Coding Methods for Emerging Storage Systems

Lara Dolecek and Anxiao (Andrew) Jiang

Department of Electrical Engineering, UCLA,
Department of Computer Science, Texas A&M

Asilomar Conference Tutorial, Nov. 2012
Data storage technologies

Storage technologies have revolutionized the way we create, manipulate, and use data.

- **1950-1960s**: Punch cards and magnetic tape
- **1970-1980s**: Optical recording (DVDs), floppy disks
- **1990-2000s**: Hard disk drives
- **2010+**: Era of Flash drives
Non-volatile memories (NMVs) are increasingly ubiquitous
Non-volatile memories (NMVs) are increasingly ubiquitous. 

- Smart Phones
- Tablets and E-Readers
- Data Centers
Non-volatile memories (NMVs) are increasingly ubiquitous.

With the on-going data revolution, applications from consumer electronics to data centers abound.
NVMs today

- NVMs are solid state memory devices that need not be periodically refreshed.
- SSDs have 20x faster read than HDDs due to negligible seek time and consume up to 90% less power.
- Market size over 70 billion USD worldwide.
- Key NVM technologies
  - Flash Memories
  - Phase Change Memories
  - STTRAMs, Memristors
NVMs of the future

Data revolution demands NVMs with

- Increased density at reduced size,
- Improved reliability at increased noise levels,
- Improved endurance and retention,
- Faster read/write access time,
- Compact cross-layer architectures.
Data revolution demands NVMs with

- Increased density at reduced size,
- Improved reliability at increased noise levels,
- Improved endurance and retention,
- Faster read/write access time,
- Compact cross-layer architectures.

**Novel coding methodologies hold promise to address all these challenges.**
Today we will learn about

- Physical characteristics of emerging non-volatile memories,
What is this tutorial about?

Today we will learn about

- Physical characteristics of emerging non-volatile memories,
- Channel models associated with these technologies,
Today we will learn about

- Physical characteristics of emerging non-volatile memories,
- Channel models associated with these technologies,
- Various recent developments in coding for memories, and
What is this tutorial about?

Today we will learn about

- Physical characteristics of emerging non-volatile memories,
- Channel models associated with these technologies,
- Various recent developments in coding for memories, and
- Open problems and future directions in communication techniques for memories.
Part I
Flash technology

- A flash memory block is an array of $2^{20} \sim \text{million cells}$.
- A cell is a floating-gate transistor

Cell array in a flash memory

A flash memory cell (Floating gate)
Flash technology

Terminology

- Single level cell (SLC) stores 1 bit per cell (2 levels)
- Multiple level cell (MLC) stores 2 bit per cell (4 levels)
- Triple level cell (TLC) stores 3 bit per cell (8 levels)

Cell levels correspond to the voltage induced by the number of electrons stored on the gate.
Flash technology

Terminology

- Single level cell (SLC) stores 1 bit per cell (2 levels)
- Multiple level cell (MLC) stores 2 bit per cell (4 levels)
- Triple level cell (TLC) stores 3 bit per cell (8 levels)
- Cell levels correspond to the voltage induced by the number of electrons stored on the gate.
Flash technology

Terminology

- Single level cell (SLC) stores 1 bit per cell (2 levels)
- Multiple level cell (MLC) stores 2 bit per cell (4 levels)
- Triple level cell (TLC) stores 3 bit per cell (8 levels)
- Cell levels correspond to the voltage induced by the number of electrons stored on the gate.
Flash programming – example 1

Flash block

<table>
<thead>
<tr>
<th>Initial state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0</td>
</tr>
</tbody>
</table>
Flash programming – example 1

Flash block

Final state

0 1 2 1
1 1 1 1
2 2 0 1
3 3 2 1
Flash programming – example 2

Flash block

Initial state

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Flash programming – example 2

Flash block

Final state

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
Flash programming – example 3

Flash block

Initial state

0 1 2 1
1 1 1 1
2 2 0 1
3 3 2 1
Flash programming – example 3
Flash programming – example 3

Flash block

Final state

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Flash endurance

- Frequent block erases cause so-called “wearout” due to charge loss on the gates.

- Flash memory lifetime is commonly expressed in terms of the number of program and erase (P/E) cycles allowed before memory is deemed unusable.

- Lifetime:
  - SLC: $\approx 10^6$ P/E cycles
  - MLC: $\approx 10^5$ P/E cycles
  - TLC: $\approx 10^4$ P/E cycles

- With frequent writes 2GB TLC lasts less than 3 months!
Flash inter-cell interference

Floating gate to floating gate inter-cell coupling

- Change in charge in one cell affects voltage threshold of a neighboring cell.
- During write/read, stressed (victim) cell appears weakly programmed.

Figure: Courtesy Micron, Flash Memory Summit Presentation, 2007.
Phase change memories (PCM) model

Two states

- Amorphous – high resistance
- Polycrystalline – low resistance
Phase change memories (PCM) model

Pros
- Erases on the cell level
- Faster access
- More P/E cycles, longer lifetime

Two states
- Amorphous – high resistance
- Polycrystalline – low resistance
Phase change memories (PCM) model

Pros

- Erases on the cell level
- Faster access
- More P/E cycles, longer lifetime

Cons

- Not nearly as dense
- Resistance and voltage drift
- Higher processing cost

Two states

- Amorphous – high resistance
- Polycrystalline – low resistance
Issues

- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

Techniques

- Error correction codes
- Codes for rewriting data
- Rank modulation
- Constrained coding
Further reading (1/2)

Modeling of Flash


Inter-cell Interference

Further reading (2/2)

PCM Model


Cross-Technology Comparisons


Resources at the Annual Flash Memory Summit Page
http://www.flashmemorysummit.com/
Error Correcting Codes
Issues
- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

Techniques
- Error correction codes
- Codes for rewriting data
- Rank modulation
- Constrained coding
**Issues**
- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

**Techniques**
- Error correction codes
- Codes for rewriting data
- Rank modulation
- Constrained coding
Issues

- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

Techniques

- Error correction codes
- Codes for rewriting data
- Rank modulation
- Constrained coding
Error correction is a must for memories

Conventional error correcting codes are in widespread use

- Early SLC technologies used basic Hamming codes
  - Simple to implement but offer limited protection
- BCH codes have become increasingly popular
  - Excellent algebraic codes but increasingly inadequate for NVM channels
- LDPC codes are now being actively investigated
  - Excellent graph-based codes but no performance guarantees in general
Raw error rate for TLC flash

LSB: least significant bit
CSB: center significant bit
MSB: most significant bit

Table: Mapping between Voltage Levels and Triple-bit Words

<table>
<thead>
<tr>
<th>Voltage Level</th>
<th>Triple-bit Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td>1</td>
<td>110</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>001</td>
</tr>
<tr>
<td>5</td>
<td>000</td>
</tr>
<tr>
<td>6</td>
<td>010</td>
</tr>
<tr>
<td>7</td>
<td>011</td>
</tr>
</tbody>
</table>

Data collected in Swanson Lab, UCSD.
Error patterns within a TLC cell

<table>
<thead>
<tr>
<th>Number of bits in symbol that err</th>
<th>Percentage of errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.9617</td>
</tr>
<tr>
<td>2</td>
<td>0.0314</td>
</tr>
<tr>
<td>3</td>
<td>0.0069</td>
</tr>
</tbody>
</table>

- Standard error-correction codes are designed to correct all symbol errors.
Error patterns within a TLC cell

<table>
<thead>
<tr>
<th>Number of bits in symbol that err</th>
<th>Percentage of errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.9617</td>
</tr>
<tr>
<td>2</td>
<td>0.0314</td>
</tr>
<tr>
<td>3</td>
<td>0.0069</td>
</tr>
</tbody>
</table>

- Standard error-correction codes are designed to correct all symbol errors.
- Usage of standard codes: overkill in terms of redundancy.
Error patterns within a TLC cell

<table>
<thead>
<tr>
<th>Number of bits in symbol that err</th>
<th>Percentage of errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.9617</td>
</tr>
<tr>
<td>2</td>
<td>0.0314</td>
</tr>
<tr>
<td>3</td>
<td>0.0069</td>
</tr>
</tbody>
</table>

- Standard error-correction codes are designed to correct all symbol errors.
- Usage of standard codes: overkill in terms of redundancy.
- Instead, design codes for the observed intracell variability.
A code $C$ is defined over the alphabet $\mathcal{A}$ of size $q = 2^m$, where $m$ is a positive integer and $q$ is the number of levels in a flash cell.
Channel coding preliminaries

- A code $C$ is defined over the alphabet $\mathcal{A}$ of size $q = 2^m$, where $m$ is a positive integer and $q$ is the number of levels in a flash cell.
- A Flash cell value is represented by a symbol in $GF(2)^m$. 
Channel coding preliminaries

- A code $C$ is defined over the alphabet $\mathcal{A}$ of size $q = 2^m$, where $m$ is a positive integer and $q$ is the number of levels in a flash cell.
- A Flash cell value is represented by a symbol in $GF(2)^m$.
- A block of $n$ flash cells is represented by a codeword $c$ in $C$, taking value in $(GF(2)^m)^n$. 

Example: $m=3$, $\mathcal{A} = GF(8)$, $n=5$ → (45702) Flash block → codeword
A code $C$ is defined over the alphabet $\mathcal{A}$ of size $q = 2^m$, where $m$ is a positive integer and $q$ is the number of levels in a flash cell.

A Flash cell value is represented by a symbol in $GF(2)^m$.

A block of $n$ flash cells is represented by a codeword $c$ in $C$, taking value in $(GF(2)^m)^n$.

An example:

$m = 3, \mathcal{A} = GF(8), n = 5$

$(45702) \rightarrow (100\ 101\ 111\ 000\ 010)$

Flash block $\rightarrow$ codeword
Definition (Bit-Error Vector)

Given positive integers $t$ and $\ell$, the vector $e = (e_1, \ldots, e_n) \in (GF(2)^m)^n$ is a $[t; \ell]_{2^m}$-bit-error vector when
Definition (Bit-Error Vector)

Given positive integers $t$ and $\ell$, the vector $e = (e_1, \ldots, e_n) \in (GF(2)^m)^n$ is a $[t; \ell]_{2^m}$-bit-error vector when

1. The number of non-zero symbols $e_i$'s, $1 \leq i \leq n$, is at most $t$. 
Bit-error patterns

**Definition (Bit-Error Vector)**

Given positive integers $t$ and $\ell$, the vector $e = (e_1, \ldots, e_n) \in (GF(2)^m)^n$ is a $[t; \ell]_{2^m}$-bit-error vector when

1. The number of non-zero symbols $e_i$'s, $1 \leq i \leq n$, is at most $t$.
2. For each symbol $e_i$, $1 \leq i \leq n$, there are at most $\ell$ non-zero bits.
Bit-error example

Suppose the vector $x$ of length 6 with 3-bit symbols was transmitted (stored):

$$x = (000 \ 110 \ 010 \ 101 \ 000 \ 111)$$
Suppose the vector $x$ of length 6 with 3-bit symbols was transmitted (stored):

$$x = (000\ 110\ 010\ 101\ 000\ 111)$$

Suppose the vector $y$ was received (retrieved):

$$y = (101\ 110\ 000\ 101\ 000\ 011).$$
Suppose the vector $x$ of length 6 with 3-bit symbols was transmitted (stored):

$$x = (000\ 110\ 010\ 101\ 000\ 111)$$

Suppose the vector $y$ was received (retrieved):

$$y = (101\ 110\ 000\ 101\ 000\ 011).$$

Then $[3; 2]_{2^3}$-bit-errors are said to have occurred since there are 3 symbols in error and for each symbol at most 2 bits are in error.
Graded bit-error patterns

**Definition (Graded Bit-Error Vector)**

Given positive integers $t_1$, $t_2$, $\ell_1$ and $\ell_2$, $\ell_1 < \ell_2$, the vector $e = (e_1, \ldots, e_n) \in (GF(2)^m)^n$ is a $[t_1, t_2; \ell_1, \ell_2]_{2^m}$-graded bit-error vector when

