FAST'19

Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques

Tianli Zhou & Chao Tian
Texas A&M University

Contents

- Motivation
- Background and Review

- Evaluating Individual Techniques
- Find the Best Strategy under Optimized Bitmatrix
- Proposed Coding Procedure and Evaluation

- Conclusion

Contents

- Motivation
- Background and Review

- Evaluating Individual Techniques
- Find the Best Strategy under Optimized Bitmatrix
- Proposed Coding Procedure and Evaluation

- Conclusion

Data Reliability

Data Reliability

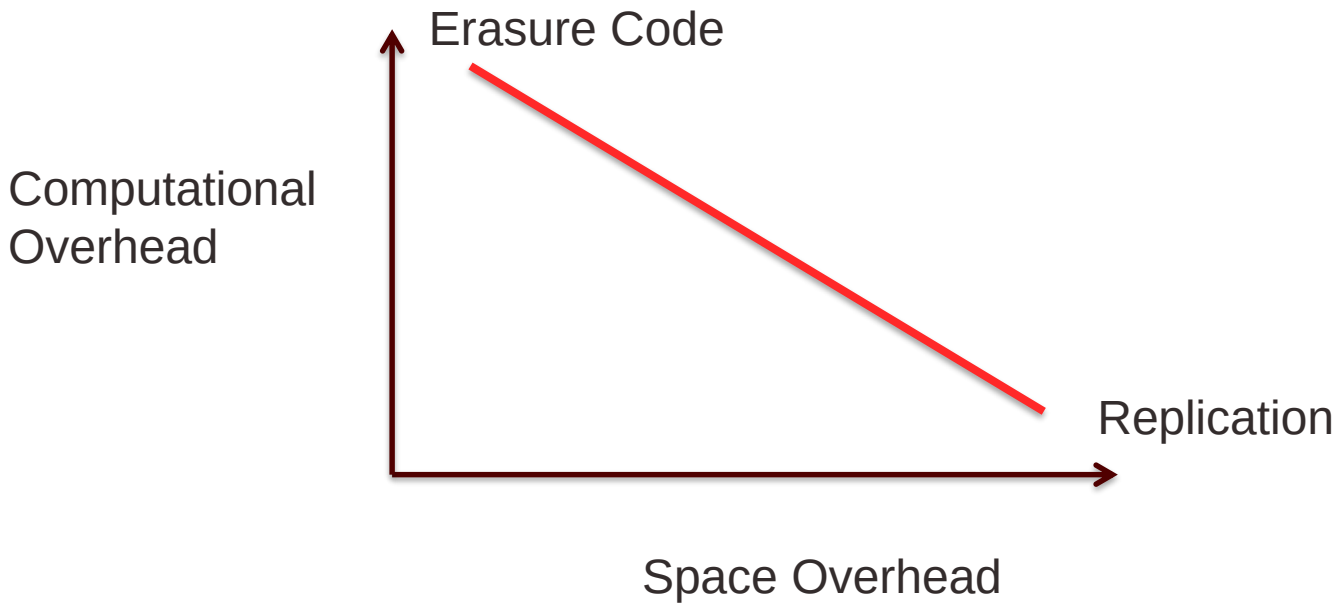
- Disaster happens

Data Reliability

- Disaster happens

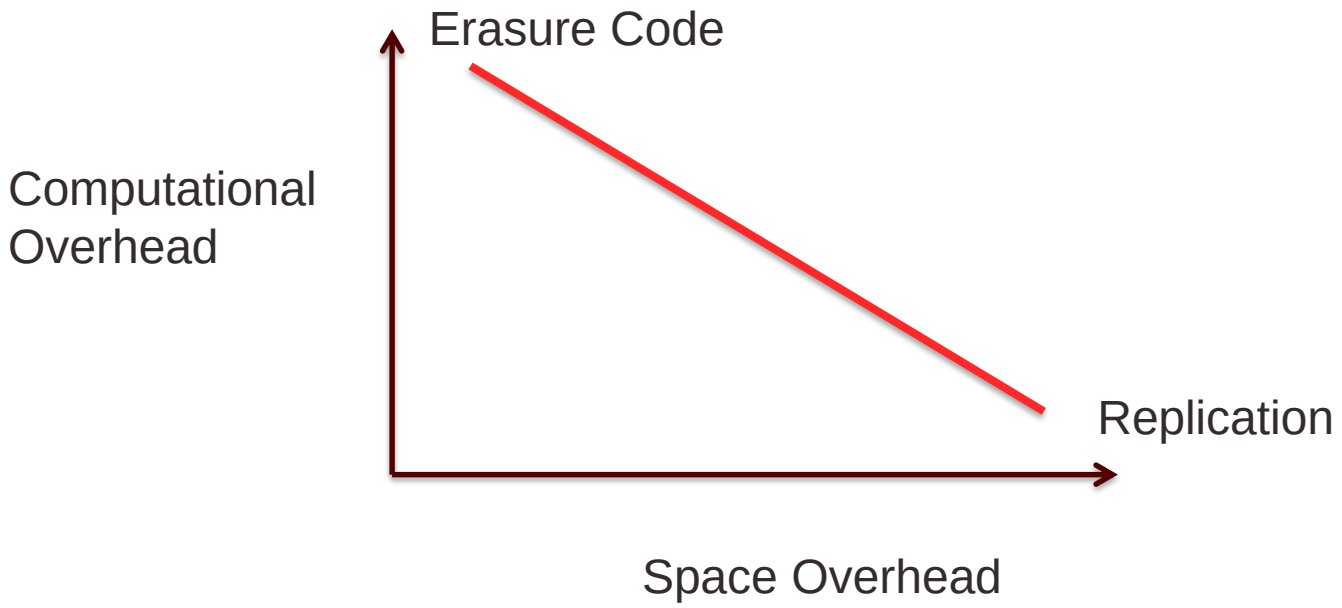
- Disk failure
- Server malfunction
- and more….

Replication - Erasure Code



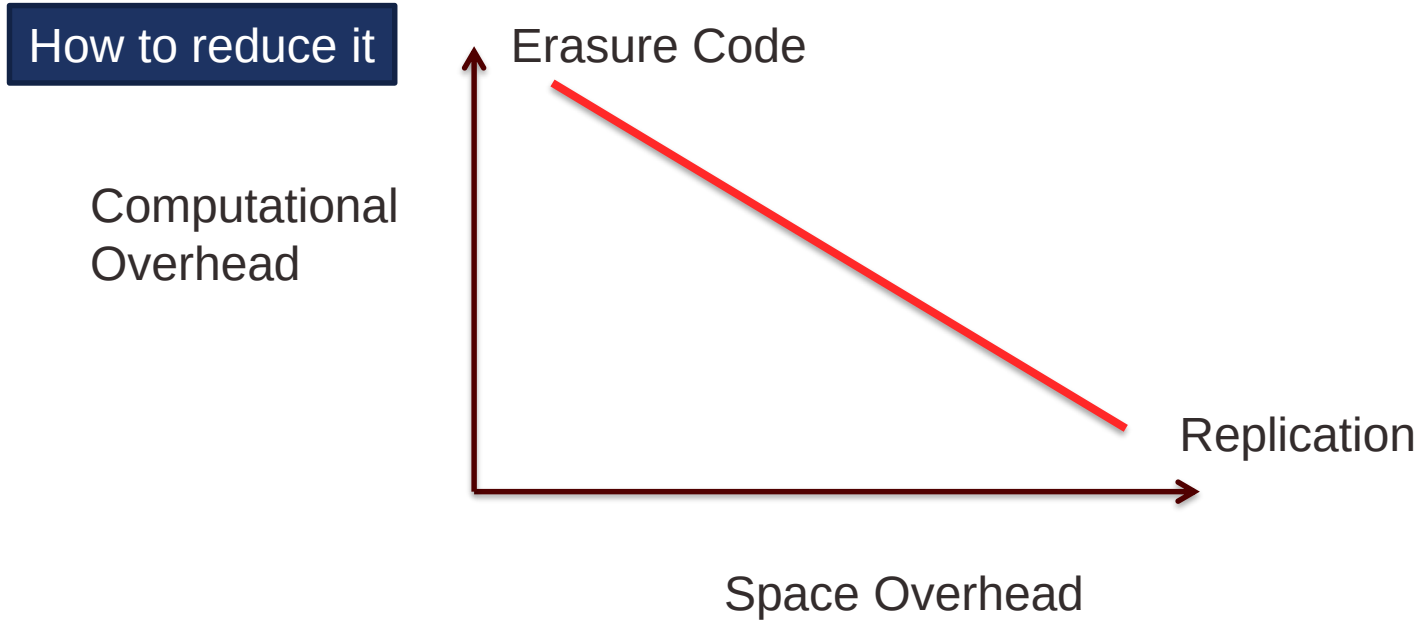
Replication - Erasure Code

- Minimum space overhead
- Much more computational load
- Erasure code relies on Galois Field (finite field) ops



Replication - Erasure Code

- Minimum space overhead
- Much more computational load
- Erasure code relies on Galois Field (finite field) ops



