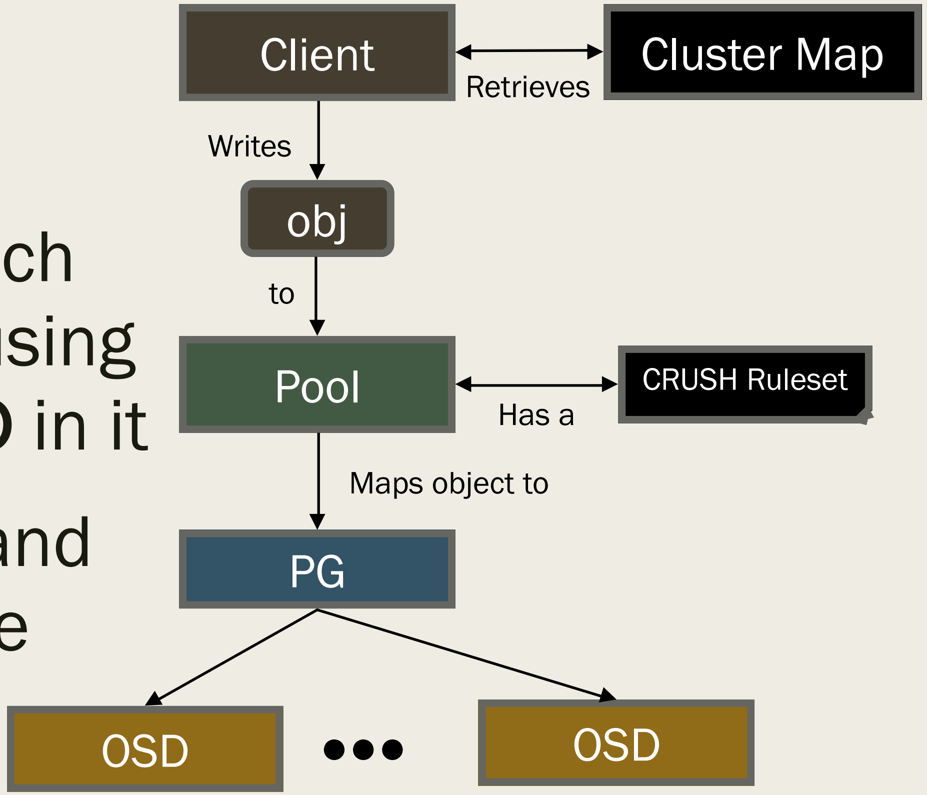
Table of Contents

Ceph’s Distributed Object Store

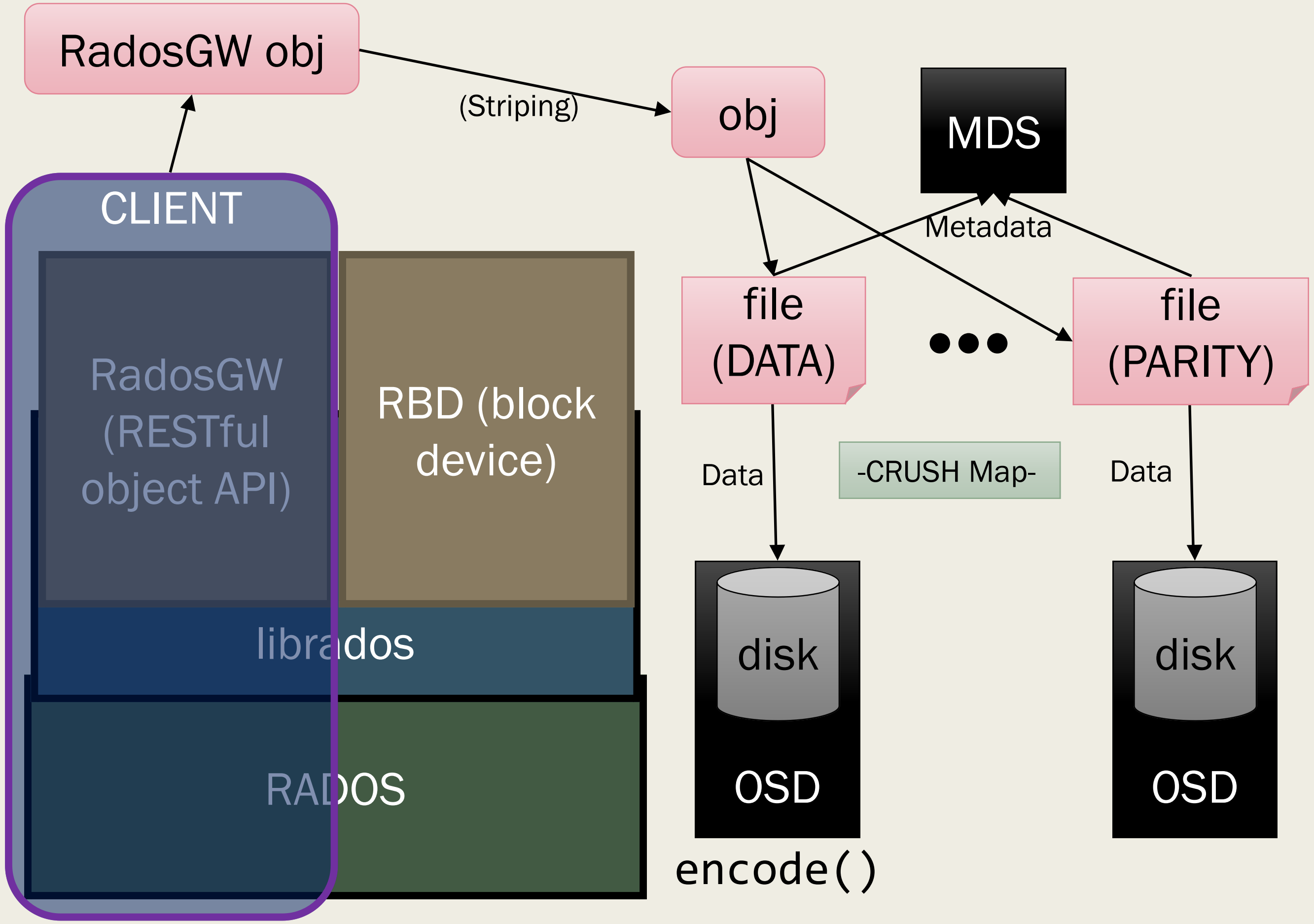
- Ceph is an open-source distributed storage system with a decentralized design, no single point of failure
- Self-healing and self-managing, guarantee high-availability and consistency with little human intervention
- RADOS is its core component, formed of **daemons** and client libs, allowing partial and complete R/W and snapshots
- Monitor - maintain consistent cluster metadata
- OSD (node) - object storage device

Ceph Notation

- Pool - a Ceph's logical partition for storing objects
