Piggybacked Erasure Codes for Distributed Storage & Findings from the Facebook Warehouse Cluster

K. V. Rashmi, Nihar Shah, D. Gu, H. Kuang, D. Borthakur, K. Ramchandran
Piggybacked Erasure Codes for Distributed Storage &
Findings from the Facebook Warehouse Cluster

K. V. Rashmi, Nihar Shah, D. Gu, H. Kuang, D. Borthakur, K. Ramchandran

Presented by Kangwook Lee
=> Unhandled questions will be happily forwarded
Outline

• Introduction & Motivation
  – Measurements from Facebook’s Warehouse cluster

• The Piggybacking framework

• Via the Piggybacking framework
  – Best known codes for several settings
  – Comparison with other codes
  – Preliminary practical experiments

• Summary & future work
Outline

• Introduction & Motivation
  – Measurements from Facebook’s Warehouse cluster

• The Piggybacking framework

• Via the Piggybacking framework
  – Best known codes for several settings
  – Comparison with other codes
  – Preliminary practical experiments

• Summary & future work
Motivation: Facebook’s Warehouse Cluster Measurements

- Multiple tens of PBs and growing
- Multiple thousands of nodes

Reducing storage requirements is of high importance

- Uses (14, 10) RS code for storage efficiency
  - on less-frequently accessed data
- Multiple PBs of RS coded data

[Rashmi et al., USENIX HotStorage 2013]
“A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster”
Repair in Conventional Erasure Codes: High disk IO & download

- Conventional repair in erasure codes:
  Download & IO = Size of entire message

For (14, 10) RS code, it’s 10x!
- Median of **180 TB** transferred across racks per day for repair
# Breakdown of repairs

<table>
<thead>
<tr>
<th># repairs</th>
<th>% of repairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>98.08</td>
</tr>
<tr>
<td>2</td>
<td>1.87</td>
</tr>
<tr>
<td>3</td>
<td>0.036</td>
</tr>
<tr>
<td>4</td>
<td>$9 \times 10^{-6}$</td>
</tr>
<tr>
<td>$\geq 5$</td>
<td>$9 \times 10^{-9}$</td>
</tr>
</tbody>
</table>

$\Rightarrow$ Code should perform efficient **single** repair.

Rashmi et al., USENIX HotStorage 2013
Outline

• Introduction & Motivation
  – Measurements from Facebook’s Warehouse cluster

• The Piggybacking framework

• Via the Piggybacking framework
  – Best known codes for several settings
  – Comparison with other codes
  – Preliminary practical experiments

• Summary & future work
Piggybacking RS codes: Toy Example

Step 1: Take 2 stripes of (4, 2) Reed-Solomon code

<table>
<thead>
<tr>
<th></th>
<th>a₁</th>
<th>b₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>systematic 1</td>
<td>a₁</td>
<td>b₁</td>
</tr>
<tr>
<td>systematic 2</td>
<td>a₂</td>
<td>b₂</td>
</tr>
<tr>
<td>parity 1</td>
<td>a₁+a₂</td>
<td>b₁+b₂</td>
</tr>
<tr>
<td>parity 2</td>
<td>a₁+2a₂</td>
<td>b₁+2b₂</td>
</tr>
</tbody>
</table>
Piggybacking RS codes: Toy Example

Step 2: Add ‘piggybacks’

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_2$</td>
</tr>
<tr>
<td>$a_1+a_2$</td>
<td>$b_1+b_2$</td>
</tr>
<tr>
<td>$a_1+2a_2$</td>
<td>$b_1+2b_2+a_1$</td>
</tr>
</tbody>
</table>

No additional storage!
Fault-Tolerance

Same fault tolerance as RS code:
can tolerate any 2 failures
Fault-Tolerance

Same fault tolerance as RS code: can tolerate any 2 failures

\[
\begin{align*}
a_1 + a_2 &\quad b_1 + b_2 \\
a_1 + 2a_2 &\quad b_1 + 2b_2 + a_1
\end{align*}
\]
Fault-Tolerance