Acceleration Techniques

Optimize Coding Matrix

Schedule Computing Order

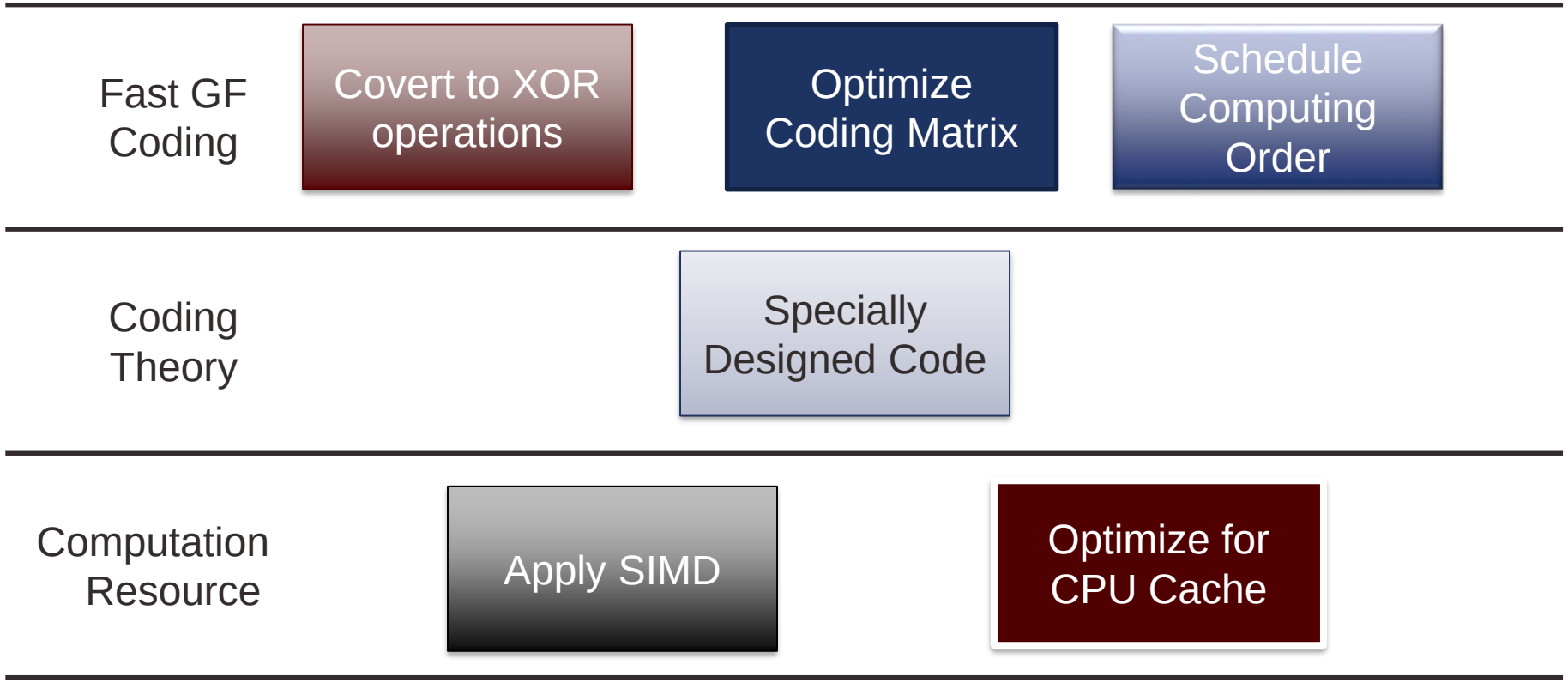
Covert to XOR operations

Apply SIMD

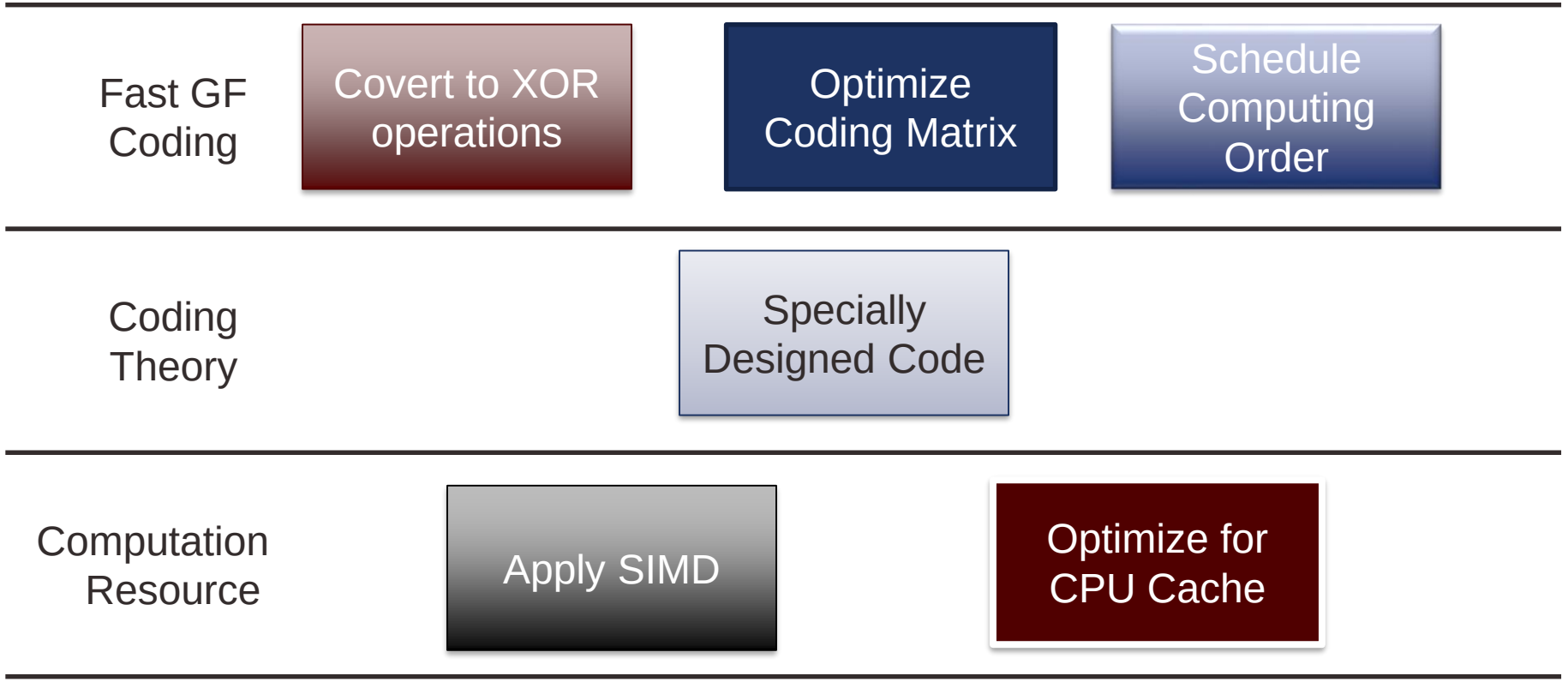
Specially Designed Code

Optimize for CPU Cache

Acceleration Techniques



Acceleration Techniques



?

Question to Answer

- Most effective technique(s)?
- Utilize together?
- Components to be optimized?

- Problem is still important
  - Reduce energy consumption
  - Virtualized environment
  - Mobile / embedded system
  - Future I/O technology

Contents

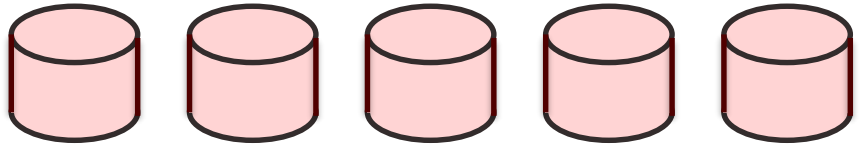
- Motivation
- Background and Review

- Evaluating Individual Techniques
- Find the Best Strategy under Optimized Bitmatrix
- Proposed Coding Procedure and Evaluation

- Conclusion

Erasure Code

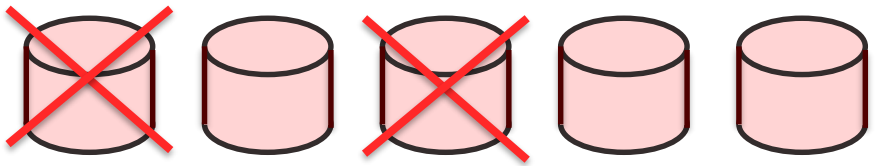
storage nodes



Store coded data to multiple nodes

Erasure Code

storage nodes



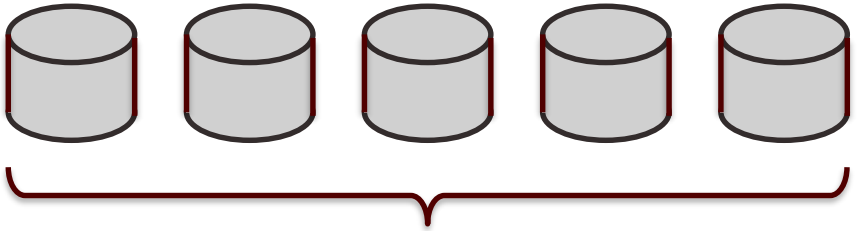
Recover even nodes failures happen

Erasure Code

$k=3$
segments

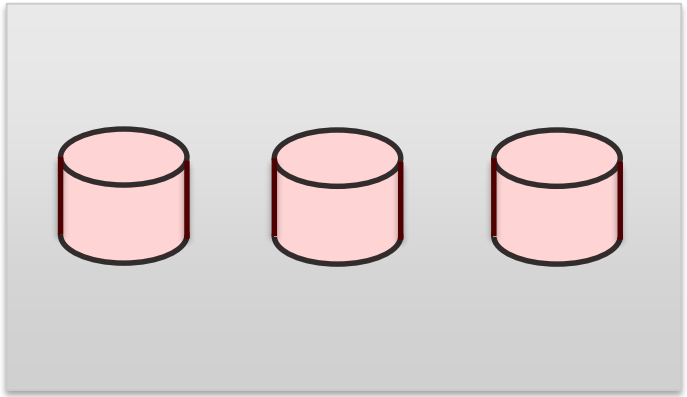
Encode

storage nodes



$n=5$
coded nodes

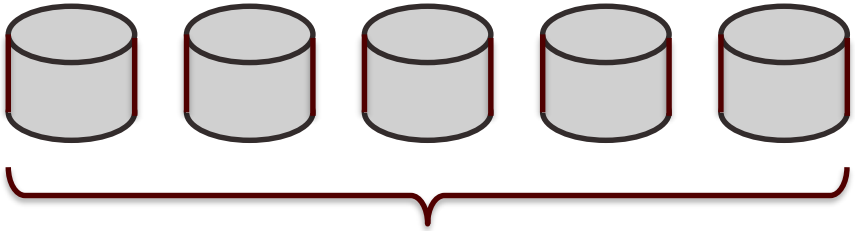
Erasure Code



$k=3$
segments

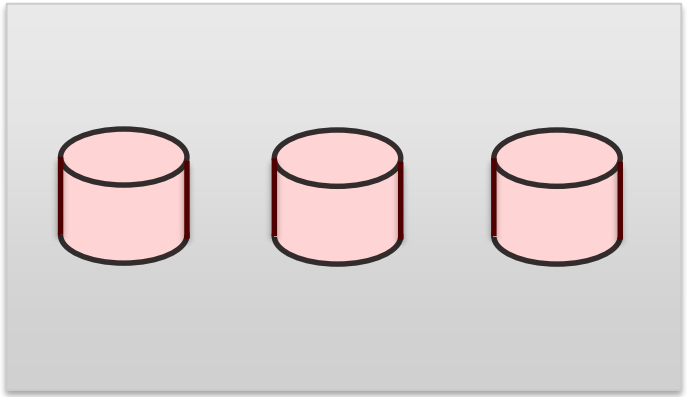
Encode

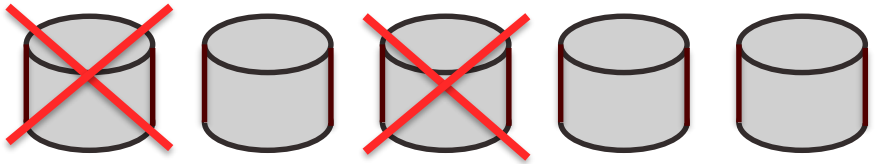
storage nodes



$n=5$
coded nodes

Erasure Code



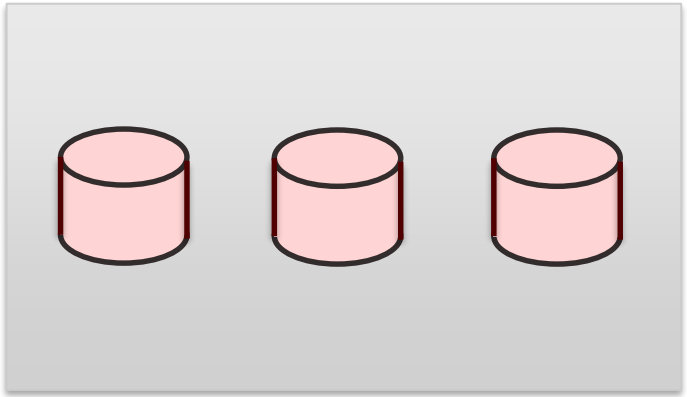


storage nodes

$k=3$
segments

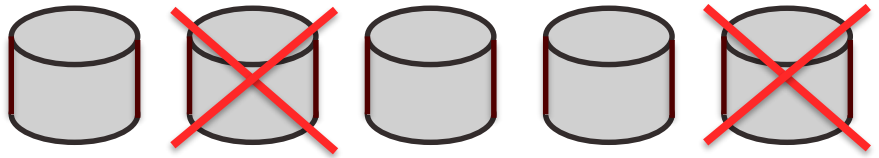
MDS code: recover data from any k surviving nodes