- Placement Group (PG) - A sub-pool, to which an object can be written, pool decides it using CRUSH ruleset. OSDs elect a **primary OSD** in it
- CRUSH – an algorithm, gets an object id and returns a vector of nodes/PG ids. Uses the CRUSH ruleset which is a storage tree



Ceph Pipeline Diagram

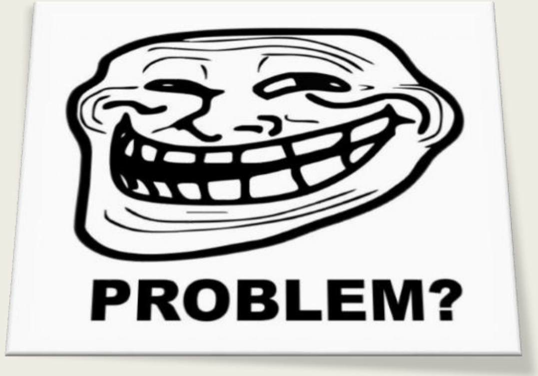


- The client sends a RadosGW object using PUT/GET
- The RadosGW object is being striped and converted into small ***Ceph objects*** ***(~4MB each)***
- The objects are partitioned to ***data chunks*** and erasure coded to ***parity chunks*** ***online***
- The chunks (which are represented as files) are sent to the corresponding ***OSDs*** using the ***CRUSH*** map
- The ***MDS*** gets the files' metadata