1. The number of non-zero symbols $e_i's$, $1 \leq i \leq n$, is at most $t_1 + t_2$.
2. For each symbol $e_i$, there are at most $\ell_2$ non-zero bits.
3. There are at most $t_2$ symbols that have more than $\ell_1$ non-zero bits each.
Graded bit-error patterns

Definition (Graded Bit-Error Vector)

Given positive integers $t_1$, $t_2$, $l_1$ and $l_2$, $l_1 < l_2$, the vector $e = (e_1, \ldots, e_n) \in (GF(2)^m)^n$ is a $[t_1, t_2; l_1, l_2]_{2^m}$-graded bit-error vector when

1. The number of non-zero symbols $e_i$’s, $1 \leq i \leq n$, is at most $t_1 + t_2$. 
Graded bit-error patterns

Definition (Graded Bit-Error Vector)

Given positive integers $t_1$, $t_2$, $l_1$ and $l_2$, $l_1 < l_2$, the vector $e = (e_1, \ldots, e_n) \in (GF(2)^m)^n$ is a $[t_1, t_2; l_1, l_2]_{2^m}$-graded bit-error vector when

1. The number of non-zero symbols $e_i$’s, $1 \leq i \leq n$, is at most $t_1 + t_2$.
2. For each symbol $e_i$, there are at most $l_2$ non-zero bits.
Graded bit-error patterns

**Definition (Graded Bit-Error Vector)**

Given positive integers $t_1$, $t_2$, $\ell_1$ and $\ell_2$, $\ell_1 < \ell_2$, the vector $e = (e_1, \ldots, e_n) \in (\text{GF}(2)^m)^n$ is a $[t_1, t_2; \ell_1, \ell_2]_{2^m}$-graded bit-error vector when

1. The number of non-zero symbols $e_i$’s, $1 \leq i \leq n$, is at most $t_1 + t_2$.
2. For each symbol $e_i$, there are at most $\ell_2$ non-zero bits.
3. There are at most $t_2$ symbols that have more than $\ell_1$ non-zero bits each.
Graded bit-error example

Suppose the vector $\mathbf{x}$ of length 6 with 3-bit symbols was transmitted (stored):

$$\mathbf{x} = (000 \ 110 \ 010 \ 101 \ 000 \ 111)$$
Suppose the vector $x$ of length 6 with 3-bit symbols was transmitted (stored):

$$x = (000 \ 110 \ 010 \ 101 \ 000 \ 111)$$

Suppose the vector $y$ was received (retrieved):

$$y = (101 \ 110 \ 000 \ 101 \ 000 \ 011).$$
Graded bit-error example

- Suppose the vector $\mathbf{x}$ of length 6 with 3-bit symbols was transmitted (stored):
  \[ \mathbf{x} = (000 \ 110 \ 010 \ 101 \ 000 \ 111) \]
- Suppose the vector $\mathbf{y}$ was received (retrieved):
  \[ \mathbf{y} = (101 \ 110 \ 000 \ 101 \ 000 \ 011). \]
- Then $[2, 1; 1, 2]_3$-graded-bit-errors are said to have occurred since there are $2 + 1$ symbols in error where there are at most 2 bits in error for each symbol and there is only 1 symbol that has more than 1 bit in error.
Error-correcting codes

Definition (Bit-Error-Correcting Code)

A $2^m$-ary linear code $C$ is a $[t; \ell]_{2^m}$-bit-error-correcting code if it can correct every $[t; \ell]_{2^m}$-bit error vector.
Error-correcting codes

Definition (Bit-Error-Correcting Code)
A $2^m$-ary linear code $C$ is a $[t; \ell]_{2^m}$-bit-error-correcting code if it can correct every $[t; \ell]_{2^m}$-bit error vector.

Definition (Graded Bit-Error-Correcting Code)
A $2^m$-ary code $C$ is a $[t_1, t_2; \ell_1, \ell_2]_{2^m}$-graded-bit-error-correcting code if it can correct every $[t_1, t_2; \ell_1, \ell_2]_{2^m}$-graded bit error vector.
Recall: tensor product

**Definition (Tensor Product)**

Let $A \in GF(q)^{m \times n}$, $B \in GF(q)^{p \times r}$. Then the tensor product of $A$ and $B$ is defined as the matrix

$$A \otimes B = \begin{pmatrix}
a_{11}B & \cdots & a_{1n}B \\
\vdots & \ddots & \vdots \\
a_{m1}B & \cdots & a_{mn}B
\end{pmatrix} \in GF(q)^{mp \times nr}$$
Recall: tensor product

**Definition (Tensor Product)**

Let \( A \in GF(q)^{m \times n} \), \( B \in GF(q)^{p \times r} \). Then the tensor product of \( A \) and \( B \) is defined as the matrix

\[
A \otimes B = \begin{pmatrix}
a_{11}B & \ldots & a_{1n}B \\
\vdots & \ddots & \vdots \\
a_{m1}B & \ldots & a_{mn}B
\end{pmatrix} \in GF(q)^{mp \times nr}
\]

**Example:**

\[
A = \begin{pmatrix}1 & 1 & 1 & 0\end{pmatrix}, \quad B = \begin{pmatrix}1 & 0 \\
1 & 1\end{pmatrix}
\]
Recall: tensor product

Definition (Tensor Product)

Let $A \in \mathbb{GF}(q)^{m \times n}, B \in \mathbb{GF}(q)^{p \times r}$. Then the tensor product of $A$ and $B$ is defined as the matrix

$$A \otimes B = 
\begin{pmatrix}
  a_{11}B & \ldots & a_{1n}B \\
  \vdots & \ddots & \vdots \\
  a_{m1}B & \ldots & a_{mn}B
\end{pmatrix} \in \mathbb{GF}(q)^{mp \times nr}$$

Example:

$$A = \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix}, B = \begin{pmatrix} 1 & 0 \\
1 & 1 \end{pmatrix}$$

$$A \otimes B = \begin{pmatrix}
  1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
  1 & 1 & 1 & 1 & 0 & 0 & 0 & 0
\end{pmatrix}$$
Tensor product codes [1]

**Theorem (Construction 1)**

- Let $H_1$ be the parity check matrix of a $[m, k_1, \ell]_2$ code $C_1$ (standard $[n, k, e]$ notation).

Tensor product codes [1]

Theorem (Construction 1)

- Let $H_1$ be the parity check matrix of a $[m, k_1, \ell]_2$ code $C_1$ (standard $[n, k, e]$ notation).
- Let $H_2$ be the parity check matrix of a $[n, k_2, t]_{2^{m-k_1}}$ code $C_2$ defined over the alphabet of size $GF(2)^{m-k_1}$.

Tensor product codes [1]

Theorem (Construction 1)

- Let $H_1$ be the parity check matrix of a $[m, k_1, \ell]_2$ code $C_1$ (standard $[n, k, e]$ notation).
- Let $H_2$ be the parity check matrix of a $[n, k_2, t]_{2^m-k_1}$ code $C_2$ defined over the alphabet of size $GF(2)^{m-k_1}$.
- Then, $H_A = H_2 \otimes H_1$ is the parity check matrix of a $[t; \ell]_{2^m}$-bit-error-correcting code $C_A$ of length $n$.

Example of a tensor product code

- Consider the parity check matrix $H_1$ of a $[3, 1, 1]_2$ code $C_1$:

$$H_1 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$
Example of a tensor product code

- Consider the parity check matrix $H_1$ of a $[3, 1, 1]_2$ code $C_1$:
  $$H_1 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

- Consider the parity check matrix $H_2$ of a $[4, 2, 1]_4$ code $C_2$ where $\alpha$ is a primitive element of $GF(4)$:
  $$H_2 = \begin{pmatrix} 1 & 1 & \alpha & \alpha^2 \\ 0 & 1 & \alpha^2 & \alpha \end{pmatrix}$$
Example of a tensor product code

- Consider the parity check matrix $H_1$ of a $[3, 1, 1]_2$ code $C_1$:

  $H_1 = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$

- Consider the parity check matrix $H_2$ of a $[4, 2, 1]_4$ code $C_2$ where $\alpha$ is a primitive element of $GF(4)$:

  $H_2 = \begin{pmatrix} 1 & 1 & \alpha & \alpha^2 \\ 0 & 1 & \alpha^2 & \alpha \end{pmatrix}$

- Map

  $0 \rightarrow \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $1 \rightarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $\alpha \rightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, $\alpha^2 \rightarrow \begin{pmatrix} 1 \\ 1 \end{pmatrix}$
Example of a tensor product code

Consider the parity check matrix $H_1$ of a $[3, 1, 1]_2$ code $C_1$:

$$H_1 = \begin{pmatrix} 1 & \alpha & \alpha^2 \end{pmatrix}$$

Consider the parity check matrix $H_2$ of a $[4, 2, 1]_4$ code $C_2$ where $\alpha$ is a primitive element of $GF(4)$:

$$H_2 = \begin{pmatrix} 1 & 1 & \alpha & \alpha^2 \\ 0 & 1 & \alpha^2 & \alpha \end{pmatrix}$$
Example of a tensor product code

- Consider the parity check matrix $H_1$ of a $[3, 1, 1]_2$ code $C_1$:

$$H_1 = \begin{pmatrix} 1 & \alpha & \alpha^2 \end{pmatrix}$$

- Consider the parity check matrix $H_2$ of a $[4, 2, 1]_4$ code $C_2$ where $\alpha$ is a primitive element of $GF(4)$:

$$H_2 = \begin{pmatrix} 1 & 1 & \alpha & \alpha^2 \\ 0 & 1 & \alpha^2 & \alpha \end{pmatrix}$$

- The parity check matrix of a $[1; 1]_8$-bit-error-correcting code $C_A$ of length $n = 4$ with 3-bit symbols is:

$$H_2 \otimes H_1 = \begin{pmatrix} 1 & \alpha & \alpha^2 & 1 & \alpha & \alpha^2 & \alpha & \alpha^2 & 1 & \alpha^2 & 1 & \alpha \end{pmatrix}.$$
Decoding algorithm

- Suppose $c \in C_A$ was transmitted, and $y = c + e$ was received where $e$ is a $[t; \ell]_{2^m}$-bit error vector.
- Two-step syndrome decoding:
  1. Decoder for $C_2$: input is the scaled syndrome $s$ of $y$, output is the decoded string $r$.
  2. Decoder for $C_1$: input is the vector of syndromes $r$, output is the error-vector $e$. 
Construction of a $[t_1, t_2; \ell_1, \ell_2]$-graded bit-error-correcting code

Suppose $H_1$ is the parity check matrix of a $[m, k_1, \ell_2]_2$ code $C_1$ such that $H_1$ is $\begin{bmatrix} H'_1 \\ H''_1 \end{bmatrix}$ where $H'_1$ itself is the parity check matrix of a $[m, m - r', \ell_1]_2$ code (here $r' < m - k_1$).
Construction of a $[t_1, t_2; \ell_1, \ell_2]$-graded bit-error-correcting code

- Suppose $H_1$ is the parity check matrix of a $[m, k_1, \ell_2]_2$ code $C_1$ such that $H_1$ is $\begin{bmatrix} H'_1 \\ H''_1 \end{bmatrix}$ where $H'_1$ itself is the parity check matrix of a $[m, m - r', \ell_1]_2$ code (here $r' < m - k_1$).

