PARIX: Speculative Partial Writes in Erasure-Coded Systems

Huiba Li*, Yiming Zhangθ, Zhiming Zhang*, Shengyun Liuθ, Dongsheng Liθ, Xiaohui Liuθ, Yuxing Pengθ

*Meituan Open Service, θNUDT
Erasure Coding (EC)

4*3=12
(300% redundancy)

4+2=6
(150% redundancy)

data blocks

parity blocks
Usage of EC

- Widely used in distributed storage systems
  - Especially large-scale cloud storage services
- Except in read-write hot storage
- Because of performance

Overheads include:
- Coding calculation
- I/O pattern deterioration, especially for partial writes
Coding with Vectorial Instructions

![Diagram showing Single Core Performance (GB/s) for different coding schemes: EC(4,2), EC(6,3), EC(8,4), EC(10,4). The performance is measured in encode, decode 1, and decode 2 categories.]
Coding with Vectorial Instructions

![Graph showing single core performance for different coding schemes]

- EC(4,2)
- EC(6,3)
- EC(8,4)
- EC(10,4)

Single Core Performance (GB/s)

- Encode
- Decode 1
- Decode 2

Coding Schemes

Coding is no longer the bottleneck
Partial Write in EC

write

data blocks

parity blocks
Partial Write in EC

write

data blocks

parity blocks

unaligned write
A Simple Approach to Partial Write

- By fully re-encoding
A Simple Approach to Partial Write

- By fully re-encoding

high I/O amplification
unfriendly to parallelism
An Incremental Approach to Partial Write

• By incremental encoding
An Incremental Approach to Partial Write

- By incremental encoding

1. Partial Write
2. Read
3. Write
4. incremental data encoding
5. Read
6. incremental parity encoding
7. Write Parity

- moderate I/O amplification
- friendly to inter-disk parallelism
An Incremental Approach to Partial Write

• By incremental encoding

① Partial Write → ③ incremental data encoding → ④ Read
② Read → ③ Write → ⑥ incremental parity encoding → ⑤ Read
⑤ Write → ⑦ Write Parity

moderate I/O amplification
friendly to inter-disk parallelism
but, in-place read-and-write is expensive!
Cost of in-place read-and-write

- its latency is equivalent to that of random seek
- performance hurts a lot
  - random write: reduced by half
  - sequential write: reduced to that of random write
- the major obstacle for EC
  - to get used in read-write hot storage
Cost of in-place read-and-write

• its latency is equivalent to that of random seek

• performance hurts a lot
  ✤ random write: reduced by half
  ✤ sequential write: reduced to that of random write

• the major obstacle for EC
  ✤ to get used in read-write hot storage
Parity Logging

- An approach to accelerate parity writing
Parity Logging

- An approach to accelerate parity writing

1. Partial Write
2. Read
3. incremental data encoding
4. Write
5. Append Incremental Change Log

(null)
Parity Logging

- **client**
- **data**
- **parity\(_1\)**
- **parity\(_2\)**
- **parity\(_3\)**

- write \(d^{(r)}\)
- read
- update
- overwrite
- append \(\Delta p_j\) to journal

Success flows between nodes.
Parity Logging

There's still one in-place read-and-write. It's the source of overhead.
Pervasiveness of Partial Write

- partial write is caused by unalignment
- unalignment is the norm
  - the upper layers of I/O stack just don’t know the fact of EC and its alignment
- It is the major obstacle
  - for EC to get used in read-write hot storage systems
- Must be addressed!

An analysis of the MSR I/O traces
PARIX: Eliminating the Read

• Consider a series of \( r \) writes to a same data block \( d_i \):
  \[
  d_i^{(0)} \leftarrow d_i^{(1)}, \quad d_i^{(2)}, \ldots, \quad d_i^{(r)}
  \]

• The parities are \( p_j \) (\( j=1,2,...,k \))

• By incremental parity update equation, we have:
  \[
  \Delta p_j^{(1)} = a_{ij} \times \Delta d_i^{(1)}, \quad \Delta p_j^{(2)} = a_{ij} \times \Delta d_i^{(2)}
  \]
  \[
  p_j^{(r)} = p_j^{(0)} + \sum_{x=1}^{r} \Delta p_j^{(x)} = p_j^{(0)} + a_{ij}(d_i^{(1)} - d_i^{(0)}) + d_i^{(2)} - d_i^{(1)} + d_i^{(3)} - d_i^{(2)} + \ldots
  \]
  \[
  = p_j^{(0)} + a_{ij} \times (d_i^{(r)} - d_i^{(0)})
  \]

In Galois Field

\( GF(2^8) \)
PARIX: Eliminating the Read

• Consider a series of $r$ writes to a same data block $d_i$:
  $$d_i^{(0)} \leftarrow d_i^{(1)}, d_i^{(2)}, \cdots, d_i^{(r)}$$

• The parities are $p_j$ ($j=1,2,\ldots,k$)

• By incremental parity update equation, we have:
  $$\Delta p_j^{(1)} = a_{ij} \times \Delta d_i^{(1)}, \Delta p_j^{(2)} = a_{ij} \times \Delta d_i^{(2)}$$

  $$p_j^{(r)} = p_j^{(0)} + \sum_{x=1}^{r} \Delta p_j^{(x)} = p_j^{(0)} + a_{ij}(d_i^{(1)} - d_i^{(0)} + d_i^{(2)} - d_i^{(1)} + d_i^{(3)} - d_i^{(2)} + \ldots)$$

  $$= p_j^{(0)} + a_{ij} \times (d_i^{(r)} - d_i^{(0)})$$

  intermediate values are ALL reduced, no need to read it at all!
PARIX: Logging Data on Parities

• Instead of parities deltas, as in parity logging

• Each parity records $m$ series of change logs
  ❪ respectively for $m$ data blocks
  ❪ stored as a single journal file
  ❪ interleaved with each other
  ❪ every $d^{(0)}$ always comes after corresponding $d^{(1)}$

• Example:
  
  $d_2^{(1)}, d_1^{(1)}, d_2^{(0)}, d_4^{(1)}, d_1^{(0)}, d_1^{(2)}, d_2^{(2)}, \ldots, d_i^{(k)} (i = 1..m, k = 0..r_i), \ldots$
The Speculation

• Whether the parities need $d_i^{(0)}$ or not?