Same fault tolerance as RS code: can tolerate any 2 failures

<table>
<thead>
<tr>
<th>a₁ + a₂</th>
<th>b₁ + b₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁ + 2a₂</td>
<td>b₁ + 2b₂ + a₁</td>
</tr>
</tbody>
</table>

subtract
Fault-Tolerance

Same fault tolerance as RS code: can tolerate any 2 failures

\[ a_1 + a_2 \]
\[ b_1 + b_2 \]
\[ a_1 + 2a_2 \]
\[ b_1 + 2b_2 + a_1 \]
IO & Download = 3
(instead of 4 as in RS)

Optimal!
**Repair**

IO & Download = 3  
(instead of 4 as in RS)

<table>
<thead>
<tr>
<th></th>
<th>a_1</th>
<th>b_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a_1+a_2</td>
<td></td>
<td>b_1+b_2</td>
</tr>
<tr>
<td>a_1+2a_2</td>
<td></td>
<td>b_1+2b_2+a_1</td>
</tr>
</tbody>
</table>

Optimal!
IO & Download = 3
(instead of 4 as in RS)

Optimal!
### General Piggybacking Framework

**Step 1:** Take 2 (or more) stripes of \((n, k)\) code \(C\)

<table>
<thead>
<tr>
<th>(a_1)</th>
<th>(b_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\vdots)</td>
<td>(\vdots)</td>
</tr>
<tr>
<td>(a_k)</td>
<td>(b_k)</td>
</tr>
<tr>
<td>(f_1(a_1, \ldots, a_k))</td>
<td>(f_1(b_1, \ldots, b_k))</td>
</tr>
<tr>
<td>(\vdots)</td>
<td>(\vdots)</td>
</tr>
<tr>
<td>(f_{n-k}(a_1, \ldots, a_k))</td>
<td>(f_{n-k}(b_1, \ldots, b_k))</td>
</tr>
</tbody>
</table>
General Piggybacking Framework

Step 2: Add `Piggybacks`

<table>
<thead>
<tr>
<th>$a_1$</th>
<th>$b_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$a_k$</td>
<td>$b_k$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$f_1(a_1, \ldots, a_k)$</th>
<th>$f_1(b_1, \ldots, b_k) + p_1(a_1, \ldots, a_k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$f_{n-k}(a_1, \ldots, a_k)$</td>
<td>$f_{n-k}(b_1, \ldots, b_k) + p_{n-k}(a_1, \ldots, a_k)$</td>
</tr>
</tbody>
</table>
Decoding: use decoder of $C$

<table>
<thead>
<tr>
<th>$a_1$</th>
<th>$b_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$a_k$</td>
<td>$b_k$</td>
</tr>
<tr>
<td>$f_1(a_1,\ldots,a_k)$</td>
<td>$f_1(b_1,\ldots,b_k) + p_1(a_1,\ldots,a_k)$</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$f_{n-k}(a_1,\ldots,a_k)$</td>
<td>$f_{n-k}(b_1,\ldots,b_k) + p_{n-k}(a_1,\ldots,a_k)$</td>
</tr>
</tbody>
</table>

- recover $a_1,\ldots,a_k$ as in $C$
- subtract piggybacks; recover $b_1,\ldots,b_k$ as in $C$
General Piggybacking Framework

- Piggybacking does not reduce minimum distance
- Can choose arbitrary functions for piggybacking

Piggybacking functions should be designed such that they can be used for repair

- 3 designs of Piggybacking functions in ISIT paper
Outline

• Introduction & Motivation
  – Measurements from Facebook’s Warehouse cluster

• The Piggybacking framework

• Via the Piggybacking framework
  – Best known codes for several settings
  – Comparison with other codes
  – Preliminary practical experiments

• Summary & future work
Via the Piggybacking framework...

