The Attack on RAID-6

James S. Plank
Professor
Department of EECS
University of Tennessee
plank@cs.utk.edu

Wayne State University
December 11, 2007
We all know what RAID-5 is:

- \( k \) disk drives of data
- 1 parity drive:
  \[ (XOR) \]
  - Can tolerate any single disk failure.

For example:
When is RAID-5 Not Enough?

Latent (undetected) sector failure

Plus a drive crash

This combination results in data loss.
Enter RAID-6: Add another “parity” disk

- **Feature #1**: It is built on top of a RAID-5 system: The P drive is simple parity.

- **Feature #2**: The Q drive is *unspecified*, but it must be constructed from an MDS code.

MDS means that the data can be recovered following *any two* failures.
RAID-6 Performance Issues

- **Three performance dimensions**
  - **Encoding**
    - Optimal is $k-1$ XOR operations per coding word.
  - **Modification**
    - Optimal is $2+1$, XOR operations per modified data word.
  - **Decoding**
    - Optimal is $k-1$ XOR operations per failed word.
Current RAID-6 Implementation Options

• All have serious problems, from performance inefficiencies to intellectual property complications.

• Those who want to implement RAID-6 are in a bind!

• Therefore, we must *LIBERATE* them!!!!
What this Talk will Cover:

- **The RAID-6 Liberation Codes:**
  - Near optimal encoding, decoding and modification.
  - Elegant construction.
  - Not patented.
  - Bit-matrix Scheduling Algorithm for faster decoding.

- Jerasure coding library.

- Some other interesting codes.
Coding Primer for RAID-6

• Classic Reed-Solomon Coding [RS60]
  – Based on a *distribution matrix* defined over \( w \)-bit words \( (w \in \{8,16,32\}) \) using “Galois Field Arithmetic” such that any \( k \) rows are invertible.

\[
\begin{array}{cccccc}
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
D_0 & D_1 & D_2 & D_3 & D_4 \\
P & Q & & & \\
\end{array}
\]

\[
\text{Data} \times \text{Data}^{+} \cdot k = \text{Data} + \text{“Parity”}
\]

Addition = XOR

Multiplication well-defined but weird and expensive.

Decode by choosing \( k \) non-failed rows & inverting the result.
Coding Primer for RAID-6

- Cauchy Reed-Solomon Coding [BKK95]
  - Each element of the distribution matrix becomes a $w \times w$ bit matrix. Bit arithmetic is modulo 2. Each piece of data becomes partitioned into $w$ packets.

\[
\begin{array}{cccccc}
I & 0 & 0 & 0 & 0 & 0 \\
0 & I & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 \\
0 & 0 & 0 & I & 0 & 0 \\
0 & 0 & 0 & 0 & I & 0 \\
I & I & I & I & I & 0 \\
X_0 & X_1 & X_2 & X_3 & X_4 & 0 \\
\end{array}
\]

\[
\begin{array}{ccc}
D_0 \\
D_1 \\
D_2 \\
D_3 \\
D_4 \\
\end{array}
\]

\[
\begin{array}{c}
P \\
Q \\
\end{array}
\]

\[
\begin{array}{cccccc}
I & 0 & 0 & 0 & 0 & 0 \\
0 & I & 0 & 0 & 0 & 0 \\
0 & 0 & I & 0 & 0 & 0 \\
0 & 0 & 0 & I & 0 & 0 \\
0 & 0 & 0 & 0 & I & 0 \\
I & I & I & I & I & 0 \\
X_0 & X_1 & X_2 & X_3 & X_4 & 0 \\
\end{array} \times \quad \begin{array}{ccc}
D_0 \\
D_1 \\
D_2 \\
D_3 \\
D_4 \\
\end{array} = \begin{array}{c}
P \\
Q \\
\end{array}
\]

\[
(k+2)w \quad k \times w
\]
Coding Primer for RAID-6

- Let's concentrate on the last two super-rows \((k=5, w=5)\):
Coding Primer for RAID-6

- Let's concentrate on the last two super-rows ($k=5, w=5$):

  - **Observation #1**: The P drive is still parity.
Let's concentrate on the last two super-rows ($k=5, w=5$):

- **Observation #1**: The P drive is still parity.
- **Observation #2**: The dot products are done with if's and XORs, not with additions and multiplications.
- **Observation #3**: Performance of each dot product is the number of one's in the row, minus one.
Cauchy Reed-Solomon Coding: The \( X_i \) are created from the \( w \)-bit numbers using Galois Field arithmetic.