Erasure Code



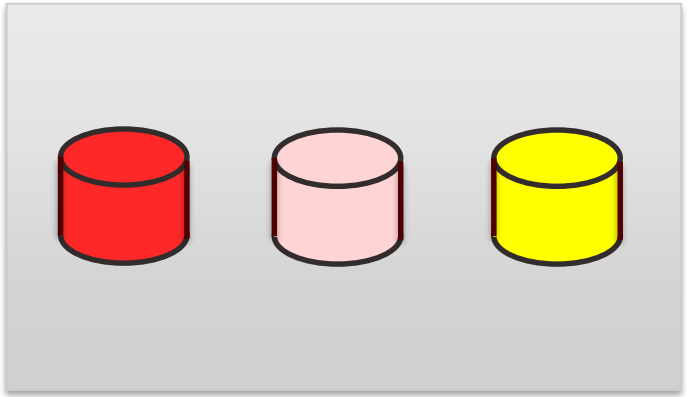
$k=3$
segments

storage nodes

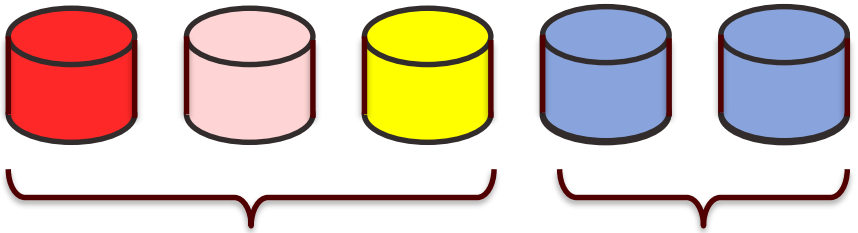


MDS code: recover data from any k surviving nodes

Erasure Code



storage nodes

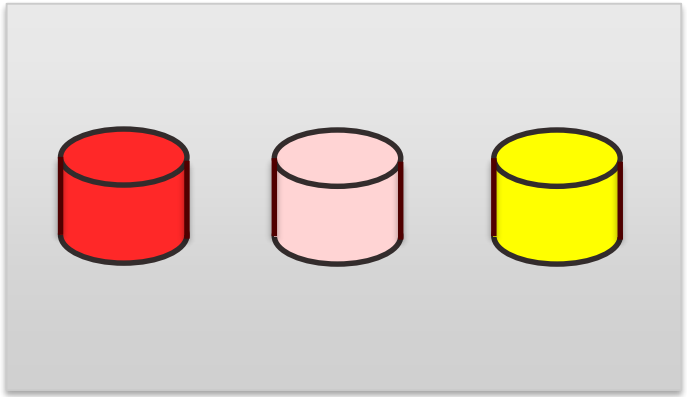


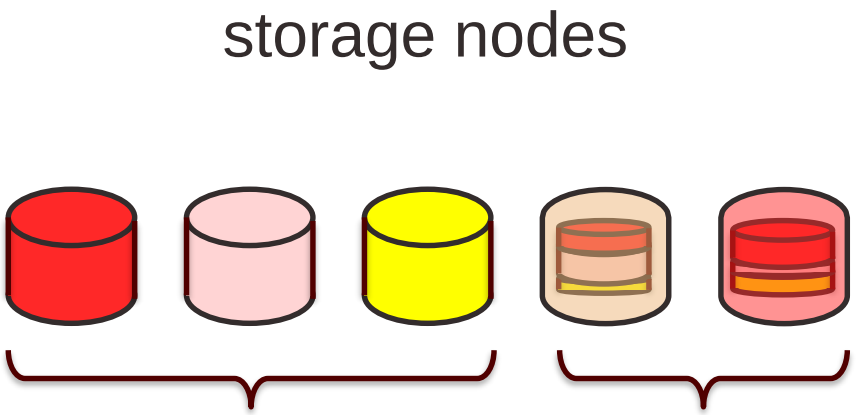
$k=3$
segments

$k=3$
systematic nodes

$m=2$
parity nodes

Erasure Code





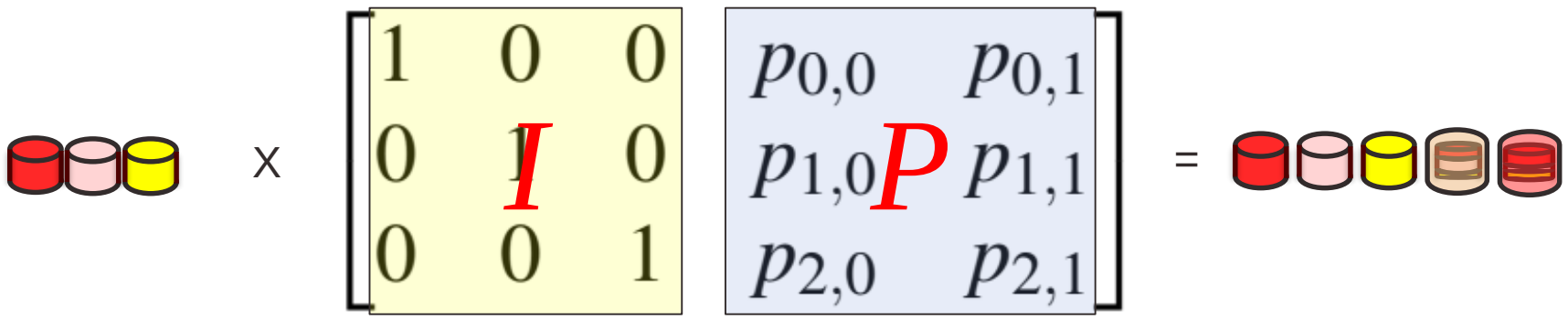
$k=3$
segments

$k=3$
systematic nodes

$m=2$
parity nodes

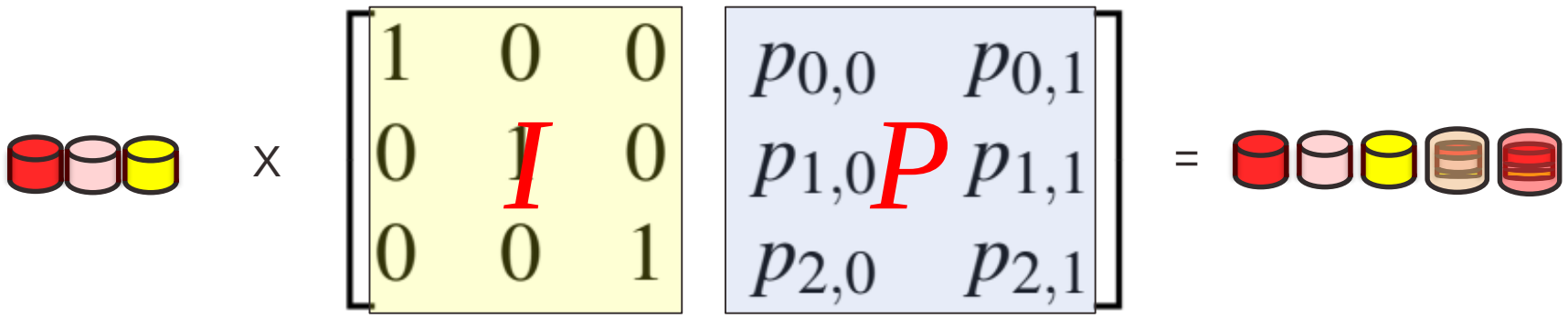
Encoding

- Matrix multiplication on finite field



Encoding

- Matrix multiplication on finite field



All elements and ops in $GF(2^w)$

Galois Field (GF)
Finite field

Decoding

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_{0,0} & p_{0,1} \\ p_{1,0} & p_{1,1} \\ p_{2,0} & p_{2,1} \end{bmatrix}$$

Decoding

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_{0,0} & p_{0,1} \\ p_{1,0} & p_{1,1} \\ p_{2,0} & p_{2,1} \end{bmatrix}$$

Decoding

- Submatrix corresponding to surviving data

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_{0,0} & p_{0,1} \\ p_{1,0} & p_{1,1} \\ p_{2,0} & p_{2,1} \end{bmatrix}$$

$$\left[\begin{array}{c|cc} 0 & p_{0,0} & p_{0,1} \\ 0 & p_{1,0} & p_{1,1} \\ 1 & p_{2,0} & p_{2,1} \end{array}\right]$$

Decoding

- Submatrix corresponding to surviving data
- Invert

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} I \qquad \begin{bmatrix} p_{0,0} & p_{0,1} \\ p_{1,0} & p_{1,1} \\ p_{2,0} & p_{2,1} \end{bmatrix} P$$

$$\left[\begin{array}{c|cc} 0 & p_{0,0} & p_{0,1} \\ 0 & p_{1,0} & p_{1,1} \\ 1 & p_{2,0} & p_{2,1} \end{array}\right]^{-1}$$

Decoding

- Submatrix corresponding to surviving data
- Invert
- Multiplication

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad I$$

$$\begin{bmatrix} p_{0,0} & p_{0,1} \\ p_{1,0} & p_{1,1} \\ p_{2,0} & p_{2,1} \end{bmatrix} \quad P$$

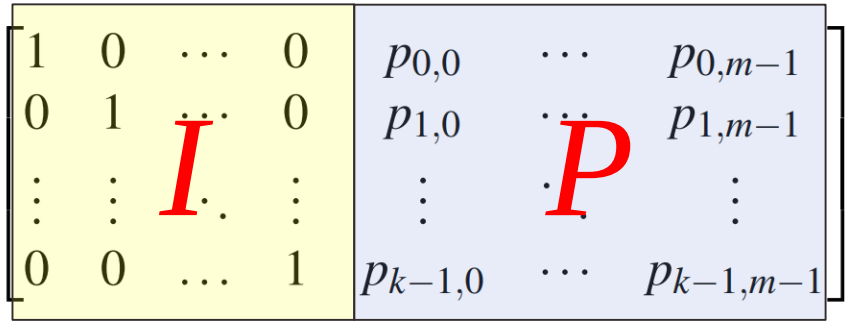
x

$$\begin{bmatrix} 0 & p_{0,0} & p_{0,1} \\ 0 & p_{1,0} & p_{1,1} \\ 1 & p_{2,0} & p_{2,1} \end{bmatrix}^{-1}$$

=

Cauchy Reed-Solomon Codes

- Direct assign P to Cauchy matrix



Cauchy Reed-Solomon Codes

$$X = (x_1, \ldots, x_k) \qquad Y = (y_1, \ldots, y_m)$$

$$C = \begin{bmatrix} \frac{1}{x_1 + y_1} & \frac{1}{x_1 + y_2} & \cdots & \frac{1}{x_1 + y_m} \\ \frac{1}{x_2 + y_1} & \frac{1}{x_2 + y_2} & \cdots & \frac{1}{x_2 + y_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_k + y_1} & \frac{1}{x_k + y_2} & \cdots & \frac{1}{x_k + y_m} \end{bmatrix}$$

Cauchy Reed-Solomon Codes

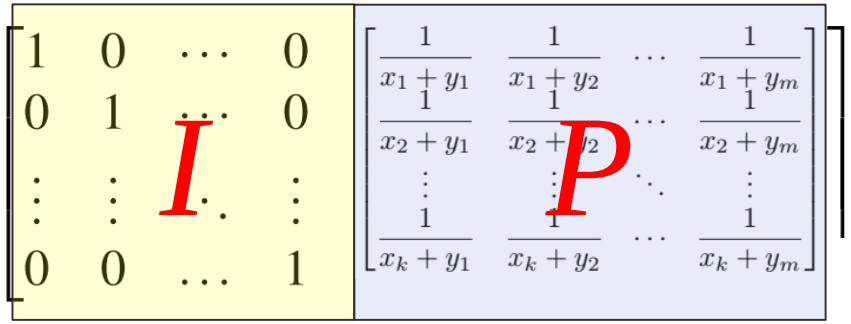
$$X = (x_1, \ldots, x_k) \qquad Y = (y_1, \ldots, y_m)$$

$$C = \begin{bmatrix} \frac{1}{x_1 + y_1} & \frac{1}{x_1 + y_2} & \cdots & \frac{1}{x_1 + y_m} \\ \frac{1}{x_2 + y_1} & \frac{1}{x_2 + y_2} & \cdots & \frac{1}{x_2 + y_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_k + y_1} & \frac{1}{x_k + y_2} & \cdots & \frac{1}{x_k + y_m} \end{bmatrix}$$