Butterfly Codes in Ceph

- Larger stripe size requires more memory, increases write latency
- On the other hand, coarser computation and read operations benefit performance of erasure codes
  - *Small elements incur **high network** and **HDD overhead***
  - 2^{k-1} *elements per code column require larger stripes*
- Erasure code plugin infrastructure, separated from OSD
- Although, designed for traditional and LRC codes $=$ No support in array codes!

Butterfly Implementation in Ceph



- The given infrastructure includes:
  - `encode()`: *returns a list of n encoded chunks*
  - `minimum_to_decode()`: *gets chunk ID and available chunks list, returns IDs of required chunks*
  - `decode()`: *given a list of chunk IDs, decodes*
- All in resolution of ***chunks***, can't access elements!
- They implemented as an external DLL the `repair()` function which works in resolution of elements

Butterfly Implementation in Ceph

- Butterfly is implemented as an external C lib and compiled as a new RADOS plug-in
- High level of algorithmic and implementation optimizations, due to programming language (C++)
- Uses the recursive approach which ***simplifies*** implementation, it also gives better data locality
  - *Better encoding throughput*

Table of Contents

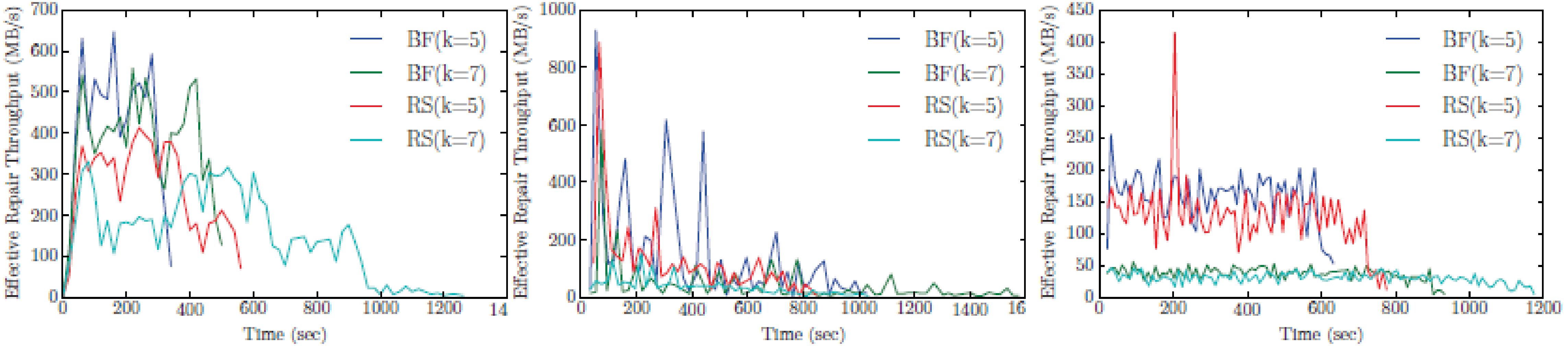
Experimental Setup

- 12 Dell R720 servers, each with one OS HDD and seven 4TB HDDs, total 336TB
- Infiniband network, 56Gbps – communication faster than HDDs
- 12 storage nodes, 1 metadata server
- 2 code constructions, $k = 5, k = 7$, storage overhead of $1.4\times$, $1.3\times$ respectively, 16 and 64 elements in a column accordingly
  - *Catches impact of IO granularity on repair performance*
- Compare Butterfly against Reed-Solomon with the same k

Evaluation method

1. Store 20K objects of 64MB, total 1.8TB
   - *each node stores 150GB for k=5, 137GB for k=7 (redundancy overhead)*
2. Power off a single storage server and let the 11 remaining servers repair the lost data
   - *log CPU, IO, network BW.*

Repair Throughput



(a) HDFS. (b) Ceph with 4MB stripes. (c) Ceph with 64MB stripes.