- $H'_1$ is a $r'$ by $m$ matrix, $H''_1$ is a $r''$ by $m$ matrix for $r'' = r - r'$.
- One choice for $H_1$ is a BCH matrix.
Construction of a $[t_1, t_2; \ell_1, \ell_2]$-graded bit error-correcting code

- Suppose $H_2$ is the parity check matrix of a $[n, k_2, t_1 + t_2]_{2r'}$ code.
Construction of a \([t_1, t_2; \ell_1, \ell_2]\)-graded bit error-correcting code

- Suppose \(H_2\) is the parity check matrix of a \([n, k_2, t_1 + t_2]_{2r'}\) code.
- Suppose \(H_3\) is the parity check matrix of a \([n, k_3, t_2]_{2r''}\) code.
Construction of a $[t_1, t_2; \ell_1, \ell_2]$-graded bit error-correcting code

- Suppose $H_2$ is the parity check matrix of a $[n, k_2, t_1 + t_2]_{2^{r'}}$ code.
- Suppose $H_3$ is the parity check matrix of a $[n, k_3, t_2]_{2^{r''}}$ code.

**Theorem (Construction 2)**

Then $H_B$ is the parity check matrix of a $[t_1, t_2; \ell_1, \ell_2]_{2^m}$-graded bit error correcting code, where

$$H_B = \left( \begin{array}{c} H_2 \otimes H_1' \\ H_3 \otimes H_1'' \end{array} \right).$$
Using sphere-packing bound argument, it follows that the excess redundancy of $C_B$ is about $t_2 \log(n)$. The code is asymptotically optimal.

Construction 1 is also a graded-bit-error correcting code. Construction 2 offers better redundancy than Construction 1 when $(\ell_2 - \ell_1)t_1/t_2 > \log(n)/\log(m)$.

Further simplifications are possible for special cases of the code parameters.
Evaluation

We compared the following rate-0.86 codes:

1. Three binary $[2^{12}, 3531, 47]_2$ codes with the same error correction capability for the LSB, CSB, and MSB pages,

2. Three binary codes with different error correction capability for the LSB, CSB, and MSB pages: codes $[2^{12}, 3351, 62]_2$, $[2^{12}, 3339, 63]_2$, and $[2^{12}, 3915, 15]_2$,

3. A non-binary $[2^{12}, 3338, 84]_4$ code over $GF(4)$ applied to the CSB and LSB sharing the same physical cells and a binary $[2^{12}, 3915, 15]_2$ code applied to the MSB (scheme A),

4. A non-binary $[2^{12}, 3534, 80]_8$ code over $GF(8)$ which corrects errors in a group of LSB, CSB, and MSB pages sharing the same physical cells, and

Coding can extend lifetime

![Error Rates of Codes Applied to TLC Flash](image)

- **Error Rates of Codes Applied to TLC Flash**
- Various codes are compared, including:
  - \([4096,3534,80]_8\)
  - \([4096,3531,47]_2\)
  - Scheme A
  - \([81,7;1,3]_2\) Code
  - Binary Codes

The graph shows the page error rate vs. P/E cycles for different codes, demonstrating the benefits of error correction in extending the lifetime of TLC flash memories.
Coding can extend lifetime

![Error Rates of Codes Applied to TLC Flash](image.png)

- Error Rates of Codes Applied to TLC Flash
- [4096, 3534, 80]
- [4096, 3531, 47]
- Scheme A
- [81, 7; 1, 3] Code
- Binary Codes

Summary and Future Directions

Error Correcting Codes
Codes for Rewriting Data
Rank Modulation
Constrained Coding

Coding can extend lifetime

![Graph showing error rates](image.png)
Issues
- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

Techniques
- Error correction codes
- Codes for rewriting data
- Rank modulation
- Constrained coding
Issues
- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

Techniques
- Error correction codes
- Codes for rewriting data
- Rank modulation
- Constrained coding
Extracting soft information

- Recall: Bit-level distribution
- 1 read compares against a single threshold
Extracting soft information

- Idea: multiple word line reads
Extracting soft information

- Idea: multiple word line reads
- 2 reads compare against two thresholds
Extracting soft information

- Idea: multiple word line reads
- 2 reads compare against two thresholds

Fig. 3. Equivalent discrete memoryless channel models for SLC with (a) two reads and (b) three reads with distinct word-line voltages.

A. PAM-Gaussian Model

This subsection describes how to select word-line voltages to achieve MMI in the context of a simple model of the flash cell read channel as PAM transmission with Gaussian noise. MMI word-line voltage selection is presented using three examples: SLC with two reads, SLC with three reads, and 4-level MLC with six reads.

For SLC flash memory, each cell can store one bit of information. Fig. 2 shows the model of the threshold voltage distribution as a mixture of two identically distributed Gaussian random variables. When either a “0” or “1” is written to the cell, the threshold voltage is modeled as a Gaussian random variable with variance $N_0/2$ and either mean $-\sqrt{E_s}$ (for “1”) or mean $+\sqrt{E_s}$ (for “0”).

For SLC with two reads using two different word line voltages, which should be symmetric (shown as $q$ and $-q$ in Fig. 2), the threshold voltage is quantized according to three regions shown in Fig. 2: the white region, the black region, and the union of the light and dark gray regions (which essentially corresponds to an erasure). This quantization produces the effective discrete memoryless channel (DMC) model as shown in Fig. 3 (a) with input $X \in \{0, 1\}$ and output $Y \in \{0, e, 1\}$.

Assuming $X$ is equally likely to be 0 or 1, the MI $I(X; Y)$ between the input $X$ and output $Y$ can be maximized to determine the best thresholds (here $q$ and $-q$).
Extraction of soft information

- Idea: multiple word line reads
- 2 reads compare against two thresholds

Maximize mutual information of the induced channel to determine the best thresholds (here $q$ and $-q$)
Extracting soft information

- Idea: multiple word line reads
Extracting soft information

- Idea: multiple word line reads
- 3 reads compare against three thresholds

Maximize mutual information of the induced channel to determine the best thresholds (here $q_1$, $-q_1$ and 0)
Coding can improve reliability [1]

**Figure:** Performance comparison for 0.9-rate LDPC and BCH codes of length $n = 9100$.

![Frame Error Rate vs. Raw Bit Error Rate (SLC Gaussian Model)](image)

Coding can improve reliability [1]

**Figure:** Performance comparison for 0.9-rate LDPC and BCH codes of length \( n = 9100 \).

- **Caution:** AWGN-optimized LDPC codes may not be the best for the quantized Flash channel!

Coding can accelerate write access

- Recall Flash programming
Coding can accelerate write access

- Recall Flash programming
- In practice incremental pulse programming is used, a.k.a. guess and verify.
- Latency increases with number of levels.
Coding can accelerate write access

- Recall Flash programming

- In practice incremental pulse programming is used, a.k.a. guess and verify.

- Latency increases with number of levels.

- If one allows for “dirty writes”, it suffices to correct errors in only one direction.
Asymmetric limited-magnitude error-correcting codes [1]

Definition (ALM Code)

Let $C_1$ be a code over the alphabet $Q_1$. The code $C$ over the alphabet $Q$ (with $|Q| > |Q_1| = q_1 = \ell + 1$) is defined as $C = \{ x = (x_1, x_2, \ldots, x_n) \in Q^n \mid x \mod q_1 \in C_1 \}$. 

Theorem

Code $C$ corrects $t$ asymmetric errors of limited magnitude $\ell$ if code $C_1$ corrects $t$ symmetric errors. New construction inherits encoding/decoding complexity of the underlying code. Connections with additive number theory (Varshamov 1970s).

Asymmetric limited-magnitude error-correcting codes [1]

Definition (ALM Code)

Let $C_1$ be a code over the alphabet $Q_1$. The code $C$ over the alphabet $Q$ (with $|Q| > |Q_1| = q_1 = \ell + 1$) is defined as

$C = \{ x = (x_1, x_2, \ldots, x_n) \in Q^n \mid x \mod q_1 \in C_1 \}$. 

Theorem

*Code $C$ corrects $t$ asymmetric errors of limited magnitude $\ell$ if code $C_1$ corrects $t$ symmetric errors.*

- New construction inherits encoding/decoding complexity of the underlying code.
- Connections with additive number theory (Varshamov 1970s).

Further reading (1/2)

Algebraic codes


Further reading (2/2)

Graph-based codes


Codes for Rewriting Data
Issues

- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

Techniques

- Error correction codes
- Codes for rewriting data
- Rank modulation
- Constrained coding
**Issues**
- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

**Techniques**
- Error correction codes
- Codes for rewriting data
- Rank modulation
- Constrained coding
Write-once-memories (WOMs)

- The memory cells represent bits that are irreversibly programmed from “0” to “1”.
- What is the total number of bits that is possible to write over $n$ cells in $t$ writes?
- WOM codes were introduced by Rivest and Shamir “How to reuse a write-once memory”, *Information and Control*, 1982.
- WOM codes were originally proposed for punch cards and optical disks.
Rivest-Shamir WOM code

<table>
<thead>
<tr>
<th>Information</th>
<th>First Generation</th>
<th>Second Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>011</td>
</tr>
</tbody>
</table>
Encoding example 1

Rivest-Shamir WOM code:

<table>
<thead>
<tr>
<th>Information</th>
<th>First Generation</th>
<th>Second Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>011</td>
</tr>
</tbody>
</table>
Encoding example 1

Rivest-Shamir WOM code:

<table>
<thead>
<tr>
<th>Information</th>
<th>First Generation</th>
<th>Second Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>011</td>
</tr>
</tbody>
</table>

Write-1: 01  →  Encode: 001.
## Encoding example 1

### Rivest-Shamir WOM code:

<table>
<thead>
<tr>
<th>Information</th>
<th>First Generation</th>
<th>Second Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>011</td>
</tr>
</tbody>
</table>

- **Write-1:** 01 → **Encode:** 001.
- **Write-2:** 10 → **Encode:** 101.
Encoding example 1

Rivest-Shamir WOM code:

<table>
<thead>
<tr>
<th>Information</th>
<th>First Generation</th>
<th>Second Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>011</td>
</tr>
</tbody>
</table>

- Write-1: 01 $\rightarrow$ Encode: 001.
- Write-2: 10 $\rightarrow$ Encode: 101.
Encoding example 2

Rivest-Shamir WOM code:

<table>
<thead>
<tr>
<th>Information</th>
<th>First Generation</th>
<th>Second Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>011</td>
</tr>
</tbody>
</table>
Encoding example 2

Rivest-Shamir WOM code:

<table>
<thead>
<tr>
<th>Information</th>
<th>First Generation</th>
<th>Second Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>011</td>
</tr>
</tbody>
</table>

- Write-1: 01 → Encode: 001.
Encoding example 2

Rivest-Shamir WOM code:

<table>
<thead>
<tr>
<th>Information</th>
<th>First Generation</th>
<th>Second Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>011</td>
</tr>
</tbody>
</table>

- Write-1: 01 → Encode: 001.
- Write-2: 01 → Encode: 001. No change.
System model

Definition (WOM Model)

The memory state is modeled as a vector $y^j$ of length $n$ where $j$ is the current write (or generation). Each element $y^j_i$, $1 \leq i \leq n$, takes values in the set $\{0, 1, \ldots, q - 1\}$. On write $j$, the encoder writes one of $M_j$ messages to the memory by updating $y^{j-1}$ to $y^j$ while satisfying the WOM-constraint $y^j \geq y^{j-1}$. 
System model

Definition (WOM Model)

The memory state is modeled as a vector $\mathbf{y}^j$ of length $n$ where $j$ is the current write (or generation). Each element $y_i^j$, $1 \leq i \leq n$, takes values in the set $\{0, 1, \ldots, q - 1\}$. On write $j$, the encoder writes one of $M_j$ messages to the memory by updating $\mathbf{y}^{j-1}$ to $\mathbf{y}^j$ while satisfying the WOM-constraint $\mathbf{y}^j \geq \mathbf{y}^{j-1}$.