- Obviously, choose the \( X_i \) so that they have a minimal number of ones.

- Better performance with larger \( w \).

- Distinctly non-optimal performance, but quite a bit better than classic Reed-Solomon coding.
• **EVENODD [BBBM95]:** A *parity array code* defined when \( w+1 \) is prime and \( k=w+1 \).

- Encoding/Decoding near optimal.
- Modification = \((3+1)\) per modified word.
Coding Primer for RAID-6

- **RDP [CEG04]**: A parity array code defined when $w+1$ is prime and $k=w$.

$k=4, w=4$

- Encoding/Decoding optimal.
- Modification = $(3+1)$ per modified word.
You can “shorten” the codes, but their performance worsens.

- Therefore, if you want to add/delete data drives, you must either re-encode, or choose a big enough $w$, which will penalize performance.

Moreover, their modification overhead is $(3+1)$ instead of $(2+1)$.

They're patented.
The Inspiration

- Can you define the $X_i$ in bit-matrix coding so that the matrix is MDS, but there is no reliance on Galois Field arithmetic?

Answer: Yes!

The $X_i$ for EVENODD, $k=w=6$

The $X_i$ for RDP, $k=w=6$
Can you define the $X_i$ in bit-matrix coding so that the matrix is MDS, but there is no reliance on Galois Field arithmetic?

Moreover, can you minimize the number of ones in these matrices?

The result was discovering the RAID-6 Liberation Codes.
The RAID-6 Liberation Codes

- $w$ must be a prime number $>2$, and $k \leq w$.
- Let $I_{\rightarrow j}$ be the $w\times w$ identity matrix rotated to the right by $j$ columns.
- Let $O_{i,j}$ be the $w\times w$ matrix composed of all zeros except for a one in row $<i>$, column $<j>$.
- Let $Y(w,i) = <[(w-1)/2]*i>_{w}$.
- $X_{0} = I_{\rightarrow 0} = I$.
- Otherwise, $X_{i} = I_{\rightarrow i} + O_{Y(w,i),Y(w,i)+i-1}$

The $X_{i}$ for the Liberation Code: $k=w=7$
Liberation Code Properties

- They are MDS.
  - See [Plank&Buchsbaum07] for the proof.
  - But we'll sketch it here.

Case #1: Suppose we lose a data drive and $P$.

Each $X_i$ must be an invertible matrix.
Liberation Code Properties

Case #2: Suppose we lose two data drives.

Every sum \((X_i + X_j)\) must be an invertible matrix.
## The RAID-6 Liberation Codes

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Case #1:** Each $X_i$ is invertible – trivial.
- **Case #2:** Each $(X_i + X_j)$ is invertible:
  - Easy when $i = 0$.
  - Note that for $i,j \neq 0$, $(X_i + X_j) \equiv (X_1 + X_{j-i+1})$:
    - Diagonals are $j-i$ columns apart.
    - Extra bits are $<(w-1)/2 \times (j-i)>\_w$ rows apart.
    - Rather involved inductive proof.

Moreover, this is a tight lower bound on # ones.
Performance: Encoding

- \((wk+k-1)/w - 1\) XORs per coding word.
- Factor over optimal: \(1 + 1/w\).
- Asymptotically optimal as \(w \to \infty\).
• Excellent performance for varying $k$ and fixing $w$. 

![Graph showing performance comparison for varying $k$ and fixed $w$.]
Modification

- $1 + (wk + k - 1)/wk$ on average.
  - Asymptotically 2 rather than 3.
Decoding

- Not as good as the others, but not bad either.
  - Requires a trick called “bit matrix scheduling.”
In Relation to Reed-Solomon Coding

- Can't count XOR's, so use implementations to convert RS performance to XOR's.