Figure 5: Repair throughput aggregated across all nodes involved in the repair process. Each system configuration we run with RS and Butterfly, with $k = 5$ and $k = 7$.

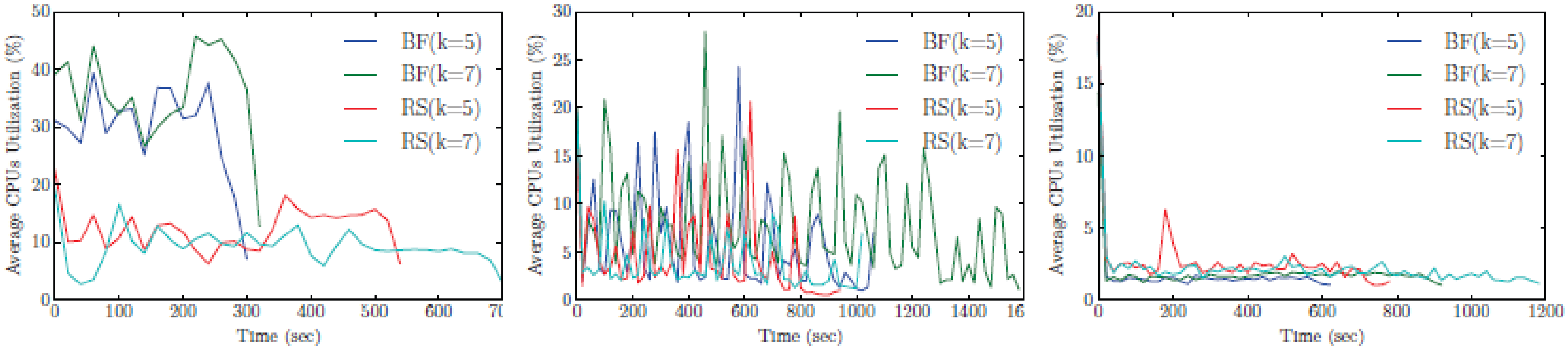
Repair Throughput in HDFS

- For HDFS, 12 reduce tasks per node, 1 per core (would be better to put 11 tasks so that background jobs don't interfere)
- Steep fall in throughput towards the end, because of high parallelization
- RS results caused by load imbalance, out of scope in the paper
- In HDFS, k=5, ~500MB/s, 1.6x higher than RS. Not twice since
  - *Higher contention in butterfly, causing HDD randomness*
  - *Vector-based communication copy-to-buffer overhead*
- k=7, difference is ~2x between BF and RS.
  - *Reducing network contention outweighs HDD drawbacks*

Repair Throughput in Ceph

- 4MB stripe size creates elements of 50KB, 9KB for k=5,7
- Small elements cause inefficient HDD utilization, additional CPU operations. Leads to inconsistent repair throughput
- 64MB stripe size results in elements of sizes 800KB, 143KB for k=5,7 respectively
- Better disk utilization and repair throughput

CPU Utilization



(a) HDFS. (b) Ceph with 4MB stripes. (c) Ceph with 64MB stripes.

Figure 6: Average CPU utilization per server. Each system configuration we run with RS and Butterfly, with $k = 5$ and $k = 7$. The graphs represent the average utilization across all 12 nodes involved in the repair process.