Definition (Sum rate)

If $M_j$ codewords can be represented at generation $j$, then generation $j$ has rate $\frac{1}{n} \log(M_j)$. The sum rate is the sum of rates across generations.
WOM capacity

- Achievable rate-region computed by Fu and Han Vinck in 1999.
- The capacity (maximum achievable sum-rate) using $t$ writes and $q$-ary cells is
  \[ \log \left( \frac{t + q - 1}{q - 1} \right). \]
- Popular setting: binary 2-write WOM, the capacity is
  \[ \log 3 \approx 1.58. \]
- Random coding achieves capacity - the bound is tight.
Early results on binary WOM codes

Rivest-Shamir WOM code:

<table>
<thead>
<tr>
<th>Information</th>
<th>First Generation</th>
<th>Second Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>011</td>
</tr>
</tbody>
</table>

- Recall that the capacity is 1.58.
- Rivest-Shamir WOM code already achieves sum rate of 
\[(\log 4 + \log 4)/3 = 4/3\] with \(M_1 = 4\), \(M_2 = 4\), and \(n = 3\).
Early results on binary WOM codes

- Here is another example from Rivest and Shamir

```
AHGGFYL wEZYr XfpnD WVz UdjoTwkeltdu
CSRcQ iozPpi huxyO zsjsnlwvcqgfkbm
BNMzLbgmK ut bng f wJwrhkuvxymjpsqcei
Ikmq lckuwteosd jvubdfget p y x n l hrza
```

- Symbol X in position \((i, j)\) is encoded as the number \(32 \times i + j\) over \(GF(2)^7\).
- First generation is in UPPER CASE.
- Example:
  - Write-1: symbol “Q” \(\rightarrow\) Encode \(32 \times 1 + 4\) in binary: 
    \((0, 1, 0, 0, 1, 0, 0)\).
  - Write-2: symbol “b” \(\rightarrow\) Encode \(32 \times 1 + 30\) in binary: 
    \((0, 1, 1, 1, 1, 1, 0)\).
- Rate \(\approx 1.34\).
Early results on binary WOM codes

- Here is another example from Rivest and Shamir

\[
\begin{array}{cccccccccccccccccccc}
A & H & G & F & Y & L & w & E & Z & Y & r & X & f & p & n & D & W & V & z & U & d & j & o & T & w & k & e & l & t & d & u \\
C & S & R & c & Q & i & o & z & P & p & i & h & u & e & x & y & O & z & s & j & s & n & i & w & v & c & q & g & f & k & b & m \\
B & N & M & z & L & b & g & m & K & u & t & b & n & g & f & w & J & w & r & h & k & v & x & y & m & j & p & s & o & q & c & i \\
I & k & m & q & l & c & k & u & w & t & e & o & s & d & j & v & u & b & d & f & g & e & t & p & y & x & n & l & h & r & z & a \\
\end{array}
\]

- Symbol X in position \((i, j)\) is encoded as the number \(32 \times i + j\) over \(GF(2)^7\).
- First generation is in UPPER CASE.
- Example:
  - Write-1: symbol “Q” \(\rightarrow\) Encode \(32 \times 1 + 4\) in binary : 
    \((0, 1, 0, 0, 1, 0, 0)\).
  - Write-2: symbol “b” \(\rightarrow\) Encode \(32 \times 1 + 30\) in binary : 
    \((0, 1, 1, 1, 1, 1, 0)\).
- Rate \(\approx 1.34\).
Sufficient conditions for achieving capacity

- Suppose there are $q$ total symbols and $t$ rewrites.
Sufficient conditions for achieving capacity

- Suppose there are $q$ total symbols and $t$ rewrites.
- Let $\alpha_{t-j,q,m}$ be $Pr(y^j_i = m)$ and let
  \[ Pr(y^{t-(j+1)}_i = m|y^{t-j}_i = r) \]
  be denoted as $\alpha_{t-i,q,m|r}$.
Sufficient conditions for achieving capacity

- Suppose there are $q$ total symbols and $t$ rewrites.
- Let $\alpha_{t-j,q,m}$ be $Pr(y_i^j = m)$ and let $Pr(y_i^{t-(j+1)} = m|y_i^{t-j} = r)$ be denoted as $\alpha_{t-i,q,m|r}$.
- Then, as $n \to \infty$, the symbol distributions for a random capacity achieving code are given by the expressions [1]:

$$\alpha_{t-0,q,0} = \frac{t}{t-1+q} \alpha_{t-0,q,0} (q+t-m-2)$$

$$\alpha_{t-0,q,m} = \left(\frac{q+t-2}{t-1}\right) \alpha_{t-0,q,m}$$

$$\alpha_{t-i,q,m|r} = \alpha(t-1)-(i-1),q-r,m-r$$

Sufficient conditions for achieving capacity

Suppose there are \( q \) total symbols and \( t \) rewrites.

Let \( \alpha_{t-j, q, m} \) be \( \Pr(y^j_i = m) \) and let
\[
\Pr(y^{t-(j+1)}_i = m | y^{t-j}_i = r) \text{ be denoted as } \alpha_{t-i, q, m|r}.
\]

Then, as \( n \to \infty \), the symbol distributions for a random capacity achieving code are given by the expressions [1]:

\[
\begin{align*}
\alpha_{t-0, q, 0} &= \frac{t}{t-1+q} \\
\alpha_{t-0, q, m} &= \frac{\alpha_{t-0, q, 0}}{\binom{q+t-2}{t-1}} \left( q+t-m-2 \right) \\
\alpha_{t-i, q, m|r} &= \alpha(t-1)-(i-1), q-r, m-r
\end{align*}
\]

We refer to the resulting code as a random recursive (RR) code.

An illustration

- Distributions conditioned on the previous level.

Capacity achieving RR code exhibits regularity.
Coset Coding [1]

- Consider an error correction code $C$ with parity check matrix $H$.
- Construct a 2-write WOM code as follows:
  1. First write: Encode message $m_1$ into codeword $c_1$ such that $Hc_1 = m_1$ and $c_1$ is “low-weight”.
  2. Second write: Encode message $m_2$ into codeword $c_2 = c_1 + c'_2$ such that $Hc'_2 = m_1 + m_2$ and $c'_2 \n c_1$.
- “Low-weight” means less than the minimum distance. Large minimum distance implies large first generation codebook.

Rivest-Shamir code interpretation via coset coding

- Let’s consider a 3-repetition code with parity check matrix

\[ H = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \]

- On first write: \( Hc_1 = m_1 \)

<table>
<thead>
<tr>
<th>Information ((m_1))</th>
<th>First Generation ((c_1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
</tr>
</tbody>
</table>
Rivest-Shamir code interpretation via coset coding

- Let’s consider a 3-repetition code with parity check matrix

\[
H = \begin{bmatrix}
1 & 1 & 0 \\
1 & 0 & 1 \\
1 & 0 & 1
\end{bmatrix}
\]

- On first write: \( HC_1 = m_1 \)

<table>
<thead>
<tr>
<th>Information ((m_1))</th>
<th>First Generation ((c_1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
</tr>
</tbody>
</table>
Rivest-Shamir code interpretation via coset coding

- Let’s consider a 3-repetition code with parity check matrix

\[
H = \begin{bmatrix}
1 & 1 & 0 \\
1 & 0 & 1 \\
1 & 0 & 1
\end{bmatrix}
\]

- On second write: Find \(c'_1\) such that \(Hc'_1 = (m_1 + m_2)\) and \(c'_1 \not\equiv c_1\).

<table>
<thead>
<tr>
<th>Information ((m_2))</th>
<th>Second Generation ((c_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
</tr>
</tbody>
</table>
Rivest-Shamir code interpretation via coset coding

- Let’s consider a 3-repetition code with parity check matrix

\[
H = \begin{bmatrix}
1 & 1 & 0 \\
1 & 0 & 1 \\
1 & 0 & 1
\end{bmatrix}
\]

- On second write: Find \( c'_1 \) such that \( Hc'_1 = (m_1 + m_2) \) and \( c'_1 \neq c_1 \). \( Hc'_1 = (11) \rightarrow c'_1 = (100) \)

<table>
<thead>
<tr>
<th>Information ((m_2))</th>
<th>Second Generation ((c_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
</tr>
</tbody>
</table>
Rivest-Shamir code interpretation via coset coding

Let’s consider a 3-repetition code with parity check matrix

\[ H = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \]

On second write: Find \( c_1' \) such that \( Hc_1' = (m_1 + m_2) \) and \( c_1' \neq c_1 \). \( Hc_1' = (11) \rightarrow c_1' = (100) \rightarrow c_1 + c_1' = (100) + (001) \)

<table>
<thead>
<tr>
<th>Information (( m_2 ))</th>
<th>Second Generation (( c_2 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
</tr>
</tbody>
</table>
Recent results on binary WOM codes

- Coset coding idea was recently generalized by Wu [1] and Yaakobi et al. [2].
- Using first-order $[16, 5, 3]$ Reed-Muller code as the error correcting code yields sum-rate of 1.4566. (capacity is 1.58!)
- Yields a recursive construction of high-rate $t$-write binary WOM codes.
- Coset coding idea was further refined by Shpilka [3]: instead of one fixed matrix, choose an appropriate element from an ensemble of matrices...capacity achieving scheme!

Constructing high-rate non-binary WOM codes

1. Begin with good $t$-write WOM codes over smaller alphabets.
2. Define a mapping into a code with a larger alphabet.
3. Construct a new non-binary code based on these constituents.
Construction I

- Consider \( q = 2^m \).
- Every cell value is converted into an \( m \)-bit binary vector using binary representation.
- For \( 1 \leq i \leq m \), the \( i \)th bits in each cell compromise a binary \( t \)-write WOM code.
- These \( m \) binary WOM codes update values independently.
- Since each constituent code obeys the WOM constraint, resultant code also obeys the WOM constraint.
Lemma

If $q = 2^m$ and there exists a binary $t$-write WOM code of sum rate $R$, then there exists a $q$-ary $t$-write WOM code of sum rate $mR$.

- Immediate extension to the case when $q = s^m$ for $s > 2$. 
An example of Construction I using Rivest and Shamir write-twice code for $q = 8$ and $n = 3$

Rivest-Shamir WOM code

<table>
<thead>
<tr>
<th>Information</th>
<th>First Generation</th>
<th>Second Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>011</td>
</tr>
</tbody>
</table>

Construction I

<table>
<thead>
<tr>
<th>Write no.</th>
<th>Data bits</th>
<th>Encoding by RS code</th>
<th>Encoded values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(11, 01, 10)</td>
<td>(100,001,010)</td>
<td>(4,1,2)</td>
</tr>
<tr>
<td>2</td>
<td>(00, 11, 01)</td>
<td>(101,111,110)</td>
<td>(5,7,6)</td>
</tr>
</tbody>
</table>
Construction II

- Assume $3 \mid q$ and partition the cell levels into three groups: 
  \{0, \ldots, q/3 - 1\}, \{q/3, \ldots, 2q/3 - 1\}, \{2q/3, \ldots, 3q - 1\}\], each of size $q/3$.

- Let $C$ be a binary two-write WOM code of length $n$.

- First write – write two words:
  - A binary codeword $u \in \{0, 1\}^n$ according to the first write in $C$.
  - A message word $w \in \{0, \ldots, q/3 - 1\}^n$.
  - For each $i, 1 \leq i \leq n$, if $u_i = 0$ write $w_i$; if $u_i = 0$ write $w_i + q/3$.

- Second write – write two words:
  - A binary codeword $u \in \{0, 1\}^n$ according to the second write in $C$.
  - A message word $w \in \{0, \ldots, q/3 - 1\}^n$.
  - For each $i, 1 \leq i \leq n$, if $u_i = 0$ write $w_i + q/3$; if $u_i = 0$ write $w_i + 2q/3$. 
Lemma

If $q = m(t + 1)$ and there exists a binary $t$-write WOM code of sum rate $R$, then there exists a $q$-ary $t$-write WOM code of sum rate $R + t \log(m)$.