All elements and ops in $GF(2^w)$

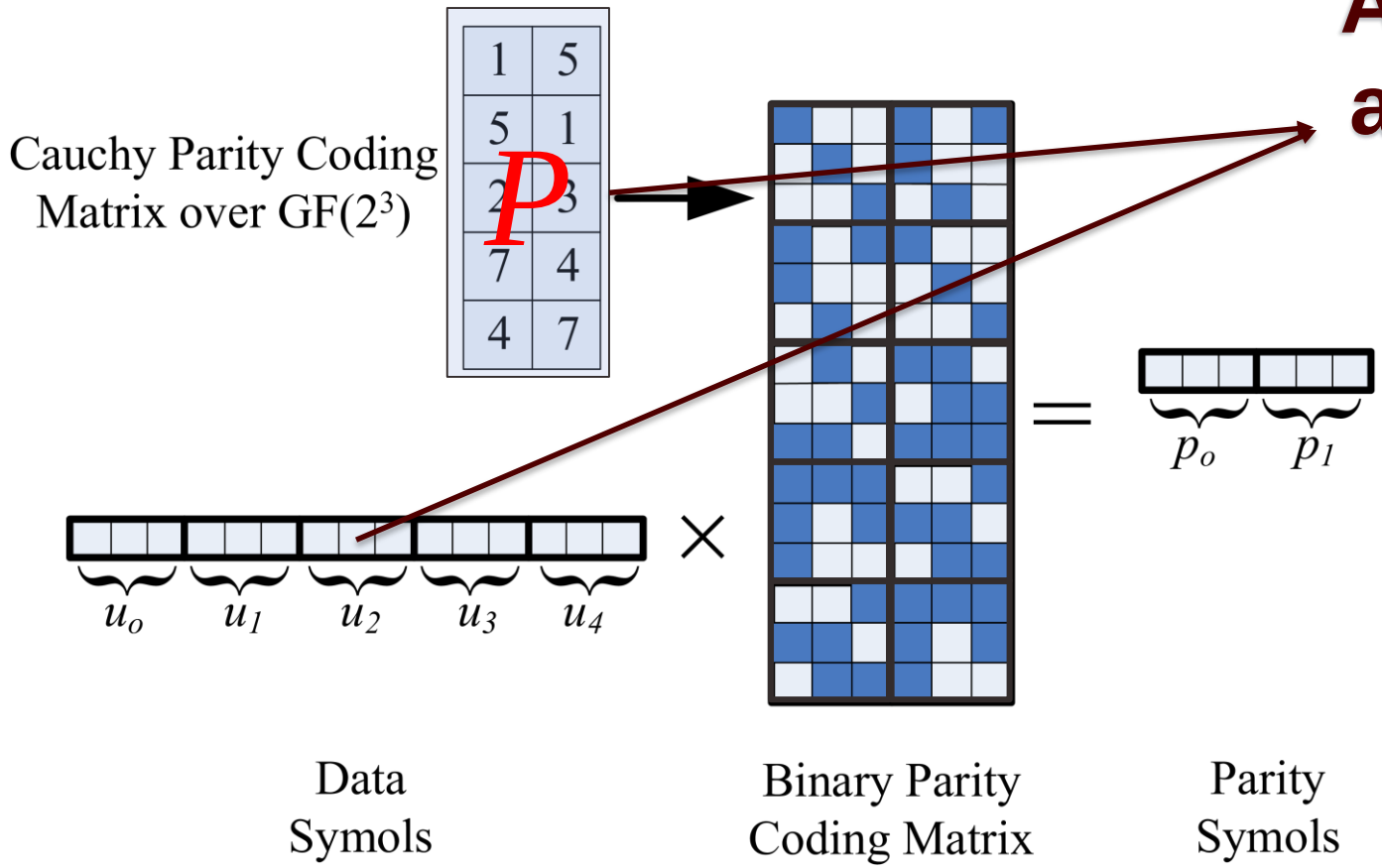
Cauchy Reed-Solomon Codes

- Direct assign P to Cauchy matrix



Fast GF Ops – Binary Representation

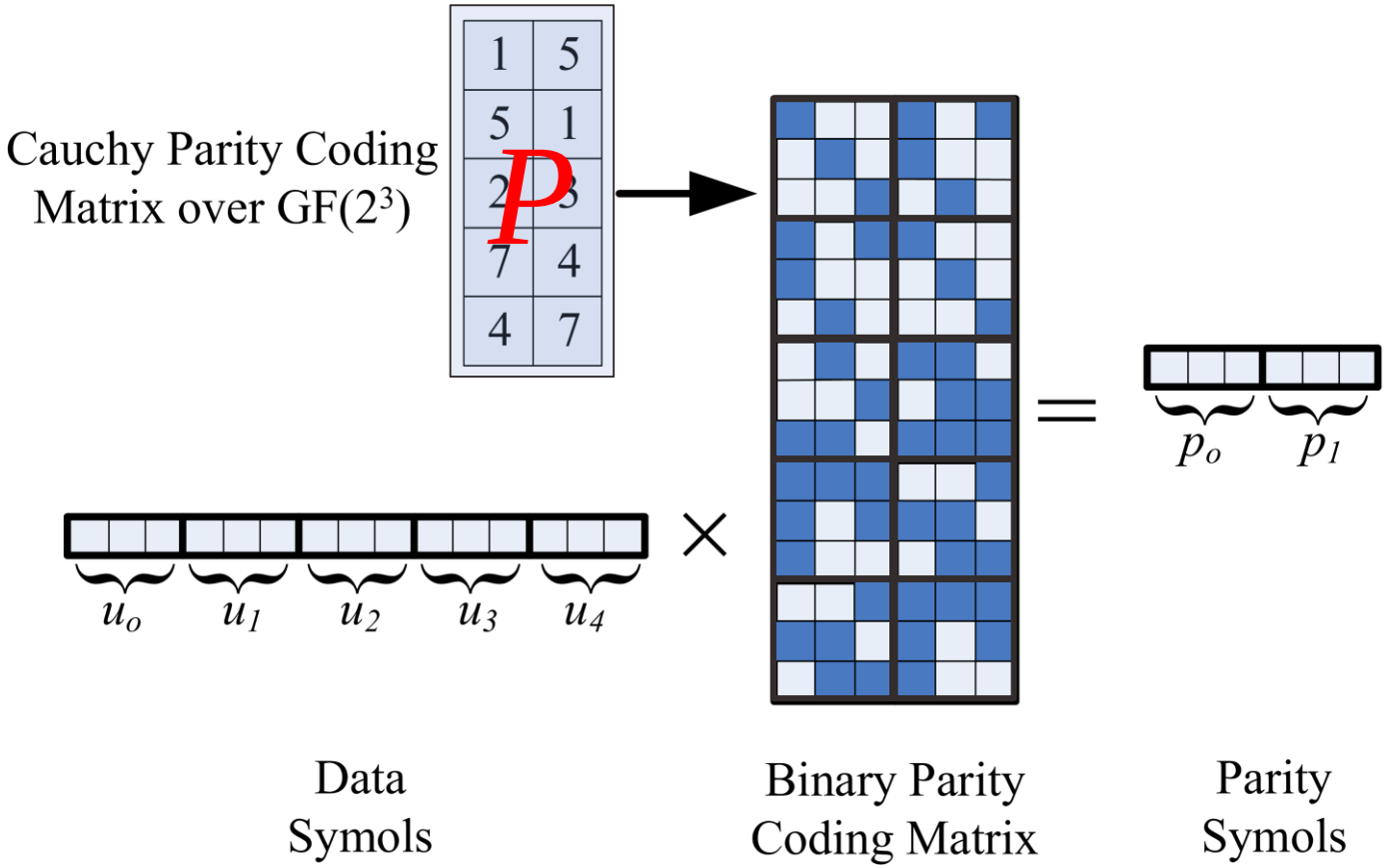
- Binary representation of GF elements



All coefficients and messages are in $GF(2^w)$

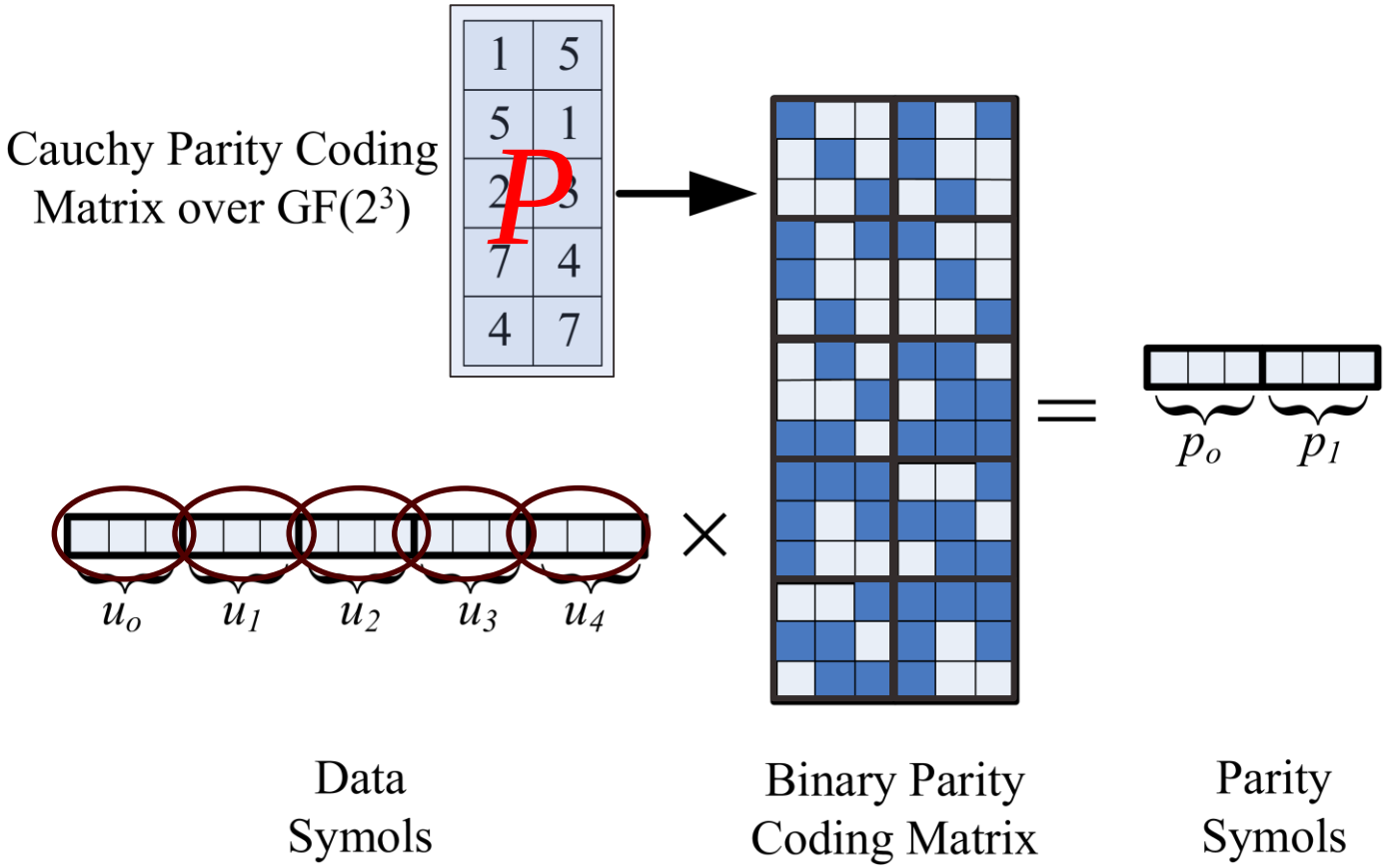
Fast GF Ops – Binary Representation

- Message bit-vector representation



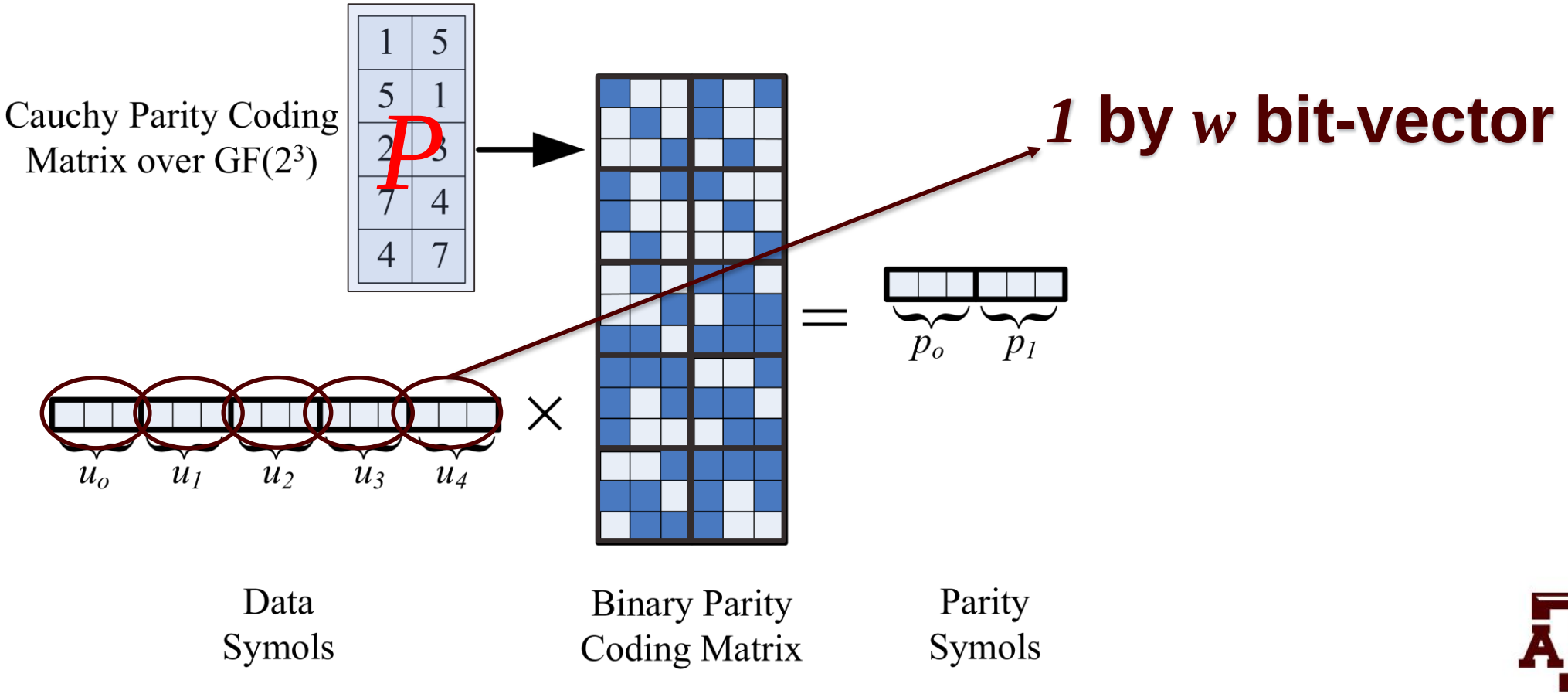