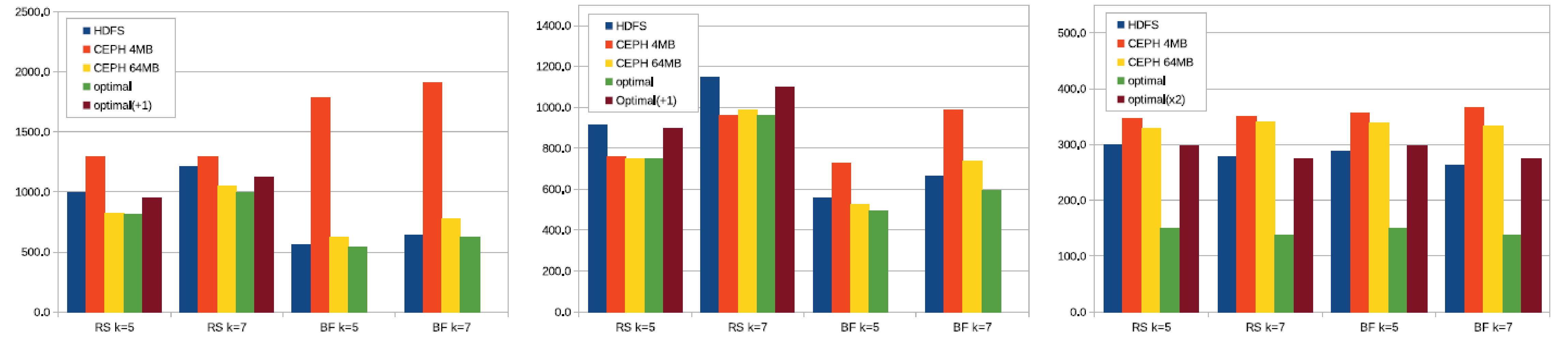
CPU Utilization in Hadoop

- Measuring capability of BF/RS to possibly share in-node resources with other applications
- Results are averaged across all nodes involved in computation
- For HDFS, BF utilization is higher by 3-4x for both k values
  - *Since RS waits more for netwrk IO partially*
  - *BF spends x2.1, x1.7 more CPU cycles for k=5,7 resp.*
  - *Strongly caused by Java, because of small granularity, but no slice access to buffer in Java*

CPU Utilization in Ceph

- For Ceph stripe size of 4MB, k=5, elements are ~50KB
  - *Fine granularity computation and communication causes unpredictable CPU utilization*
  - *Same applies for k=7*
- CPU utilization for RS is lower compared to Butterfly but unstable
  - *Memory management, function calls, cache misses cause that*
- For 64MB stripe, lower and predictable CPU utilization
  - *Cache-aware implementation achieves utilization comparable to RS*
  - *~2-3% utilization for both codes, both k configurations*

Network Traffic

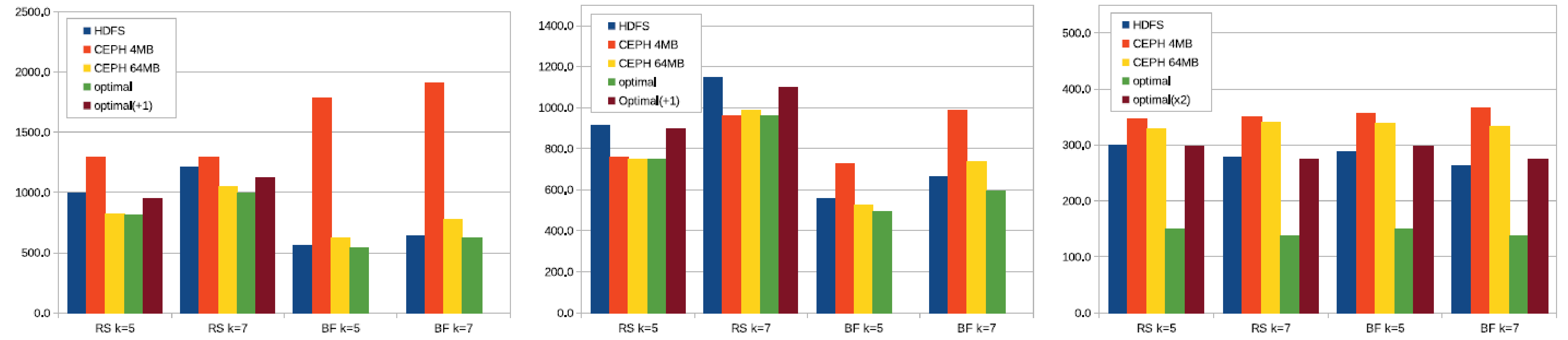


(a) Overall Network Traffic in GB (b) Overall Disk Reads in GB (c) Overall Disk Writes in GB

Figure 7: The aggregate amount (across all 11 nodes included in the repair process) of network traffic and IOs during the repair process. We observe RS and Butterfly with $k = 5$ and $k = 7$.