- Immediate extension to the case when $q = m(s + t - 1)$ where there exists a $s$-ary $t$-write WOM code for $s > 2$. 
An example of Construction II using Rivest and Shamir write-twice code for $q = 9$ and $n = 3$

**Rivest-Shamir WOM code**

<table>
<thead>
<tr>
<th>Information</th>
<th>First Generation</th>
<th>Second Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>011</td>
</tr>
</tbody>
</table>

**Construction II**

<table>
<thead>
<tr>
<th>Write no.</th>
<th>Information</th>
<th>RS code + info</th>
<th>Encoded values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0,1),(0,1,2)</td>
<td>(001),(012)</td>
<td>(0,1,5)</td>
</tr>
<tr>
<td>2</td>
<td>(0,0),(2,1,2)</td>
<td>(111),(212)</td>
<td>(8,7,8)</td>
</tr>
</tbody>
</table>
Results and comparison

- In general, Construction I gives better results for smaller $q$, Construction II gives better results for larger $q$.

<table>
<thead>
<tr>
<th>$q$</th>
<th>Achieved sum-rate</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.9856 (I)</td>
<td>3.3219</td>
</tr>
<tr>
<td>8</td>
<td>4.4784 (I)</td>
<td>5.1699</td>
</tr>
<tr>
<td>16</td>
<td>6.3083 (I)</td>
<td>7.0875</td>
</tr>
<tr>
<td>32</td>
<td>8.9684 (II)</td>
<td>9.0444</td>
</tr>
<tr>
<td>64</td>
<td>10.3083 (II)</td>
<td>11.0244</td>
</tr>
<tr>
<td>128</td>
<td>12.3083 (II)</td>
<td>13.0112</td>
</tr>
</tbody>
</table>

- Very high rate two-write non-binary WOM codes (outperform [1,2]).

Facets of WOM and rewriting codes

- Reduction of write amplification with a help of WOM code (Luijie et al., 2012)
Facets of WOM and rewriting codes

- Reduction of write amplification with a help of WOM code (Luijie et al., 2012)
- WOM codes robust to inter-cell interference (Li, 2011)
Facets of WOM and rewriting codes

- Reduction of write amplification with a help of WOM code (Luijie et al., 2012)
- WOM codes robust to inter-cell interference (Li, 2011)
- Error correction/detection in conjunction with WOM property e.g., polar codes (Burshtein and Strugatski, 2012)
Facets of WOM and rewriting codes

- Reduction of write amplification with a help of WOM code (Luijje et al., 2012)
- WOM codes robust to inter-cell interference (Li, 2011)
- Error correction/detection in conjunction with WOM property e.g., polar codes (Burshtein and Strugatski, 2012)
- Codes that maximize the number of writes between subsequent erases (cf. Floating codes invented by A. Jiang et al., 2007)
Further Reading (1/2)

Early Capacity and Code Construction Results


Recent Works


See also WOM Codes Session at IEEE ISIT 2012 as well as papers at ISIT 2007+.
Part II
Rank Modulation
Difficulties with Flash Memories

- To program (i.e., write) cells, we can only increase cell levels due to the high cost of block erasure.
Difficulties with Flash Memories

- To program (i.e., write) cells, we can only increase cell levels due to the high cost of block erasure.
- Charge injection is a noisy random process.
Difficulties with Flash Memories

- To program (i.e., write) cells, we can only increase cell levels due to the high cost of block erasure.
- Charge injection is a noisy random process.
- Thousands of cells are programmed in parallel sharing the same programming voltage. The worse behavior of cells determines performance.
Difficulties with Flash Memories

- To program (i.e., write) cells, we can only increase cell levels due to the high cost of block erasure.
- Charge injection is a noisy random process.
- Thousands of cells are programmed in parallel sharing the same programming voltage. The worse behavior of cells determines performance.
- There are various mechanisms for noise: Read/Write disturbs, charge leakage, inter-cell interference, random noise.
Challenges to Flash Memories

- How to program cells reliably? Can we remove the risk of overshooting (i.e., injecting too much charge into cells)?
Challenges to Flash Memories

- How to program cells reliably? Can we remove the risk of overshooting (i.e., injecting too much charge into cells)?
- When errors appear, can we physically correct cell levels (instead of just finding out what the errors are via ECC decoding) without block erasures?
Challenges to Flash Memories

- How to program cells reliably? Can we remove the risk of overshooting (i.e., injecting too much charge into cells)?
- When errors appear, can we physically correct cell levels (instead of just finding out what the errors are via ECC decoding) without block erasures?
- Can we rewrite data easily without block erasures?
Challenges to Flash Memories

- How to program cells reliably? Can we remove the risk of overshothing (i.e., injecting too much charge into cells)?
- When errors appear, can we physically correct cell levels (instead of just finding out what the errors are via ECC decoding) without block erasures?
- Can we rewrite data easily without block erasures?
- Can we balance the cell levels in the same page by adaptively setting the gaps between adjacent cell levels, during writing and rewriting?
Increasing and decreasing cell levels have different cost.

The change of cell levels during programming is a random process.

Thousands of cells are programmed in parallel sharing the same programming voltage. The worst behavior of cells determines performance.

Cell levels drift toward the amorphous state after programming, which is a serious problem.

How to program cells reliably, fast, and adaptively?
Definition of Rank Modulation [1-2]

Rank Modulation:

We use the relative order of cell levels (instead of their absolute values) to represent data.


**Examples and Extensions of Rank Modulation**

- **Example:** Use 2 cells to store 1 bit.

  Relative order: (1,2)
  Value of data: 0
  
  ![Cell 1](image1)
  ![Cell 2](image2)

  Relative order: (2,1)
  Value of data: 1
  
  ![Cell 1](image3)
  ![Cell 2](image4)

- Extensions:
  1. We can partition cells into groups, and apply rank modulation to each group.
  2. We can let each rank contain multiple cells, namely, to extend permutations to multi-set permutations.
Examples and Extensions of Rank Modulation

- Example: Use 2 cells to store 1 bit.

  ![Diagram showing two cells with relative order and value of data]

- Example: Use 3 cells to store $\log_2 6$ bits. The relative orders (1, 2, 3), (1, 3, 2), \ldots, (3, 2, 1) are mapped to data 0, 1, \ldots, 5.
Examples and Extensions of Rank Modulation

- Example: Use 2 cells to store 1 bit.

- Example: Use 3 cells to store \( \log_2 6 \) bits. The relative orders (1, 2, 3), (1, 3, 2), \( \cdots \), (3, 2, 1) are mapped to data 0, 1, \( \cdots \), 5.

- In general, \( k \) cells can represent \( \log_2(k!) \) bits.
Examples and Extensions of Rank Modulation

- Example: Use 2 cells to store 1 bit.

  - Relative order: (1,2)
  - Value of data: 0
  - Relative order: (2,1)
  - Value of data: 1

- Example: Use 3 cells to store \( \log_2 6 \) bits. The relative orders \((1, 2, 3), (1, 3, 2), \ldots, (3, 2, 1)\) are mapped to data \(0, 1, \ldots, 5\).

- In general, \(k\) cells can represent \(\log_2 (k!)\) bits.

- Extensions: (1) We can partition cells into groups, and apply rank modulation to each group. (2) We can let each rank contain multiple cells, namely, to extend permutations to multi-set permutations.
Advantages of Rank Modulation

- Reliable and fast cell programming: No risk of overshooting.
- More tolerance of asymmetric noise and cell-level drifting.
- Ability to remove errors and balance cell levels by physically adjusting cell levels: Only the relative order of the cell levels matters, so we can adaptively increase cell levels.

In the following, we consider “Codes for Rewriting” and “Error-Correcting Codes”, in the framework of flash memories. (But many of the concepts can be applied to phase-change memories, too.)
Rewriting Codes for Rank Modulation

**Issues**
- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

**Techniques**
- Error correction codes
- Codes for rewriting data
- Rank modulation (with rewriting)
- Constrained coding
Rewriting Codes for Rank Modulation

Issues
- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

Techniques
- Error correction codes
- Codes for rewriting data
- Rank modulation (with rewriting)
- Constrained coding
Rewriting Codes for Rank Modulation

We can change the relative order of cell levels by only increasing cell levels, and therefore rewrite data.

Example: Use 3 cells to store data of 6 possible values: 0, 1, ···, 5. When the data change as 0 → 1 → 2 → 5 → 3 → ···, the cell state changes as:

Cell state: (1,2,3) → (1,3,2) → (2,1,3) → (3,2,1) → (2,3,1)
Data: 0 → 1 → 2 → 5 → 3
**Prefix Codes with Push-to-Top Operation**

**Push-to-Top operation:** Each time, we push one cell’s level to be higher than all the other cells’ levels. (It is easy to implement.)

Example: Say we need to change the permutation from \((1, 2, 3, 4)\) to \((4, 3, 1, 2)\).

![Diagram illustrating the push-to-top operation]

**Cost of Rewriting:** The number of cells that are pushed. (It represents by how much the highest cell level is increased.)
Prefix Codes with Push-to-Top Operation

**Prefix Code:** We use the prefixes of the permutations to represent data. Those prefixes cannot be prefixes of each other.
Prefix Code: We use the prefixes of the permutations to represent data. Those prefixes cannot be prefixes of each other.

Intuitive reason: When we push cells to the top, we are changing the prefix of the permutation.
Prefix Code: We use the prefixes of the permutations to represent data. Those prefixes cannot be prefixes of each other.

Intuitive reason: When we push cells to the top, we are changing the prefix of the permutation.

Example: We use 3 cells to represent data of 6 values. Use prefixes of length 2: (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1). The maximum rewriting cost is 2.

Example: Use 3 cells to represent data of 3 values. Use prefixes of length 1: (1,∗,∗), (2,∗,∗), (3,∗,∗). Maximum rewriting cost is 1.

There is a tradeoff between code rate and rewriting cost.
Prefix Codes with Push-to-Top Operation

**Prefix Code:** We use the prefixes of the permutations to represent data. Those prefixes cannot be prefixes of each other.

Intuitive reason: When we push cells to the top, we are changing the prefix of the permutation.

Example: We use 3 cells to represent data of 6 values. Use prefixes of length 2: (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1). The maximum rewriting cost is 2.

Example: Use 3 cells to represent data of 3 values. Use prefixes of length 1: (1, *, *), (2, *, *), (3, *, *). Maximum rewriting cost is 1.
Prefix Code: We use the prefixes of the permutations to represent data. Those prefixes cannot be prefixes of each other.

Intuitive reason: When we push cells to the top, we are changing the prefix of the permutation.

Example: We use 3 cells to represent data of 6 values. Use prefixes of length 2: (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1). The maximum rewriting cost is 2.

Example: Use 3 cells to represent data of 3 values. Use prefixes of length 1: (1,*,*), (2,*,*), (3,*,*). Maximum rewriting cost is 1.

There is a tradeoff between code rate and rewriting cost.
State Diagram for Permutations

*State Diagram:* Vertices are permutations. There is a directed edge \((u, v)\) from \(u\) to \(v\) if the cost of changing \(u\) to \(v\) is 1, with the push-to-top operation.
State Diagram for Permutations

*State Diagram*: Vertices are permutations. There is a directed edge \((u, v)\) from \(u\) to \(v\) if the cost of changing \(u\) to \(v\) is 1, with the push-to-top operation.

In general, the cost of changing \(u\) to \(v\) equals the length of the shortest path from \(u\) to \(v\) in the state graph.
State Diagram for Permutations

*State Diagram:* Vertices are permutations. There is a directed edge \((u, v)\) from \(u\) to \(v\) if the cost of changing \(u\) to \(v\) is 1, with the push-to-top operation.

In general, the cost of changing \(u\) to \(v\) equals the length of the shortest path from \(u\) to \(v\) in the state graph.

Example: 3 cells. The state diagram is:

```
(1,2,3) -> (3,1,2) -> (2,3,1)
(2,1,3) -> (1,3,2) -> (3,2,1)
```

(1,2,3) -> (3,1,2) -> (2,3,1)
(2,1,3) -> (1,3,2) -> (3,2,1)
Mapping Permutations to Data Values

By mapping permutations to data values appropriately (with dominating set type of partitions in the state diagram), we can minimize the maximum rewriting cost.
Mapping Permutations to Data Values

By mapping permutations to data values appropriately (with dominating set type of partitions in the state diagram), we can minimize the maximum rewriting cost.