1. “Practical” High-rate MDS codes:
   Lowest known IO & download during repair
• Storage constrained systems: MDS & high-rate

• Then, why not high-rate Minimum Storage Regenerating (MSR) codes?
  – Require exponential block length (*Tamo et al. 2011*)

<table>
<thead>
<tr>
<th>n</th>
<th>k</th>
<th>Block length</th>
<th>IO &amp; Download (% of message size)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RS</td>
<td>Piggy-RS</td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>22</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>210</td>
<td>200</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
### Comparison With Other Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>MDS</th>
<th>Parameters</th>
<th>Block length (in k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-rate MSR</td>
<td>Y</td>
<td>all</td>
<td>exponential</td>
</tr>
<tr>
<td>Product-matrix MSR etc.</td>
<td>Y</td>
<td>low rate</td>
<td>linear</td>
</tr>
<tr>
<td>Rotated-RS</td>
<td>Y</td>
<td>≤ 3 parities</td>
<td>constant</td>
</tr>
<tr>
<td>EVENODD/RDP</td>
<td>Y</td>
<td>≤ 2 parities</td>
<td>linear</td>
</tr>
<tr>
<td>MBR</td>
<td>N</td>
<td>all</td>
<td>linear</td>
</tr>
<tr>
<td>Local repair</td>
<td>N</td>
<td>all</td>
<td>constant</td>
</tr>
<tr>
<td>Piggyback</td>
<td>Y</td>
<td>all</td>
<td>constant / linear</td>
</tr>
</tbody>
</table>

- These are the only other codes that satisfy our requirements of being MDS, high rate and have small block lengths.
- Piggyback codes have smaller repair download and IO than them both.
Via the Piggybacking framework...

2. Binary MDS (vector) codes

- lowest known IO & download during repair (when \#parity>2)

- for all parameters where binary MDS (vector) codes exist
Enabling parity repair in regenerating codes designed for only systematic repair

– efficiency in systematic repair retained
– parity repair improved

Example...
• Regenerating code that repairs systematic nodes efficiently
• Parity node repair performed by downloading all data

systematic node repair: download & IO = 1.5x
• Take two stripes of this code
• Add Piggybacks of parities from 1\textsuperscript{st} stripe to 2\textsuperscript{nd} stripe
• Systematic repair: same efficiency as original code (Piggyback subtracted out in downloaded data)
• Original code:
  Parity repair download & IO = 2x
- Using Piggybacks:

  
  Second parity repair download & IO = 1.5x

<table>
<thead>
<tr>
<th>a</th>
<th>c</th>
<th>e</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>d</td>
<td>f</td>
<td>h</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2a+b+2c</th>
<th>2b+c+d</th>
<th>2e+f+2g +a+b+2c</th>
<th>2f+g+h +a+2b+4d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+b+2c</td>
<td>a+2b+4d</td>
<td>e+f+2g</td>
<td>e+2f+4h</td>
</tr>
</tbody>
</table>

• Using Piggybacks:

  Second parity repair download & IO = 1.5x
Via the Piggybacking framework...

4. Have implemented (14, 10) Piggyback-RS in the Hadoop Distributed File System (HDFS)
   – 35% reduction in disk-IO and download
   – same storage & fault tolerance
   – testing on Facebook’s Warehouse cluster underway
Is connecting to more nodes a concern?

We performed measurements for various data-sizes in the Facebook Warehouse cluster in production.

Piggyback-RS codes:

• Reduce primary metrics of IO & download
• Time to repair also reduces upon connecting to more

Locality/Connectivity not an issue in this setting
Outline

• Introduction & Motivation
  – Measurements from Facebook’s Warehouse cluster

• The Piggybacking framework

• Via the Piggybacking framework
  – Best known codes for several settings
  – Comparison with other codes
  – Preliminary practical experiments

• Summary & future work
Summary

• “Piggybacking” code design framework
• 3 piggyback function designs

• Best known codes for several settings
  – MDS + high-rate + small block length
  – binary MDS (vector)
  – parity repair in regenerating codes

• Implemented in HDFS, testing in Facebook
Future work & open problems

• Comprehensive experiments at Facebook
• Other Piggybacking designs / applications
• Bounds for Piggybacking approach?

• High-rate MDS: Tradeoff between block length & IO/download
Future work & open problems

- Comprehensive experiments at Facebook
- Other Piggybacking designs / applications
- Bounds for Piggybacking approach?

- High-rate MDS: Tradeoff between block length & IO/download

THANKS!