- Optimal bars represent lower bound on traffic
- Optimal+1 represents minimum + single HDFS block, from HDFS-RAID implementation
- Optimal+1 matches RS HDFS traffic
- In HDFS Butterfly close to theoretical minimum, different because of metadata

Network Traffic

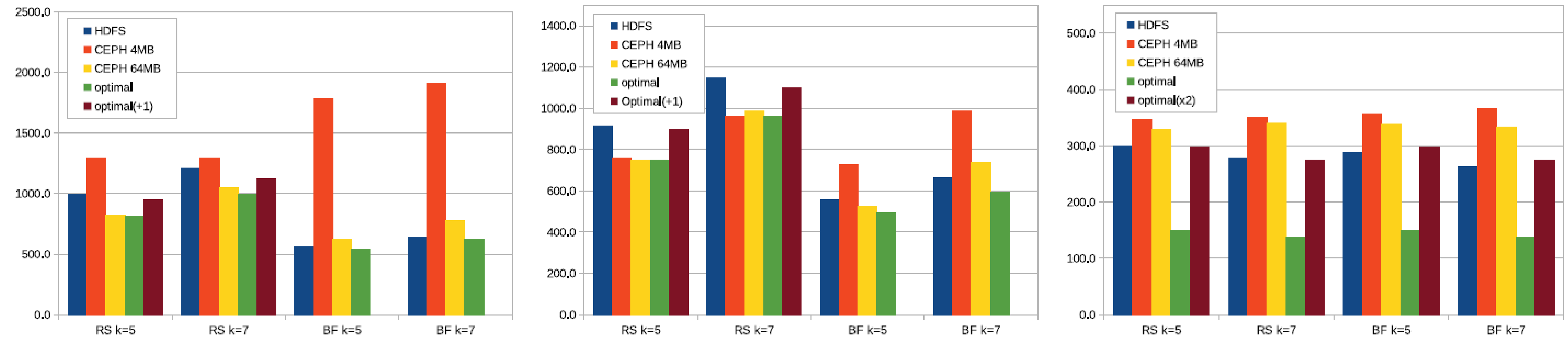


(a) Overall Network Traffic in GB (b) Overall Disk Reads in GB (c) Overall Disk Writes in GB

Figure 7: The aggregate amount (across all 11 nodes included in the repair process) of network traffic and IOs during the repair process. We observe RS and Butterfly with $k = 5$ and $k = 7$.

- In Ceph, overhead is higher, for 4MB blocks
  - Small chunks being transferred between nodes and per-message overhead.
- For 64MB blocks, overhead increases with k because of reduced message size
- *on-line* approach reduces size of encoded messages