Fast GF Ops – Binary Representation

- Message bit-vector representation



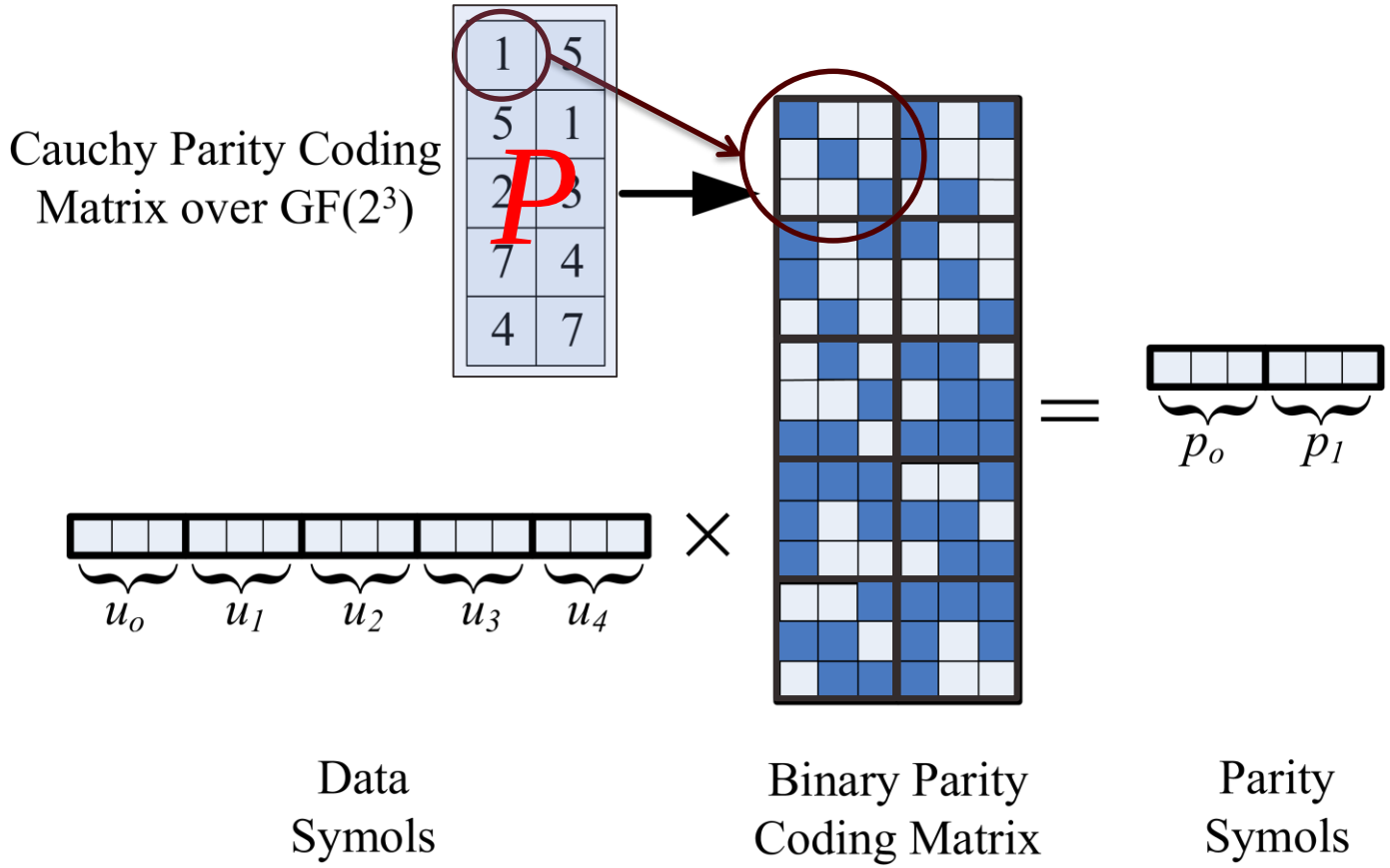
Fast GF Ops – Binary Representation

- Message bit-vector representation



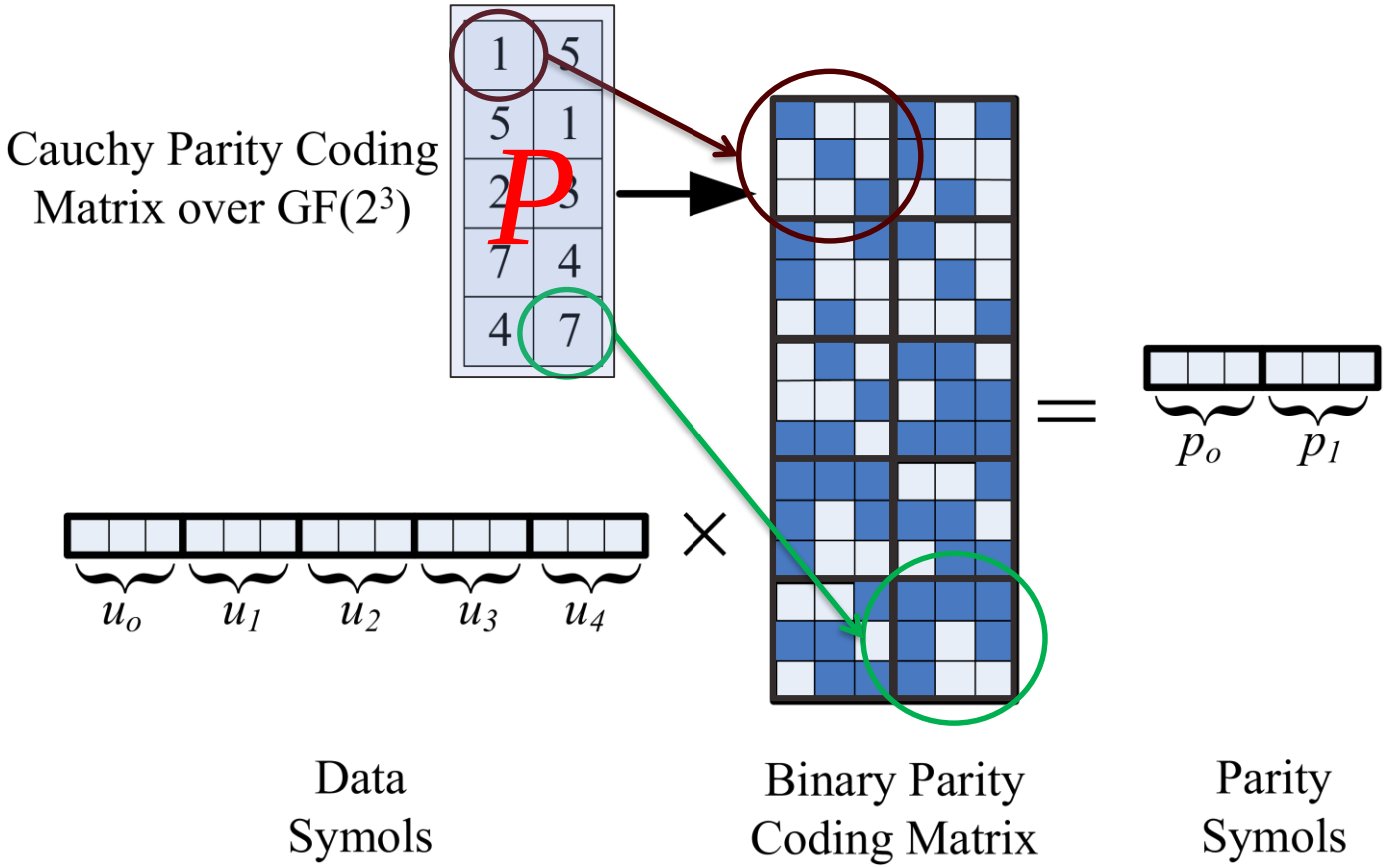
Fast GF Ops – Binary Representation

- Bitmatrix representation



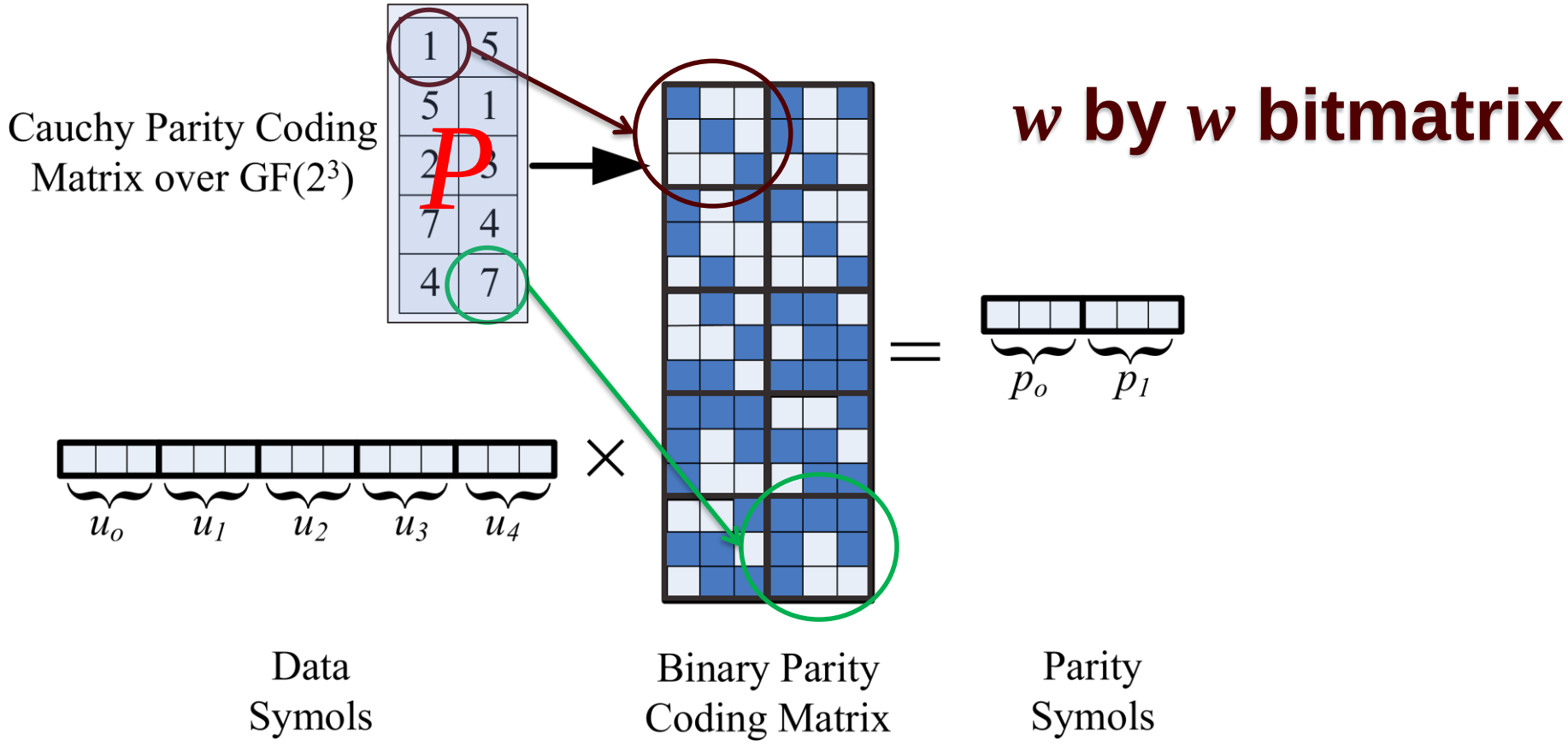
Fast GF Ops – Binary Representation

- Bitmatrix representation



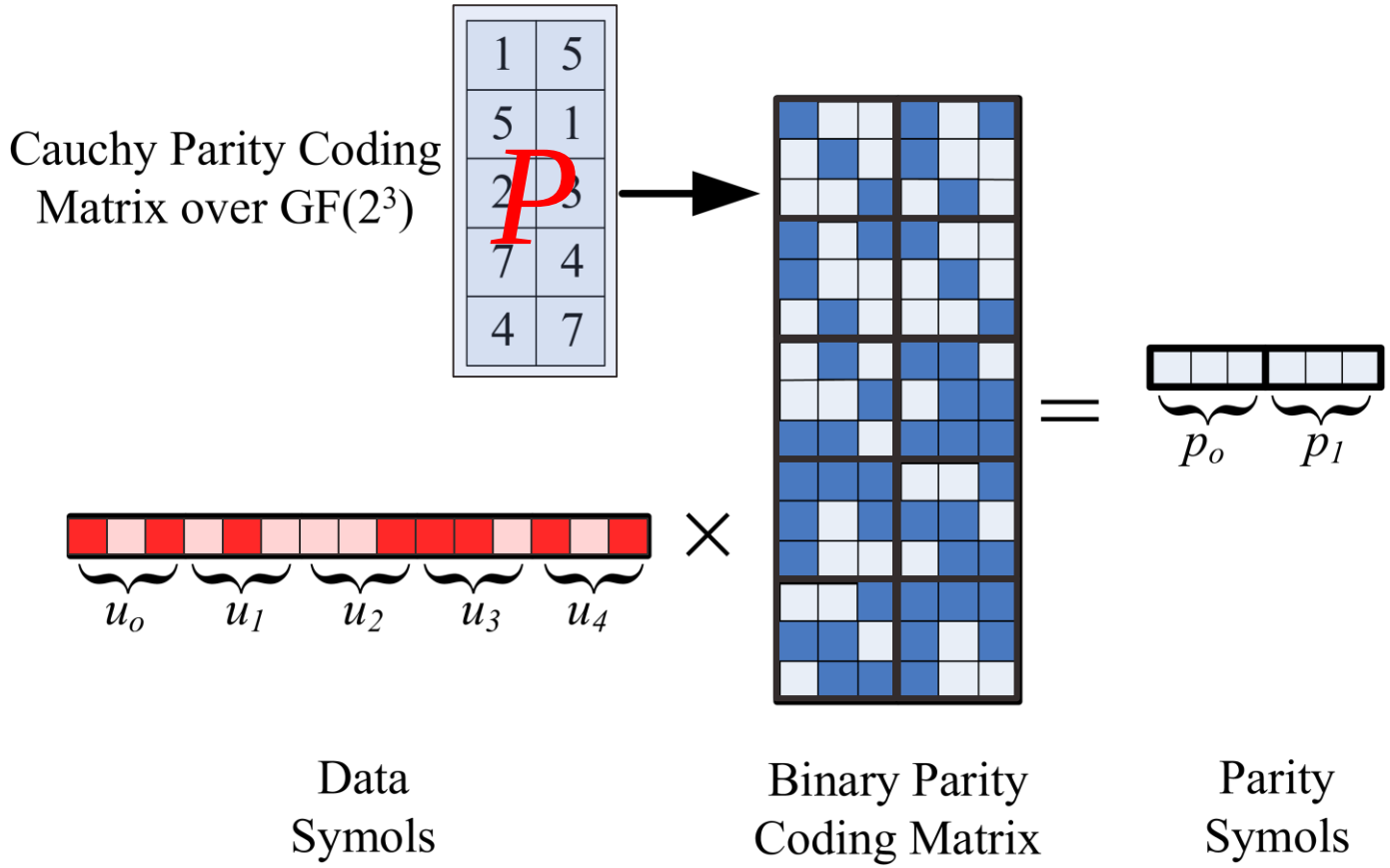