Example: Use 3 cells to store data of 3 values. With the following mapping, the maximum rewriting cost is 1.

![Diagram](image-url)
Basic Concepts

\( n \): The number of cells.

\( S_n \): The set of \( n! \) permutations.

\( d(u, v) \): The distance from \( u \in S_n \) to \( v \in S_n \) in the state graph.

We define the ball \( B_r(u) \) as \( B_r(u) = \{ v \in S_n \mid d(u, v) \leq r \} \).

\( V \): The set of values of the stored data.

The rate of the code is defined as \( \frac{\log_2 |V|}{\log_2(n!)} \).

Since for any \( u, v \in S_n \), \( |B_r(u)| = |B_r(v)| \), let \( |B_r| \triangleq |B_r(u)| \).

Fact 1: \( |B_r| = \frac{n!}{(n-r)!} \), which equals the number of prefixes of length \( r \).

Fact 2: If maximum rewrite cost = \( r \), code rate \( \leq \frac{\log_2 |B_r|}{\log_2(n!)} \).
Given $n$ and $|V|$, how to build a rewriting code that minimizes the maximum (i.e., worst-case) rewrite cost?

Let $\rho$ be the smallest integer such that $|B_\rho| \geq |V|$.

Fact: The maximum rewrite cost is at least $\rho$. 
Given $n$ and $|V|$, how to build a rewriting code that minimizes the maximum (i.e., worst-case) rewrite cost?

Let $\rho$ be the smallest integer such that $|B_{\rho}| \geq |V|$. 

Fact: The maximum rewrite cost is at least $\rho$.

**Construction (Optimal Prefix Code)**

Choose $|V|$ prefixes (of permutations) of length $\rho$. Map permutations of the same prefix to the same data value.
Example: Say we are given 4 cells to store $\log_2 10$ bits. Since there are only 4 prefixes of length 1 but $12 \geq 10$ prefixes of length 2, we should choose prefixes of length 2, and the maximum rewrite cost is 2.

We can map prefixes to data values as follows:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>(1,2)</th>
<th>(1,3)</th>
<th>(1,4)</th>
<th>(2,1)</th>
<th>(2,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prefix</th>
<th>(2,4)</th>
<th>(3,1)</th>
<th>(3,2)</th>
<th>(3,4)</th>
<th>(4,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

All permutations with the same prefix represent the same data. For example, (1, 2, 3, 4) and (1, 2, 4, 3) represent the same data 0.
Example (continued): Now consider rewriting. Say the current cell state (permutation) is (1, 2, 3, 4), which represents data 0. We want to change the data to 7. How to do it?
Example (continued): Now consider rewriting. Say the current cell state (permutation) is (1, 2, 3, 4), which represents data 0. We want to change the data to 7. How to do it?

Solution: Choose a permutation such that: (1) Its prefix represents 7; (2) The remaining numbers have the same order as in the original permutation (1, 2, 3, 4). So we choose (3, 2, 1, 4) as the new permutation. Then the process of cell programming with push-to-top operations is as follows:

\[
(1,2,3,4) \xrightarrow{} (2,1,3,4) \xrightarrow{} (3,2,1,4)
\]
**Model:** The sequence of rewritten data are i.i.d., with a known distribution. We want to minimize the expected cost of rewriting.
Rewriting Codes for Average-case Performance

**Model:** The sequence of rewritten data are i.i.d., with a known distribution. We want to minimize the expected cost of rewriting. Let $n$ denote the number of cells, and $|V|$ denote the number of data values. We will present a prefix code with the following performance:

**Construction (Prefix Code for Expected Rewriting Cost)**

We present a prefix code with this performance: For every rewrite (not just for a sequence of rewrites), when $|V| \leq \frac{n!}{2}$, its expected rewrite cost is at most 3 times the optimal expected cost; when $n \geq 4$ and $|V| \leq \frac{n!}{6}$, it is at most 2 times the optimal.
Observation: The $n!$ permutations can be represented by a tree. Their prefix-free prefixes can be represented by the leaves of a subtree.

Example: The tree for permutations of length $n = 4$ is:

And the following subtree represents a prefix code of 9 codewords:
Rewriting Codes for Average-case Performance [1]

Construction (Prefix Code for Expected Rewriting Cost)

*Build a prefix code that minimizes the expected length of the codewords (i.e., prefixes). The code construction has time complexity $O(n|V|^4)$, and achieves approximation ratios as introduced previously (compared to any code, not just prefix code).*
Rewriting Codes for Average-case Performance [1]

Construction (Prefix Code for Expected Rewriting Cost)

Build a prefix code that minimizes the expected length of the codewords (i.e., prefixes). The code construction has time complexity \( O(n |V|^4) \), and achieves approximation ratios as introduced previously (compared to any code, not just prefix code).

Example: Let \( n = 4, |V| = 9 \). Let the probabilities and a code be as follows. Then the expected codeword length is:

\[
p_0 + p_1 + 2p_2 + 2p_3 + 2p_4 + 2p_5 + 2p_6 + 3p_7 + 3p_8.
\]

<table>
<thead>
<tr>
<th>Data</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Probability</td>
<td>( p_0 )</td>
<td>( p_1 )</td>
<td>( p_2 )</td>
<td>( p_3 )</td>
<td>( p_4 )</td>
<td>( p_5 )</td>
<td>( p_6 )</td>
<td>( p_7 )</td>
<td>( p_8 )</td>
</tr>
<tr>
<td>Codewords</td>
<td>(1)</td>
<td>(2)</td>
<td>(3, 1)</td>
<td>(3, 2)</td>
<td>(3, 4)</td>
<td>(4, 1)</td>
<td>(4, 2)</td>
<td>(4, 3, 1)</td>
<td>(4, 3, 2)</td>
</tr>
</tbody>
</table>

Rewriting Codes for Average-case Performance

Idea of Proof: Consider a rewrite. Ideally, which code will minimize the expected rewrite cost for this rewrite?

Assume the data before rewriting is \( i \in \{0, 1, \cdots, |V| - 1\} \). Without loss of generality, assume

\[
p_0 \geq p_1 \geq \cdots \geq p_{|V| - 1}.
\]

Then the following “ideal code” minimizes the expected rewrite cost for this rewrite:

- Let \( u \in S_n \) denote the permutation before the rewrite. Sort the \( n! \) permutations as \( v_0, v_1, \cdots, v_{n! - 1} \) such that the rewrite cost

\[
d(u, v_0) \leq d(u, v_1) \leq \cdots \leq d(u, v_{n! - 1}).
\]

Map \( v_0, v_1, \cdots, v_{|V| - 1} \) to data \( i, 0, 1, \cdots, i - 1, i + 1, \cdots, |V| - 1 \), respectively.
Idea of Proof (continued): Consider the case $|V| \leq \frac{n!}{2}$. By the Kraft-McMilan Inequality, there exists a subtree of the tree for permutations, where the $|V|$ data values are mapped to its leaves, such that:

- The codeword for data $i$ has length 1.
- For every other data value $i \in \{0, 1, \cdots, |V| - 1\} - \{i\}$, its codeword’s length is at most 3 times of its corresponding “ideal codeword length”.

This subtree is a “specific code” for this rewrite.
Rewriting Codes for Average-case Performance

Idea of Proof (continued): We have:

- Expected rewrite cost of the constructed prefix code
  \[ \leq \text{Expected codeword length of the constructed prefix code, but excluding data } i \]
  \[ \leq \text{Expected codeword length of the “specific prefix code” in the previous slide, but excluding data } i \]
  \[ \leq 3 \times \text{Expected rewrite cost of the “ideal code.”} \]

- When \( n \geq 4 \) and \( |V| \leq \frac{n!}{6} \), similar proof applies.
Rewriting Codes with Minimal-Push-Up Operation [1]

Push-to-top is simple, but not necessary optimal (in minimizing rewrite cost).

**Minimal-Push-Up operation**: Each time, we push one cell’s level to be higher than those cells that should be below it in the final permutation (instead of higher than all the other cells).

Note: For both push-to-top and minimal-push-up, cells are pushed in the same order. The difference is by how much cell levels are pushed.

Example: We want to change the permutation from \((2, 1, 3, 4)\) to \((2, 1, 4, 3)\).

**Push-to-Top:**

\[(2, 1, 3, 4) \rightarrow (4, 2, 1, 3) \rightarrow (1, 4, 2, 3) \rightarrow (2, 1, 4, 3)\]

**Minimal-Push-Up:**

\[(2, 1, 3, 4) \rightarrow (2, 1, 4, 3) \rightarrow (2, 1, 4, 3)\]
Rewriting Codes with Minimal-Push-Up Operation

To measure rewrite cost, we use discrete “virtual levels.”

Example: Changing \((2, 1, 3, 4)\) to \((2, 1, 4, 3)\).

Since highest virtual level increases by 1, rewrite cost = 1.
Rewriting Codes with Minimal-Push-Up Operation

Definition (Ball of Radius $r$)
Given $u \in S_n$ and non-negative integer $r$, let $B_r(u)$ be the set of permutations such that changing $u$ to them has rewrite cost at most $r$.

- With push-to-top operations, $|B_r(u)| = \frac{n!}{(n-r)!}$.  
  Example: When $r = 1$, $|B_1(u)| = n$.
- With minimal-push-up, $|B_r(u)| = r!(r + 1)^{n-r}$.  
  Example: When $r = 1$, $|B_1(u)| = 2^{n-1}$.

So minimal-push-up can improve performance significantly.
Example of Specific Rewriting Code with Minimal-Push-Up Operation [1]

Given Parameters: Use $n = 4$ cells to store data of $|V| = 6$ values, and maximum rewrite cost $= 1$.

In the rewrite code, the mapping from permutations to data is:

<table>
<thead>
<tr>
<th>Permutations</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2,3,4)</td>
<td>0</td>
</tr>
<tr>
<td>(1,2,4,3)</td>
<td>1</td>
</tr>
<tr>
<td>(1,3,2,4)</td>
<td>2</td>
</tr>
<tr>
<td>(1,3,4,2)</td>
<td>3</td>
</tr>
<tr>
<td>(1,4,2,3)</td>
<td>4</td>
</tr>
<tr>
<td>(1,4,3,2)</td>
<td>5</td>
</tr>
<tr>
<td>(2,3,4,1)</td>
<td></td>
</tr>
<tr>
<td>(2,4,3,1)</td>
<td></td>
</tr>
<tr>
<td>(3,2,4,1)</td>
<td></td>
</tr>
<tr>
<td>(3,4,2,1)</td>
<td></td>
</tr>
<tr>
<td>(4,2,3,1)</td>
<td></td>
</tr>
<tr>
<td>(4,3,2,1)</td>
<td></td>
</tr>
<tr>
<td>(3,4,1,2)</td>
<td></td>
</tr>
<tr>
<td>(4,3,1,2)</td>
<td></td>
</tr>
<tr>
<td>(2,4,1,3)</td>
<td></td>
</tr>
<tr>
<td>(4,2,1,3)</td>
<td></td>
</tr>
<tr>
<td>(2,1,3,4)</td>
<td></td>
</tr>
<tr>
<td>(3,1,4,2)</td>
<td></td>
</tr>
<tr>
<td>(2,1,4,3)</td>
<td></td>
</tr>
</tbody>
</table>

Proof of correctness: Each coset of permutations is a dominating set.
Example of Specific Rewriting Code with Minimal-Push-Up Operation [1]

Given Parameters: Use $n = 4$ cells to store data of $|V| = 6$ values, and maximum rewrite cost = $1$.