Storage Traffic - Reads



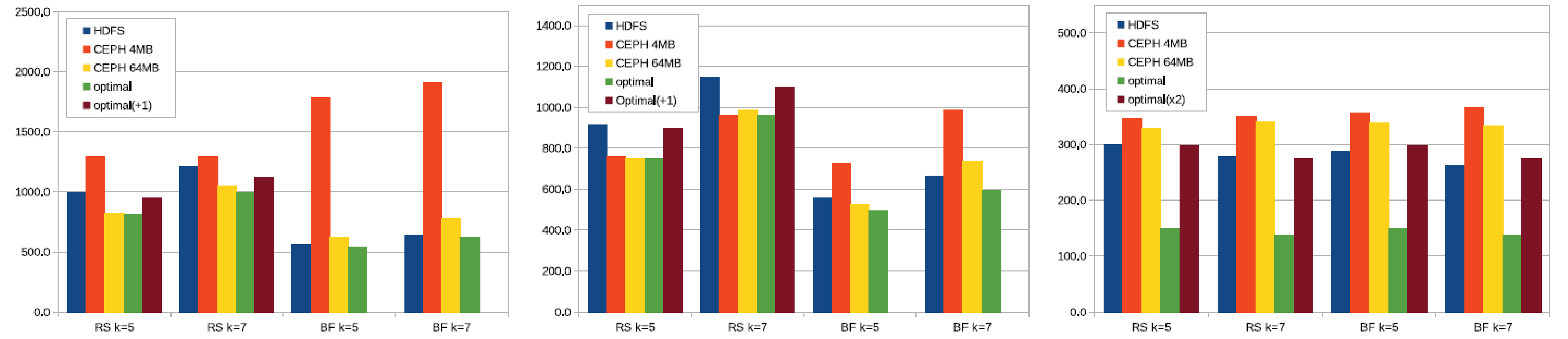
(a) Overall Network Traffic in GB (b) Overall Disk Reads in GB (c) Overall Disk Writes in GB

Figure 7: The aggregate amount (across all 11 nodes included in the repair process) of network traffic and IOs during the repair process. We observe RS and Butterfly with $k = 5$ and $k = 7$.

- Traffic is recorded from all HDDs
- HDFS-Butterfly achieves nearly optimal read traffic, with metadata
- HDFS-RS close to optimal+1

- Ceph-Butterfly disk I/O is smaller for k=5
  - Small I/O sizes cause misaligned reads
  - Read-ahead interferes
- Large stripes cause read overhead to nullify

Storage Traffic - Writes



(a) Overall Network Traffic in GB

(b) Overall Disk Reads in GB

(c) Overall Disk Writes in GB

Figure 7: The aggregate amount (across all 11 nodes included in the repair process) of network traffic and IOs during the repair process. We observe RS and Butterfly with $k = 5$ and $k = 7$.

- For both systems and code configurations, writes amount exceeds *optimal* by ~2x
- Ceph allows updates of stored data, relies on journaling
  - Journal co-located with data: write traffic is doubled
- Ceph load-balancing on the same server affect only disk I/O
- In HDFS, intermediate local file causes 2x write

Table of Contents

Summary

- Since we didn't mention in evaluation, degraded reads will be the same as in Reed-Solomon
- According to CPU utilization measurements, MSR codes over GF(2) achieve low CPU usage and are a good candidate for multi-user environment
  - *Programming language is important*
  - *Relatively coarse data chunks are necessary*
- Carefully implemented, MSR codes can reduce repair network traffic by 2x compared to traditional erasure codes
  - *System design that avoids fine-grain communication is necessary.*
- In practical usage, Microsoft Azure and Facebook Xorbas use LRC code family, which require additional storage overhead

Summary

- Provided answers to the following questions:
  1. *Can the theoretical reduction in repair traffic translate to actual performance improvement*
  2. *In what way system design affects MSR code repair performance*
- Show that MSR reduce network traffic and I/O during repairs in a practical system
- However, encoding/decoding performance depends on system design, memory allocation, etc.
- HDFS experiences CPU overhead because of Java memory mgmt.
- On-line encoding causes high access latency due to fragmentation
- Batch encoding achieves better performance but reduces storage efficiency (intermediate buffer)

References

1. L. Pamies-Juarez, F. Blagojević, R. Mateescu, C. Gyuot, E. En Gad, Z. Bandic, Opening the Chrysalis: On the Real Repair Performance of MSR Codes, FAST 2016
2. *XORing Elephants: Novel Erasure Codes for Big Data M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, VLDB 2013*

The End

Butterfly Construction

- Butterfly code is over $GF(2)$, requires only XOR, AND operations
- This paper claims to be the first to show an achievable performance of MSR codes
- Let:
  - k – *number of* ***systematic*** *chunks*
  - r – *number of parity chunks,* $n = k + r$
  - *Each chunk consists of a* α*-dimensional data vector and is stored in a separate node*