Fast GF Ops – Binary Representation

- Bitmatrix representation



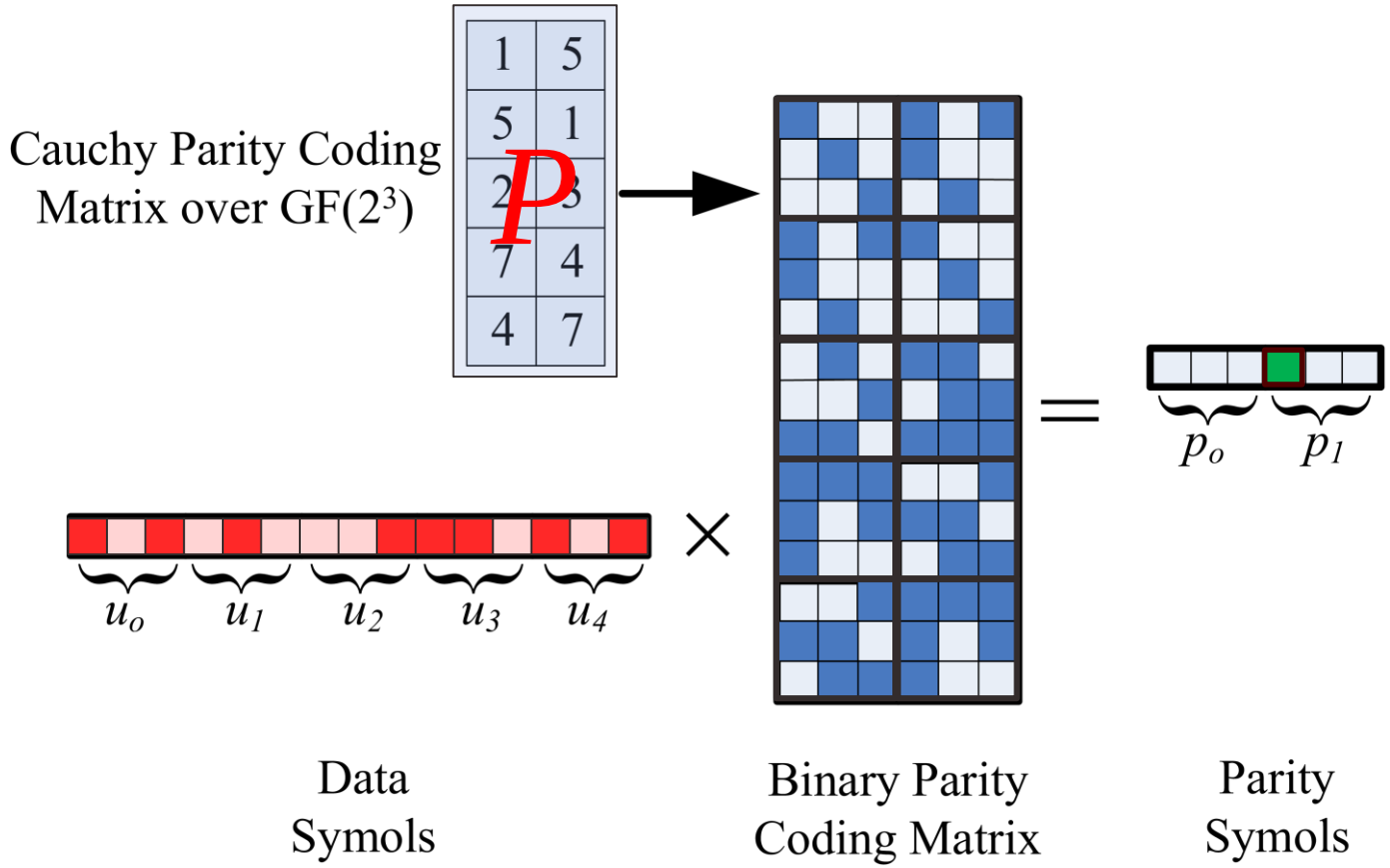
Fast GF Ops – Binary Representation

- Matrix multiplication on GF -> XORs



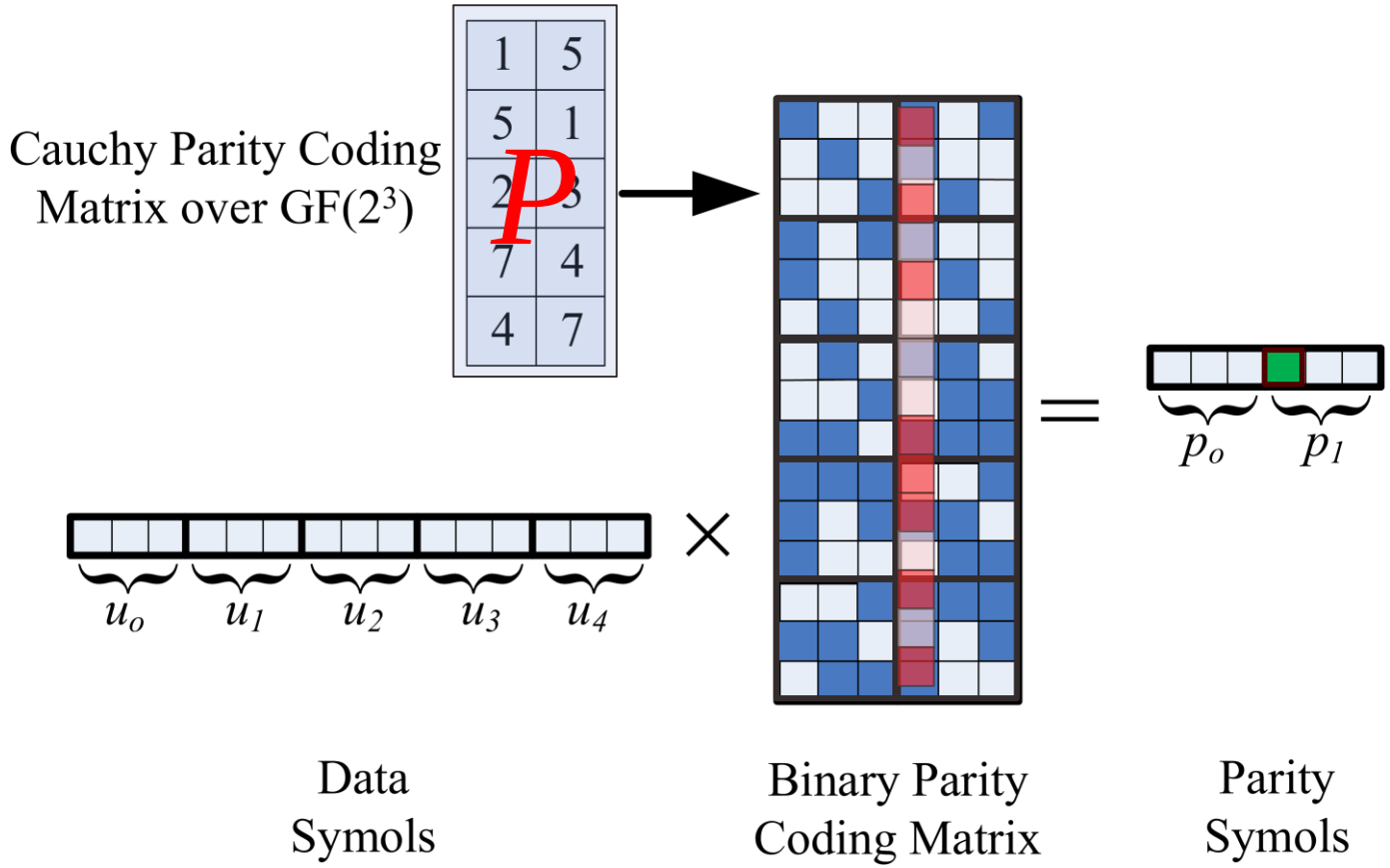
Fast GF Ops – Binary Representation

- Matrix multiplication on GF -> XORs



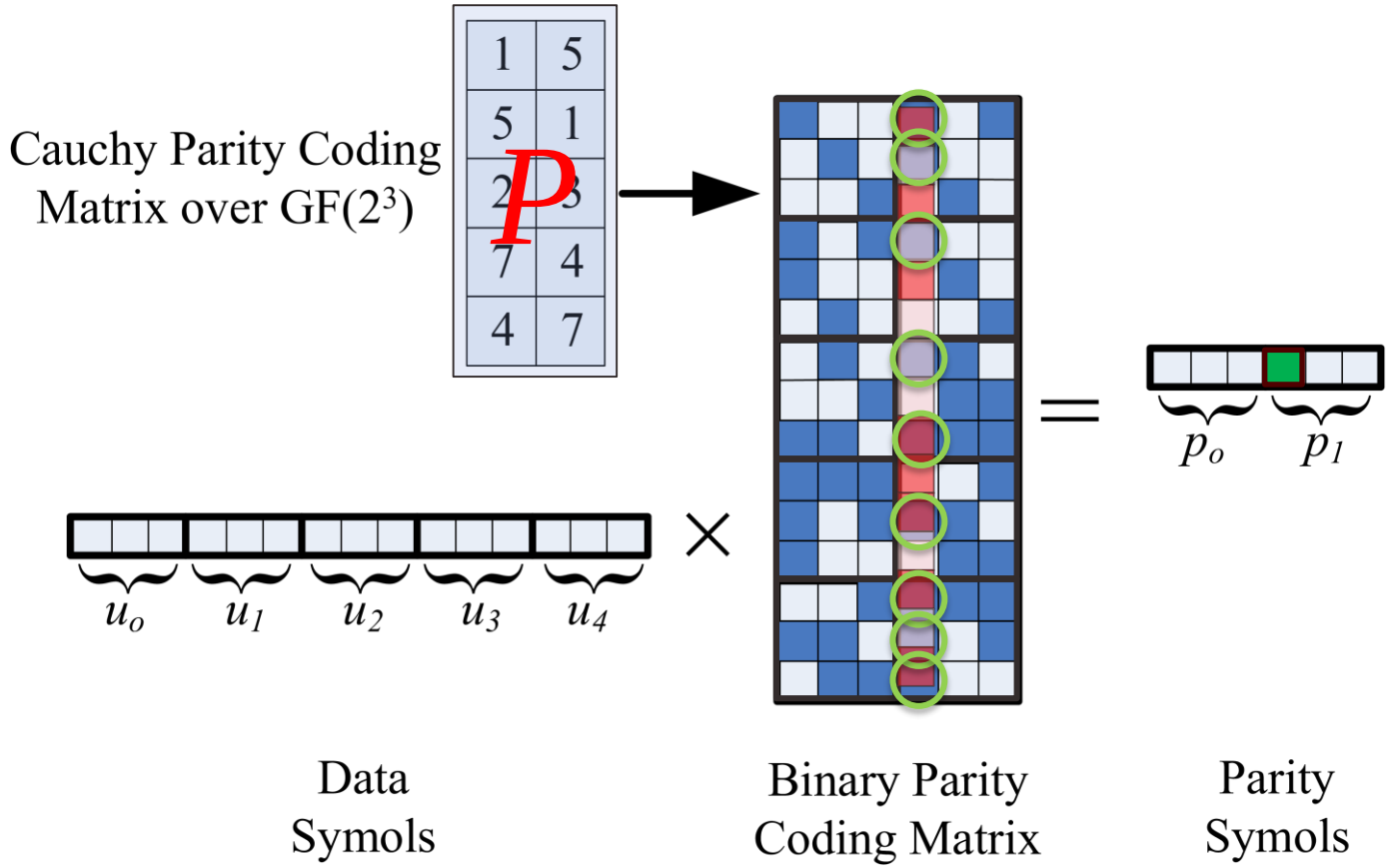
Fast GF Ops – Binary Representation

- Matrix multiplication on GF -> XORs



Fast GF Ops – Binary Representation