In the rewrite code, the mapping from permutations to data is:

<table>
<thead>
<tr>
<th>Permutations</th>
<th>(1,2,3,4)</th>
<th>(1,2,4,3)</th>
<th>(1,3,2,4)</th>
<th>(1,3,4,2)</th>
<th>(1,4,2,3)</th>
<th>(1,4,3,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(2,3,4,1)</td>
<td>(2,4,3,1)</td>
<td>(2,4,1,3)</td>
<td>(2,4,3,1)</td>
<td>(2,3,1,4)</td>
<td>(2,1,4,3)</td>
</tr>
<tr>
<td></td>
<td>(3,4,1,2)</td>
<td>(3,4,1,2)</td>
<td>(3,2,4,1)</td>
<td>(3,4,2,1)</td>
<td>(2,3,1,4)</td>
<td>(3,2,1,4)</td>
</tr>
<tr>
<td></td>
<td>(4,1,2,3)</td>
<td>(3,1,2,4)</td>
<td>(4,1,3,2)</td>
<td>(4,1,3,2)</td>
<td>(2,1,3,4)</td>
<td>(2,1,4,3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

Proof of correctness: Each coset of permutations is a dominating set.

In comparison, with push-to-top operations, when $n = 4$ and $r = 1$, the stored data can have at most 4 values. So minimal-push-up can achieve higher rate. But currently, relatively less is known about its codes than push-to-top coding.

Further reading (1/3)

Rank Modulation and Its Rewriting Codes


Further reading (3/3)


Error-Correcting Codes for Rank Modulation

Issues
- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

Techniques
- Error correction codes
- Codes for rewriting data
- Rank modulation (with error correction)
- Constrained coding
Error-Correcting Codes for Rank Modulation

**Issues**
- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

**Techniques**
- Error correction codes
- Codes for rewriting data
- Rank modulation (with error correction)
- Constrained coding
Based on the error model, there are various reasonable choices for the distance between permutations:

- Kendall-tau distance. (To be introduced in detail.)
- $L_\infty$ distance.
- Gaussian noise based distance.
- Distance defined based on asymmetric errors or inter-cell interference.

We should choose the distance appropriately based on the type and magnitude of errors.
Kendall-tau Distance for Rank Modulation ECC [1]

When errors happen, the smallest change in a permutation is the local exchange of two adjacent numbers in the permutation. That is,

\[(a_1, \cdots, a_{i-1}, a_i, a_{i+1}, a_{i+2}, \cdots, a_n) \rightarrow (a_1, \cdots, a_{i-1}, a_{i+1}, a_i, a_{i+2}, \cdots, a_n)\]
Kendall-tau Distance for Rank Modulation ECC [1]

When errors happen, the smallest change in a permutation is the local exchange of two adjacent numbers in the permutation. That is,

\[(a_1, \cdots, a_{i-1}, a_i, a_{i+1}, a_{i+2}, \cdots, a_n) \rightarrow (a_1, \cdots, a_{i-1}, a_{i+1}, a_i, a_{i+2}, \cdots, a_n)\]

Example:

Original Cell Levels

Noisy Cell Levels

\[(2,1,5,3,4) \rightarrow (2,1,3,5,4)\]
Kendall-tau Distance for Rank Modulation ECC [1]

When errors happen, the smallest change in a permutation is the local exchange of two adjacent numbers in the permutation. That is,

\[(a_1, \cdots, a_{i-1}, a_i, a_{i+1}, a_{i+2}, \cdots, a_n) \rightarrow (a_1, \cdots, a_{i-1}, a_{i+1}, a_i, a_{i+2}, \cdots, a_n)\]

Example:

Original Cell Levels

\[(2,1,5,3,4)\]

Noisy Cell Levels

\[(2,1,3,5,4)\]

We can extend the concept to multiple such “local exchanges” (for larger errors).

Kendall-tau Distance for Rank Modulation ECC

Definition (Adjacent Transposition)

An adjacent transposition is the local exchange of two neighboring numbers in a permutation, namely,

\[(a_1, \cdots, a_{i-1}, a_i, a_{i+1}, a_{i+2}, \cdots, a_n) \rightarrow (a_1, \cdots, a_{i-1}, a_{i+1}, a_i, a_{i+2}, \cdots, a_n)\]
Kendall-tau Distance for Rank Modulation ECC

**Definition (Adjacent Transposition)**

An adjacent transposition is the local exchange of two neighboring numbers in a permutation, namely,

\[(a_1, \cdots, a_{i-1}, a_i, a_{i+1}, a_{i+2}, \cdots, a_n) \rightarrow (a_1, \cdots, a_{i-1}, a_{i+1}, a_i, a_{i+2}, \cdots, a_n)\]

**Definition (Kendall-tau Distance)**

Given two permutations \(A\) and \(B\), the Kendall-tau distance between them, \(d_\tau(A, B)\), is the minimum number of adjacent transpositions needed to change \(A\) into \(B\). (Note that \(d_\tau(A, B) = d_\tau(B, A)\).)
Kendall-tau Distance for Rank Modulation ECC

Definition (Adjacent Transposition)

An adjacent transposition is the local exchange of two neighboring numbers in a permutation, namely,

\[(a_1, \cdots, a_{i-1}, a_i, a_{i+1}, a_{i+2}, \cdots, a_n) \rightarrow (a_1, \cdots, a_{i-1}, a_{i+1}, a_i, a_{i+2}, \cdots, a_n)\]

Definition (Kendall-tau Distance)

Given two permutations \(A\) and \(B\), the Kendall-tau distance between them, \(d_\tau(A, B)\), is the minimum number of adjacent transpositions needed to change \(A\) into \(B\). (Note that \(d_\tau(A, B) = d_\tau(B, A)\).)

If the minimum Kendall-tau distance of a code is \(2t+1\), then it can correct \(t\) adjacent transposition errors.
Kendall-tau Distance for Rank Modulation ECC

*Example:* Let $A = (2, 1, 3, 4)$ and $B = (2, 3, 4, 1)$. Then $d_\tau(A, B) = 2$, and the transition from $A$ to $B$ is

$$A = (2, 1, 3, 4) \rightarrow (2, 3, 1, 4) \rightarrow (2, 3, 4, 1) = B.$$
Kendall-tau Distance for Rank Modulation ECC

Example: Let $A = (2, 1, 3, 4)$ and $B = (2, 3, 4, 1)$. Then $d_\tau(A, B) = 2$, and the transition from $A$ to $B$ is

$$A = (2, 1, 3, 4) \rightarrow (2, 3, 1, 4) \rightarrow (2, 3, 4, 1) = B.$$

Fact: For two permutations $A, B \in S_n$, $d_\tau(A, B) \leq \binom{n}{2}$. 
Kendall-tau Distance for Rank Modulation ECC

Example: Let \( A = (2, 1, 3, 4) \) and \( B = (2, 3, 4, 1) \). Then \( d_\tau(A, B) = 2 \), and the transition from \( A \) to \( B \) is
\[
A = (2, 1, 3, 4) \rightarrow (2, 3, 1, 4) \rightarrow (2, 3, 4, 1) = B.
\]

Fact: For two permutations \( A, B \in S_n \), \( d_\tau(A, B) \leq \binom{n}{2} \).

Definition (State Diagram)

Vertices are permutations. There is an undirected edge between two permutations \( A, B \in S_n \) iff \( d_\tau(A, B) = 1 \).

Example: The state diagram for \( n = 3 \) cells is

```
(1,2,3) ←[2] (2,1,3) ←[2] (2,3,1) ←[2] (3,2,1)
(1,3,2) ←[2] (3,1,2)
```
Kendall-tau Distance for Rank Modulation ECC

Example: The state diagram for $n = 4$ cells is
One-Error-Correcting Code

We introduce an error-correcting code of minimum Kendall-tau distance 3, which corrects one Kendall (i.e., adjacent transposition) error.
We introduce an error-correcting code of minimum Kendall-tau distance 3, which corrects one Kendall (i.e., adjacent transposition) error.

**Definition (Inversion Vector)**

Given a permutation \((a_1, a_2, \cdots, a_n)\), its inversion vector \((x_1, x_2, \cdots, x_{n-1}) \in \{0, 1\} \times \{0, 1, 2\} \times \cdots \times \{0, 1, \cdots, n-1\}\) is determined as follows:

- For \(i = 1, 2, \cdots, n-1\), \(x_i\) is the number of elements in \(\{1, 2, \cdots, i\}\) that are behind \(i + 1\) in the permutation \((a_1, \cdots, a_n)\).
One-Error-Correcting Code

We introduce an error-correcting code of minimum Kendall-tau distance 3, which corrects one Kendall (i.e., adjacent transposition) error.

**Definition (Inversion Vector)**

Given a permutation \((a_1, a_2, \cdots, a_n)\), its inversion vector \((x_1, x_2, \cdots, x_{n-1}) \in \{0, 1\} \times \{0, 1, 2\} \times \cdots \times \{0, 1, \cdots, n-1\}\) is determined as follows:

- For \(i = 1, 2, \cdots, n-1\), \(x_i\) is the number of elements in \(\{1, 2, \cdots, i\}\) that are behind \(i+1\) in the permutation \((a_1, \cdots, a_n)\).

**Example:** The inversion vector for \((1, 2, 3, 4)\) is \((0, 0, 0)\). The inversion for \((4, 3, 2, 1)\) is \((1, 2, 3)\). The inversion vector for \((2, 4, 3, 1)\) is \((1, 1, 2)\).
By viewing the inversion vector as coordinates, we embed permutations in an \((n - 1)\)-dimensional space.
One-Error-Correcting Code [1]

By viewing the inversion vector as coordinates, we embed permutations in an \((n - 1)\)-dimensional space.

Fact: For any two permutations \(A, B \in S_n\), \(d_\tau(A, B)\) is no less than their \(L_1\) distance in the \((n - 1)\)-dimensional space.
By viewing the inversion vector as coordinates, we embed permutations in an \((n - 1)\)-dimensional space.

Fact: For any two permutations \(A, B \in S_n\), \(d_\tau(A, B)\) is no less than their \(L_1\) distance in the \((n - 1)\)-dimensional space.

Idea: We can construct a code of minimum \(L_1\) distance \(D\) in the \((n - 1)\)-dimensional array of size \(2 \times 3 \times \cdots \times n\). Then it is a code of Kendall-tau distance at least \(D\) for the permutations.

One-Error-Correcting Code

Example: When $n = 3$ or $n = 4$, the embedding is as follows. (Only the solid edges are the edges in the state graph of permutations.)

<table>
<thead>
<tr>
<th>Permutation</th>
<th>Coordinates</th>
<th>Permutation</th>
<th>Coordinates</th>
<th>Permutation</th>
<th>Coordinates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
<td>(0,0)</td>
<td>1 2 3 4</td>
<td>(0,0,0)</td>
<td>3 1 2 4</td>
<td>(0,2,0)</td>
</tr>
<tr>
<td>1 3 2</td>
<td>(0,1)</td>
<td>1 2 4 3</td>
<td>(0,0,1)</td>
<td>3 1 4 2</td>
<td>(0,2,1)</td>
</tr>
<tr>
<td>2 1 3</td>
<td>(1,0)</td>
<td>1 3 2 4</td>
<td>(0,1,0)</td>
<td>3 2 1 4</td>
<td>(1,2,0)</td>
</tr>
<tr>
<td>2 3 1</td>
<td>(1,1)</td>
<td>1 3 4 2</td>
<td>(0,1,1)</td>
<td>3 2 4 1</td>
<td>(1,2,1)</td>
</tr>
<tr>
<td>3 1 2</td>
<td>(0,2)</td>
<td>1 4 2 3</td>
<td>(0,0,2)</td>
<td>3 4 1 2</td>
<td>(0,2,2)</td>
</tr>
<tr>
<td>3 2 1</td>
<td>(1,2)</td>
<td>1 4 3 2</td>
<td>(0,1,2)</td>
<td>3 4 2 1</td>
<td>(1,2,2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 1 3 4</td>
<td>(1,0,0)</td>
<td>4 1 2 3</td>
<td>(0,0,3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 1 4 3</td>
<td>(1,0,1)</td>
<td>4 1 3 2</td>
<td>(0,1,3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 3 1 4</td>
<td>(1,1,0)</td>
<td>4 2 1 3</td>
<td>(1,0,3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 3 4 1</td>
<td>(1,1,1)</td>
<td>4 2 3 1</td>
<td>(1,1,3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 4 1 3</td>
<td>(1,0,2)</td>
<td>4 3 1 2</td>
<td>(0,2,3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 4 3 1</td>
<td>(1,1,2)</td>
<td>4 3 2 1</td>
<td>(1,2,3)</td>
</tr>
</tbody>
</table>

(a) ![Diagram of Permutations](image-a)
(b) ![Diagram of Permutations](image-b)
(c) ![Diagram of Permutations](image-c)
(d) ![Diagram of Permutations](image-d)
One-Error-Correcting Code

Construction (One-Error-Correcting Rank Modulation Code)

Let $C_1, C_2 \subseteq S_n$ denote two rank modulation codes constructed as follows. Let $A \in S_n$ be a general permutation whose inversion vector is $(x_1, x_2, \cdots, x_{n-1})$. Then $A$ is a codeword in $C_1$ iff the following equation is satisfied:

$$
\sum_{i=1}^{n-1} ix_i \equiv 0 \pmod{2n-1}
$$

A is a codeword in $C_2$ iff the following equation is satisfied:

$$
\sum_{i=1}^{n-2} ix_i + (n-1) \cdot (-x_{n-1}) \equiv 0 \pmod{2n-1}
$$

Between $C_1$ and $C_2$, choose the code with more codewords as the final output.
One-Error-Correcting Code

For the above code, it can be proved that:

- The code can correct one Kendall error.
- The size of the code is at least \( \frac{(n-1)!}{2} \).
- The size of the code is at least half of optimal.
Codes correcting more Kendall errors are constructed based on embedding.