Recall, the Liberation code for $w=7$ is roughly 1.07.
Bit-matrix Scheduling

- Inverted matrices for Liberation decoding are \textit{not} sparse.

A decoding matrix for $k=5$, $w=5$

This is a problem for efficient decoding:
12.3 XOR's per coding word instead of 4 (optimal)
Bit-matrix Scheduling

• Take inspiration from RDP – intermediate coding elements may be used as starting points.

RDP matrices for $k = w = 4$. 
Bit-matrix Scheduling

• Take inspiration from RDP – intermediate coding elements may be used as starting points.

RDP matrices for $k = w = 4$.

First $Q$ packet only requires 3 XORs when you start with the second $P$ packet.
Bit-matrix Scheduling

- Take inspiration from RDP – intermediate coding elements may be used as starting points.

RDP matrices for \( k = w = 4 \).

2nd \( Q \) packet only requires 3 XORs when you start with the third \( P \) packet.
The idea: you use intermediate results to perform a bit-matrix vector product with fewer XOR's than the number of ones.

A decoding matrix for $k=5$, $w=5$

Row 0: 4 XORs instead of 15 when you start with row 5.
The Algorithm

- Two tables:
  - Start, initialized to -1
  - XOR, initialized to \# ones minus one.

A decoding matrix for $k=5$, $w=5$
The Algorithm

- Find row with minimal XOR.
  - That row will be created from scratch with the given number of XORs.

A decoding matrix for $k=5$, $w=5$
The Algorithm

- For every other row:
  - See if fewer XOR's are required if that row is used as a starting point and update the tables accordingly.

A decoding matrix for k=5, w=5

E.g. Creating row 0 from row 8 requires 18 XORs, so no update.
The Algorithm

- For every other row:
  - See if fewer XOR's are required if that row is used as a starting point and update the tables accordingly.

E.g. However, row 3 only requires 4 XORs: Update the tables.
The Algorithm

- Repeat the process
  - Find row with minimum XORs
  - Update Start/XOR of other rows.

A decoding matrix for $k=5$, $w=5$
The Algorithm

• The Final Result:
  – 45 XORs instead of 123.
  – Works with RDP too (but not EVENODD).

A decoding matrix for $k=5$, $w=5$

<table>
<thead>
<tr>
<th>Start</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>-1</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>
Bit-matrix Scheduling

- Put graphically:
Liberation Code Bottom Line

- Exciting and elegant new code.
- Encoding performance on par with the best.
- Modification performance superior.
- Decoding performance slightly worse but still good.
- Achieves lower bound for # ones in an MDS matrix.
- Not patented.
- Free implementation.
- Paper appearing in USENIX FAST in February.
But, you forgot Lihao's code

- The X-Code doesn't fit RAID-6 paradigm.
- Nor do:
  - WEAVER codes
  - HoVeR codes
  - Pyramid codes
  - Tornado or other LDPC codes
Two More Things: #1

• Jerasure version 1.0: Summer, 2007
  – Implements generic matrix-based coding.
  – Implements generic bit-matrix based coding.
  – Implements Classic Reed-Solomon coding.
  – Implements Optimized RAID-6 Reed-Solomon coding.
  – Implements Cauchy Reed-Solomon coding.
  – Implements Liberation coding.
  – Implements bit-matrix scheduling.

• Not restricted to RAID-6.
• Works for any \( m \) failures.
Two More Things: #2

- The quest to defeat RAID-6 – What's left?
- Discover codes with minimal #1's for \( w=8, 16, 32 \).
  - Massive Monte-Carlo Search with pruning.

\( k=w=4 \)

\( k=w=8 \)

\( k=12 \)
\( w=16 \)

Actually, enumerated 5088 of these.
The Attack on RAID-6

James S. Plank
Professor
Department of EECS
University of Tennessee
plank@cs.utk.edu

Wayne State University
December 11, 2007