- Matrix multiplication on GF -> XORs



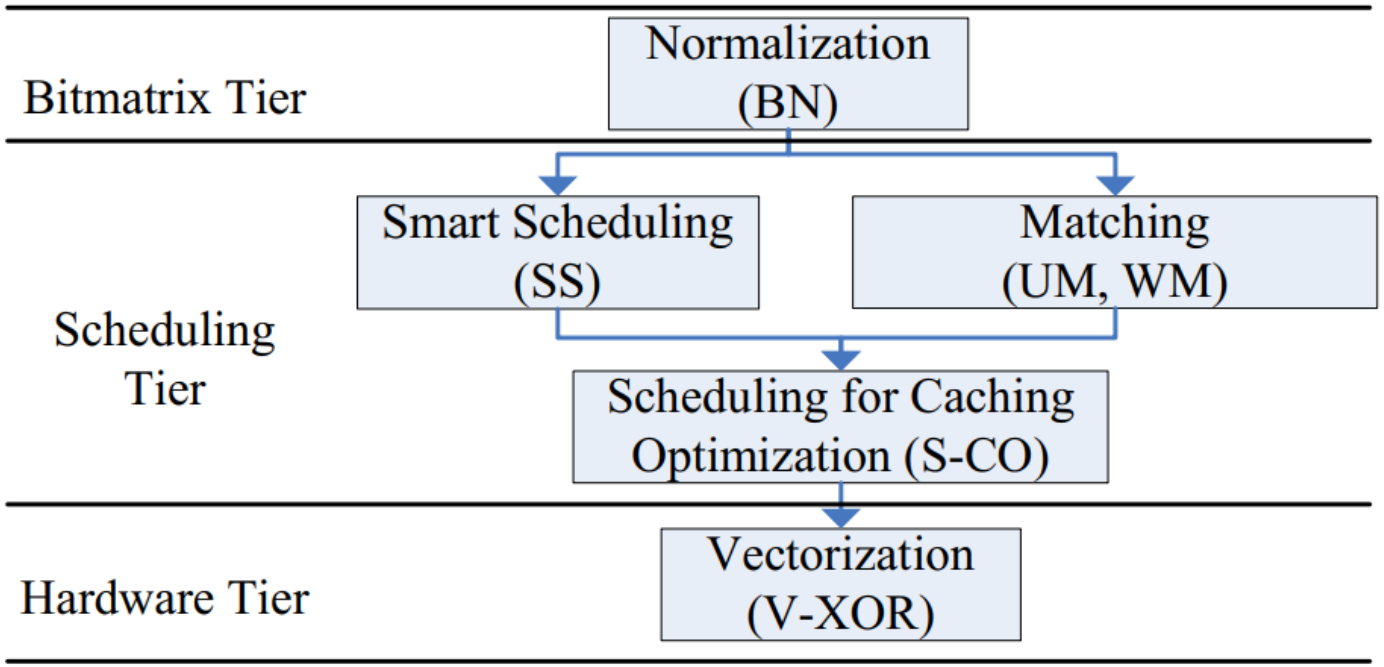
Contents

- Motivation
- Background and Review

- Evaluating Individual Techniques
- Find the Best Strategy under Optimized Bitmatrix
- Proposed Coding Procedure and Evaluation

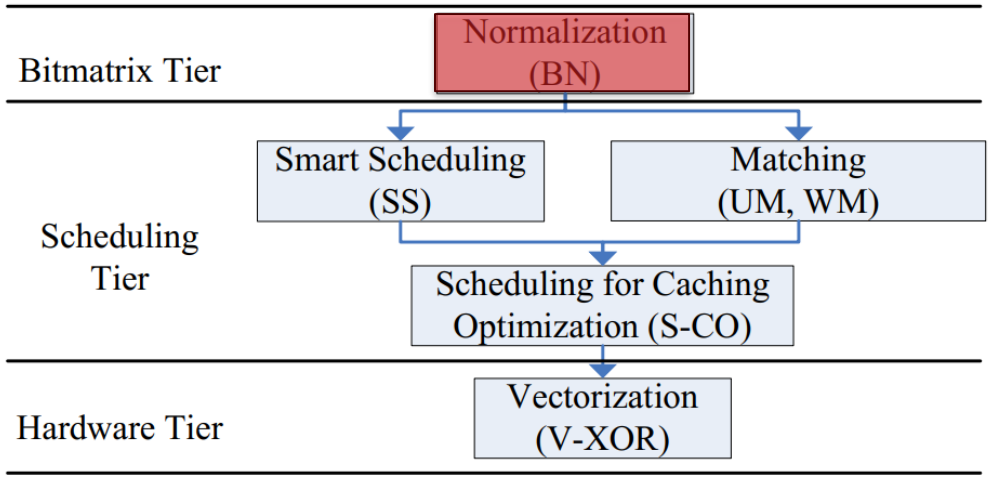
- Conclusion

Techniques Tiers



Bitmatrix Normalization (BN)

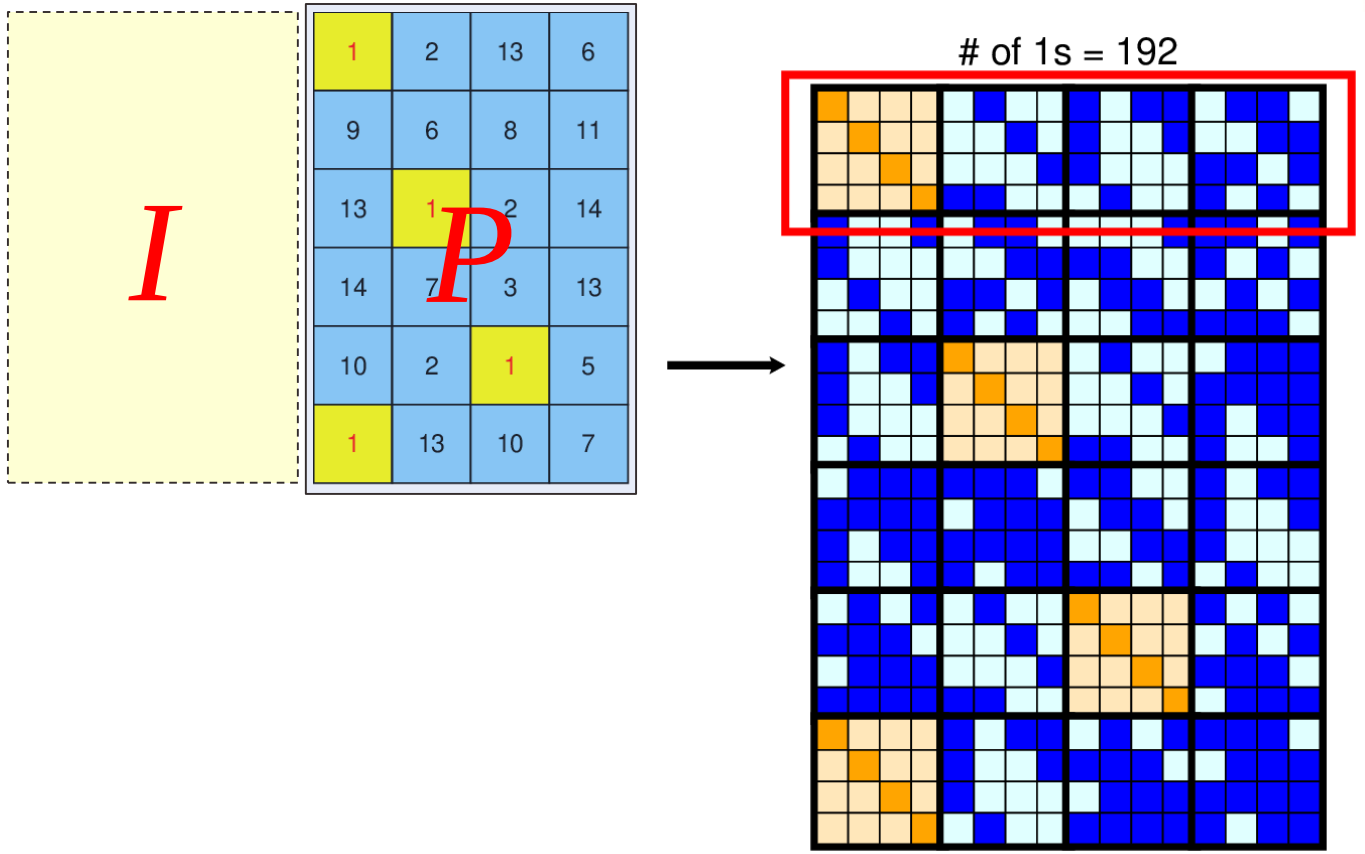
- Normalize by each element, find less bit 1s