• It is too costly to maintain consensus among nodes about this

• Instead, we speculate about it:
  * Assume $d_i^{(0)}$ is NOT needed (mostly right)
  * Send $d_i^{(0)}$ only when it is actually needed (sometimes only)
PARIX: Eliminating the Read

client → data → parity₁ → parity₂ → parity₃
PARIX: Eliminating the Read

write \( d^{(r)} \) to journal

write \( d^{(r)} \)
PARIX: Eliminating the Read

- **client**
- **data**
- **parity\_1**
- **parity\_2**
- **parity\_3**

- write $d^{(r)}$
- write $d^{(r)}$
- write $d^{(r)}$ to journal

need $d^{(0)}$?
PARIX: Eliminating the Read

client -> data
write d⁽r⁾

data -> parity₁
write d⁽r⁾

parity₁ -> parity₂
write d⁽r⁾ to journal

parity₂ -> parity₃

parity₃

need d⁽0⁾?

write d⁽c⁾ N

write d⁽c⁾ W
PARIX: Eliminating the Read
PARIX: Eliminating the Read

client  data  parity\textsubscript{1}  parity\textsubscript{2}  parity\textsubscript{3}

write \(d(r)\)  write \(d(r)\)  write \(d(r)\) to journal  write \(d(r)\) to journal  write \(d(r)\)

write \(d(r)\)  write \(d(r)\)  write \(d(0)\) to journal  write \(d(0)\) to journal  write \(d(0)\)

need \(d(0)?\)  read \(d(0)\)  here’s \(d(0)\)  here’s \(d(0)\)  here’s \(d(0)\)

write \(d(0)\)  write \(d(0)\)  write \(d(0)\) to journal  write \(d(0)\) to journal  write \(d(0)\)

\(\text{write d}(r)\)  \(\text{write d}(r)\)  \(\text{write d}(r)\) to journal  \(\text{write d}(r)\) to journal  \(\text{write d}(r)\)
**PARIX: Eliminating the Read**

The diagram illustrates the process of write and read operations in the PARIX system. The client initiates a write operation (write \(d^{(r)}\)) to the data. The data then writes \(d^{(r)}\) to each of the parity blocks \(\text{parity}_1\), \(\text{parity}_2\), and \(\text{parity}_3\). If any parity block fails, the system retries the write operation (write \(d^{(r)}\) to journal) to ensure data consistency.

- **Write Operation**:
  - Write \(d^{(r)}\) to data block.
  - Write \(d^{(r)}\) to each parity block.
  - If any parity block fails, retry write to journal.

- **Read Operation**:
  - Read \(d^{(0)}\) from data block.

The system ensures data integrity by journaling the write operations and using parity blocks for redundancy. The fail state is labeled as \((2\text{ IOOP} + 2\text{ RTT})\).
PARIX: Eliminating the Read

(1 IOOP + 1 RTT) is the same as that of replication!

penalty of failure is only a network RTT!
Implementation

• Based on Ursa, a block store for our public & private cloud, which hosts all our businesses:
  • food delivery: ~13M orders/day, ~67M monthly active users, ~200M total users
  • crowd-sourced reviews (about businesses), coupons, hotel reservation, tourism, plane & train tickets, movie tickets, payment, ...

• Architecture
  • Master-Server-Client
  • No single-point-of-failure
Evaluation

• 10 Servers, each with:
  ✤ 12 HDD, 7200 RPM,
    - attached to an LSI 3008 SAS HBA, w/o flash cache
  ✤ 2-way Intel Xeon CPU
  ✤ 128GB RAM

• 10Gb Ethernet
  ✤ connected with a non-blocking switch
### Evaluation

#### Random I/O Latency (ms)

<table>
<thead>
<tr>
<th></th>
<th>write (non-cached)</th>
<th>write (cached)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDD</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>R3</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>PBS-2</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>PBS-1</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>PLog</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>EC</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

*(the lower, the better)*

#### Random IOPS

<table>
<thead>
<tr>
<th></th>
<th>write (non-cached)</th>
<th>write (cached)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDD</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>EC</td>
<td>1</td>
<td>38.8</td>
</tr>
<tr>
<td>PLog</td>
<td>2.5</td>
<td>2.5</td>
</tr>
<tr>
<td>PBS-1</td>
<td>2.6</td>
<td>2.5</td>
</tr>
<tr>
<td>PBS-2</td>
<td></td>
<td>35.2</td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td>38.8</td>
</tr>
</tbody>
</table>

*(the higher, the better)*
Evaluation

Recovery Time (s)

Journal Size (normalized to chunk size)
Summary

• Performance is the major obstacle
  ✤ *for EC to get used in read-write hot storage*
  ✤ *especially in the case of partial write*

• PARiX: a novel approach to eliminate overhead in common cases
  ✤ *with a very small penalty in corner cases*

• Evaluations show that its performance meets expectations
  ✤ *much better than existing approaches*
  ✤ *close to that of 3-replica scheme*
2017 USENIX Annual Technical Conference

https://www.mtyun.com
Thanks!

2017 USENIX Annual Technical Conference