First, consider codes of the following form:

- Let $m \geq n - 1$ and let $h_1, \cdots, h_{n-1}$ be a set of integers, where $0 < h_i < m$ for $i = 1, \cdots, n - 1$. Define the code as follows:

$$\mathcal{C} = \{(x_1, x_2, \cdots, x_{n-1}) \mid \sum_{i=1}^{n-1} h_i x_i \equiv 0 \mod m\}$$

Fact: The above code can correct $t$ Kendall errors if all the syndromes caused by up to $t$ errors are all distinct.

How to find such integers $h_1, \cdots, h_{n-1}$?
Codes Correcting More Errors

Fact: The above code can correct $t$ Kendall errors if all the syndromes caused by up to $t$ errors are all distinct.

How to find such integers $h_1, \cdots, h_{n-1}$?

**Theorem (Bose-Chowla)**

Let $q$ be a power of a prime, and let $m = \frac{q^{t+1} - 1}{q-1}$. Then there exist $q + 1$ integers $j_0 = 0, j_1, \cdots, j_q$ in $\mathbb{Z}_m$ such that the sums

$$j_{i_1} + j_{i_2} + \cdots + j_{i_t} \quad (0 \leq i_1 \leq i_2 \leq \cdots \leq i_t \leq q)$$

are all different modulo $m$. 
The Bose-Chowla theorem is useful when all the errors in the embedded 
\((n - 1)\)-dimensional \(L_1\) space are positive errors. 
To also handle negative errors, we can “enlarge” the coefficients:

**Theorem**

For \(1 \leq i \leq q + 1\) let

\[
h_i = \begin{cases} 
  j_{i-1} + \frac{t-1}{2} m & \text{for odd } t \\
  j_{i-1} + \frac{t}{2} m & \text{for even } t
\end{cases}
\]

where the numbers \(j_i\) are given by the Bose-Chowla theorem. Let 
\(m_t = t(t + 1)m\) if \(t\) is odd and \(m_t = t(t + 2)m\) if \(t\) is even. For all 
\(e \in \mathbb{Z}^{q+1}\) such that \(\|e\| \leq t\) the sums (i.e., syndromes) \(\sum_{i=1}^{q+1} e_i h_i\) are all 
distinct and nonzero modulo \(m_t\).
More idea: Map each dimension of the \((n - 1)\)-dimensional space to bits using Gray code. Then binary ECC can be turned into ECC for permutations.

Let the number of cells $n \to \infty$. Consider capacity.

Theorem (Capacity of Rank Modulation ECC)

Let $A(n, d)$ be the maximum number of permutations in $S_n$ with minimum Kendall-tau distance $d$. We call

$$C(d) = \lim_{n \to \infty} \frac{\ln A(n, d)}{\ln n!}$$

the capacity of rank modulation ECC of Kendall-tau distance $d$. Then,

$$C(d) = \begin{cases} 1 & \text{if } d = O(n) \\ 1 - \epsilon & \text{if } d = \Theta(n^{1+\epsilon}), \ 0 < \epsilon < 1 \\ 0 & \text{if } d = \Theta(n^2) \end{cases}$$

$L_\infty$ distance between two cell states: The maximum change in the rank of a cell. (We consider the absolute value of the change.)

Example: In the following, cell 2’s rank has changed by 4, which is the greatest.

A practical way to use rank modulation is to partition cells into groups, and apply rank modulation to each cell group. The code, such as ECC, can be defined for all the cell groups together.

Example: Partition cells into groups of size 3.

LDPC code has been studied for this setting, where each cell group is seen as a codeword symbol.

Further reading (1/4)

Error-Correcting Codes for Rank Modulation:


Further reading (2/4)


Further reading (4/4)

Constrained Coding
Issues
- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

Techniques
- Error correction codes
- Codes for rewriting data
- Rank modulation (with rewriting)
- Constrained coding
Issues
- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

Techniques
- Error correction codes
- Codes for rewriting data
- Rank modulation (with rewriting)
- Constrained coding
Example of a NAND flash memory block: A block has \( \sim 64 \) pages. A page has thousands of cells.
Intercell Coupling: The threshold voltage $V_{th}$ is shifted due to parasitic capacitance between neighboring cells, including:

- Horizontal direction (bit-line to bit-line coupling)
- Vertical direction (word-line to word-line coupling)
- Diagonal direction
Intercell Coupling in NAND Flash Memories

- Intercell Coupling: The threshold voltage $V_{th}$ is shifted due to parasitic capacitance between neighboring cells, including:
  - Horizontal direction (bit-line to bit-line coupling)
  - Vertical direction (word-line to word-line coupling)
  - Diagonal direction

- The amount of shift in $V_{th}$ is affected by:
  - Coupling coefficient $C$
  - $V_{th}$ shift of neighboring cell due to programming

Intercell Coupling: The threshold voltage $V_{th}$ is shifted due to parasitic capacitance between neighboring cells, including:
- Horizontal direction (bit-line to bit-line coupling)
- Vertical direction (word-line to word-line coupling)
- Diagonal direction

The amount of shift in $V_{th}$ is affected by:
- Coupling coefficient $C$
- $V_{th}$ shift of neighboring cell due to programming
Intercell Coupling in NAND Flash Memories

- Intercell Coupling: The threshold voltage $V_{th}$ is shifted due to parasitic capacitance between neighboring cells, including:
  - Horizontal direction (bit-line to bit-line coupling)
  - Vertical direction (word-line to word-line coupling)
  - Diagonal direction

- The amount of shift in $V_{th}$ is affected by:
  - Coupling coefficient $C$
  - $V_{th}$ shift of neighboring cell due to programming

- When memory density scales up, intercell coupling becomes more severe.
The $V_{th}$ shift of middle cell caused by shifting of neighboring cells is

$$\Delta V_{i,j} = C_x(\Delta V_{i-1,j} + \Delta V_{i+1,j}) + C_y(\Delta V_{i,j-1} + \Delta V_{i,j+1}) + C_{x,y}(\Delta V_{i-1,j-1} + \Delta V_{i+1,j-1} + \Delta V_{i-1,j+1} + \Delta V_{i+1,j+1})$$
Consider MLC with $q$ levels: $0, 1, \cdots, q - 1$.

- Proposed Constraint [1]: A cell of level 0 cannot be adjacent to a cell of level $q - 1$.
  Example: When $q = 4$, two cells of level 0 and level 3 cannot be adjacent.

Consider MLC with $q$ levels: $0, 1, \cdots, q - 1$.

- Proposed Constraint [1]: A cell of level 0 cannot be adjacent to a cell of level $q - 1$.
  
  Example: When $q = 4$, two cells of level 0 and level 3 cannot be adjacent.

- Proposed Constraint [2]: For every cell, the difference between its own level and the summation of its neighboring cells’ levels cannot be too large.


Further reading (1/2)


- A. Berman and Y. Birk, “Error Correction Scheme for Constrained Inter-Cell Coupling in Flash Memory,” presented at Non-Volatile Memories Workshop (NVMW), San Diego, CA, March 2011.
Further reading (2/2)


Summary and Future Directions

**Issues**
- Delay onset of errors
- Improve reliability
- Improve write access
- Increase storage capacity
- Reduce interference

**Techniques**
- Error correction codes
- Codes for rewriting data
- Rank modulation
- Constrained coding
Open Problems on Coding for NVMs
Open Problems on Coding for NVMs
Open Problems on Coding for NVMs

Codes for Error Correction

Data Representation: MLC, Rank Modulation

Signal Processing

Codes for Rewriting

Codes for Fast Read
Open Problems on Coding for NVMs

- Codes for Error Correction
  - Data Representation: MLC, Rank Modulation
- Codes for Rewriting
- Signal Processing
- Codes for Different NVMs: Flash Memory, PCM, etc.
- Codes for Fast Read
- Data Representation: MLC, Rank Modulation
Open Problems on Coding for NVMs

- Codes for Error Correction
  - In-Memory Source/Channel Coding
  - RAID-like Systems
- Codes for Different NVMs: Flash Memory, PCM, etc.
- Signal Processing
- 3D Memory
- Data Representation: MLC, Rank Modulation
- Short-term and Long-term Memory
- Codes for Computing
- 3D Memory
- Memory Scrubbing
- Codes for Fast Read
- Codes for Rewriting
- Codes for Rewriting
Open Problems on Coding for NVMs

- Codes for Error Correction
- Data Representation: MLC, Rank Modulation
- Codes for Rewriting
- Signal Processing
- Codes for Computing
- In-Memory Source/Channel Coding
- RAID-like Systems
- Codes for Different NVMs: Flash Memory, PCM, etc.
- 3D Memory
- Data Representation: MLC, Rank Modulation
- Short-term and Long-term Memory
- Memory Scrubbing
- Codes for Fast Read
Open Problems on Coding for NVMs
Research Problems

1. Design of graph-based non-binary codes for Flash channels with respect to spatio-temporal variability.
2. Design of LDPC codes in the high-reliability region of quantized Flash channels with respect to absorbing/trapping sets.
3. Error Correction coding methods and information theory for STTRAM.
4. Design of codes combining WOM, ECC and ICI properties.
5. Probabilistic vs. worst case analysis of various coding methods for NVM-inspired channels.
IEEE JSAC Special Issue on Communication Methodologies for the Next-Generation Storage Systems

Topics include:

- Device-level channel modeling for emerging storage technologies
- Information theory and fundamental data transmission limits for new storage channels
- Practical coding and signal processing methods cognizant of underlying physical constraints
- Architecture and design of large-scale storage subsystems based on new non-volatile memories
- Security, data compression and communication techniques for cloud storage and distributed storage networks

Deadline: May 1st, 2013
2013 Non-Volatile Memories Workshop (NVMW)
March 3-5, 2013, University of California, San Diego
The workshop solicits presentations on any topic related to non-volatile, solid state memories, including:

- Advances in memory devices or memory cell design.
- Characterization of commercial or experimental memory devices.
- Error correction and data encoding schemes for non-volatile memories.
- Advances in non-volatile memory-based storage system.
- Operating system and file system designs for non-volatile memories.
- Security and reliability of solid-state storage systems.
- Applications of non-volatile memories to scientific, “big data,” and high-performance workloads.
- Implications of non-volatile memories for applications such as databases and NoSQL systems.

Deadline: November 19, 2012
NVMW 2013

4th Non-Volatile Memories Workshop

March 3-5, 2013

UC San Diego

La Jolla, California USA

http://nvmw.ucsd.edu