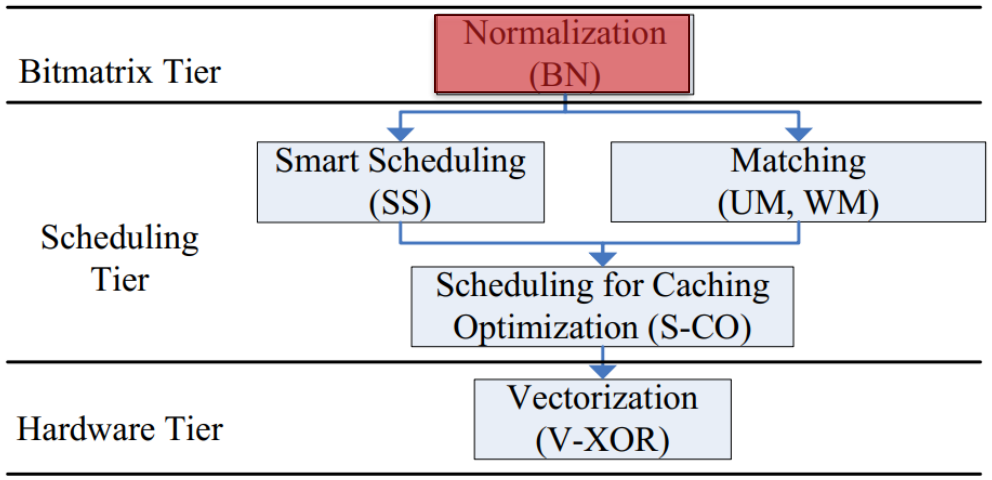


James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s

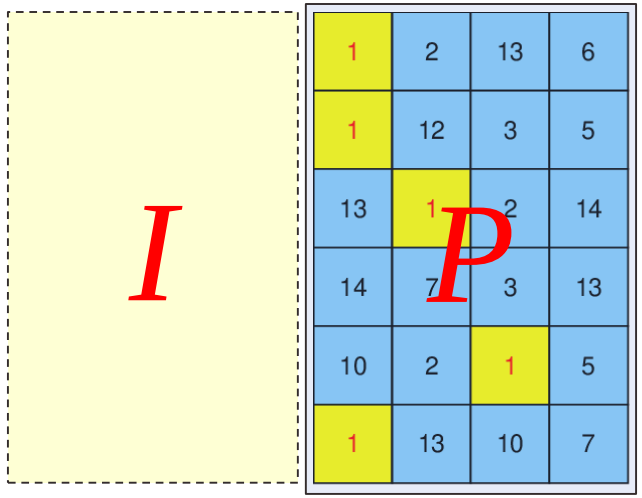


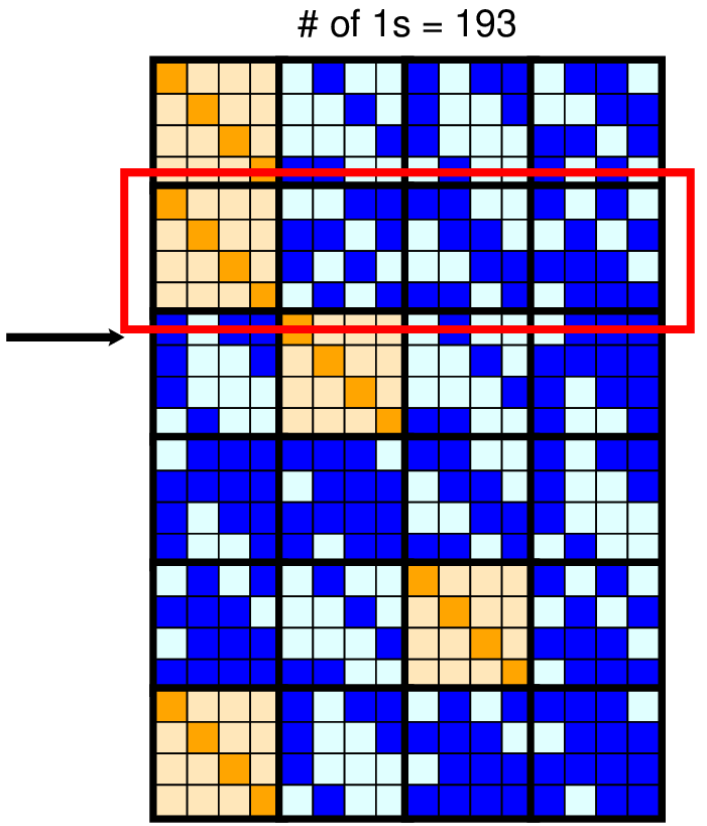


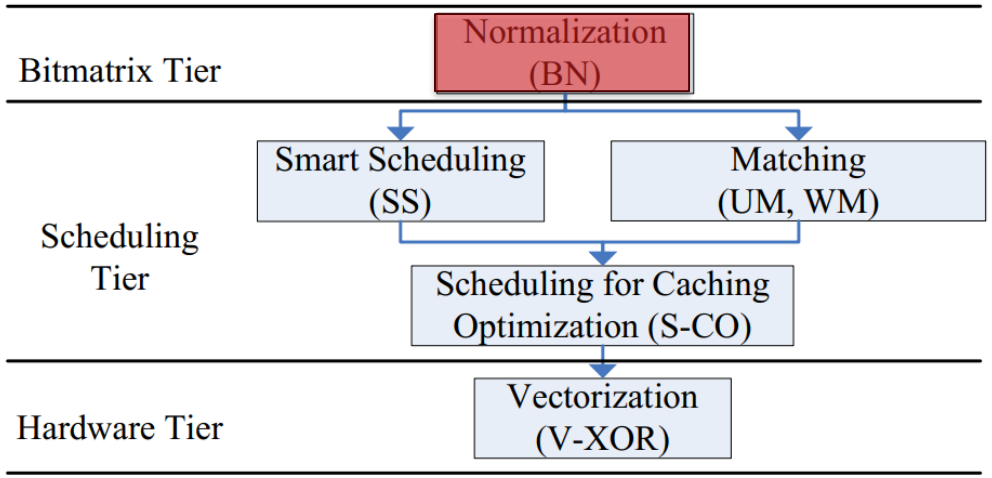
James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s



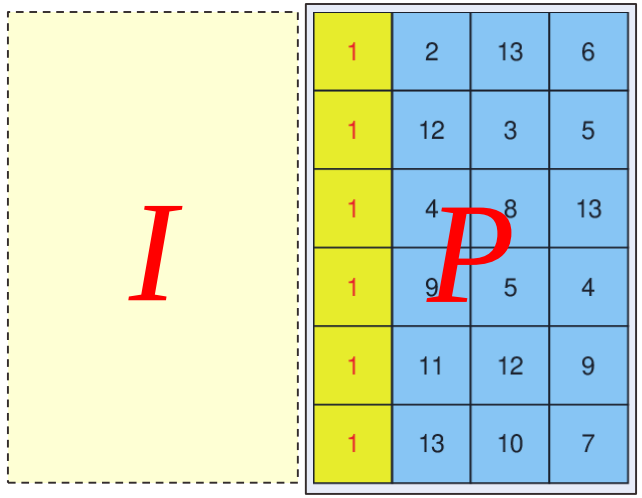


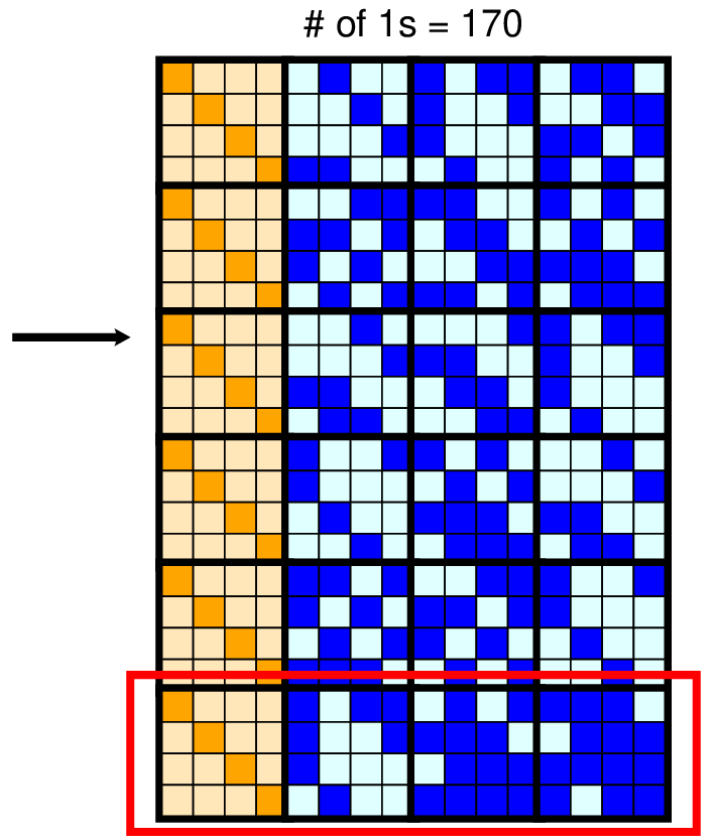


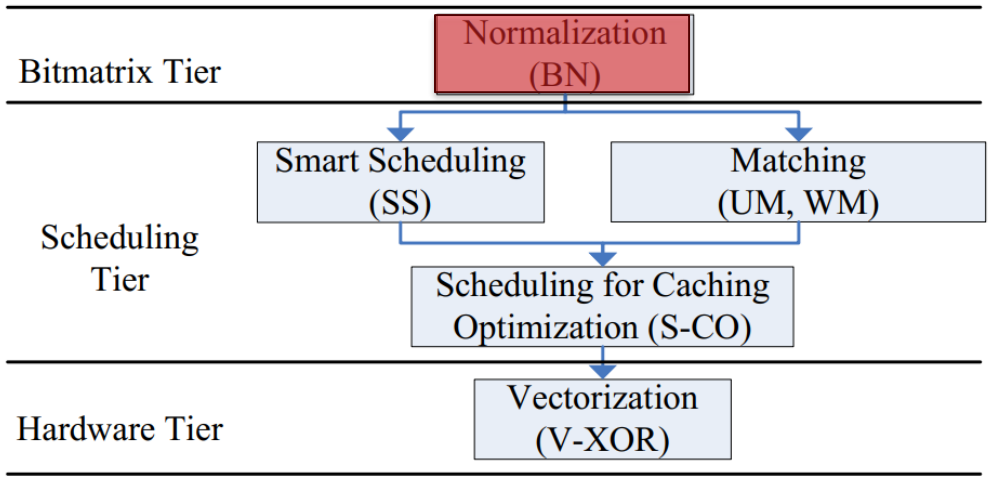
James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s



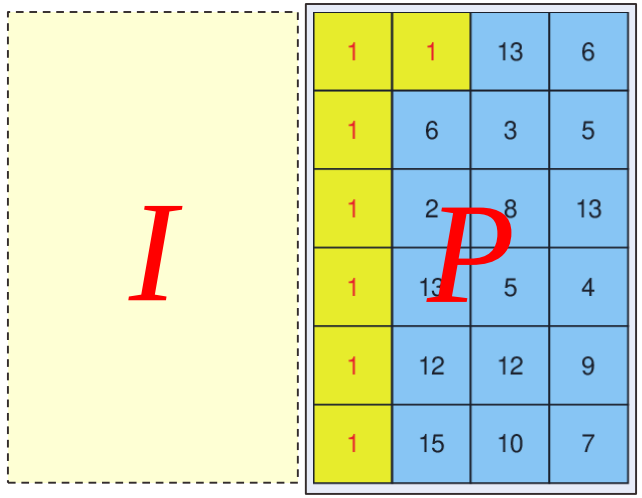


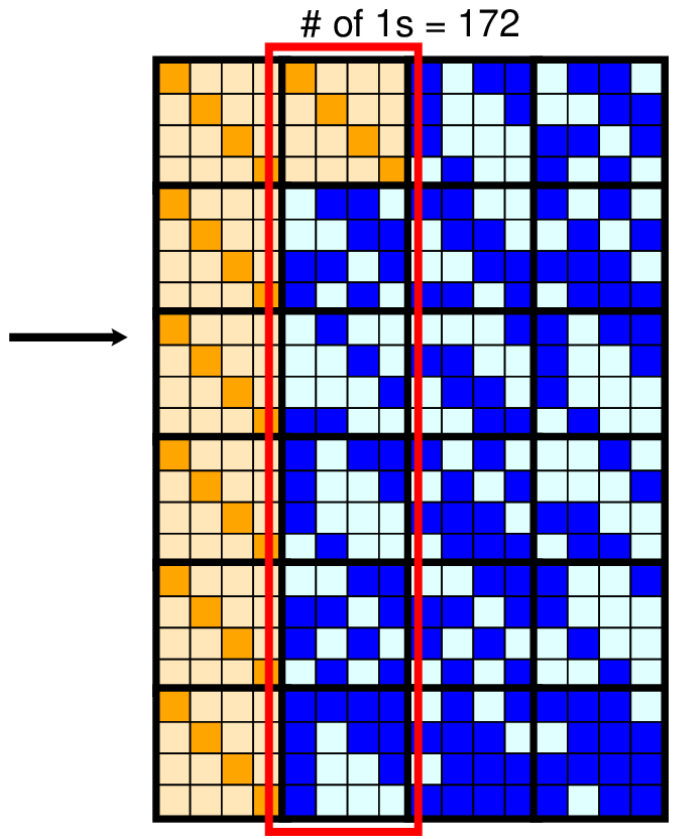


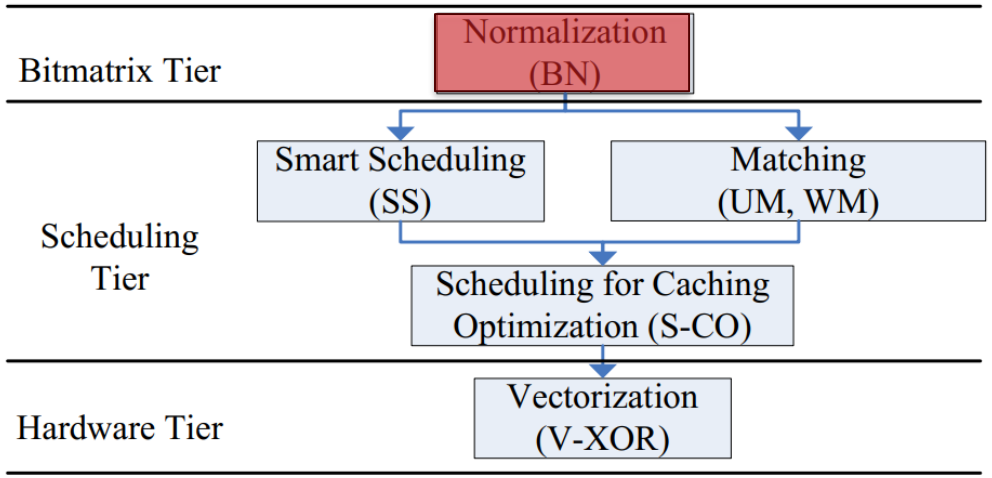
James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s



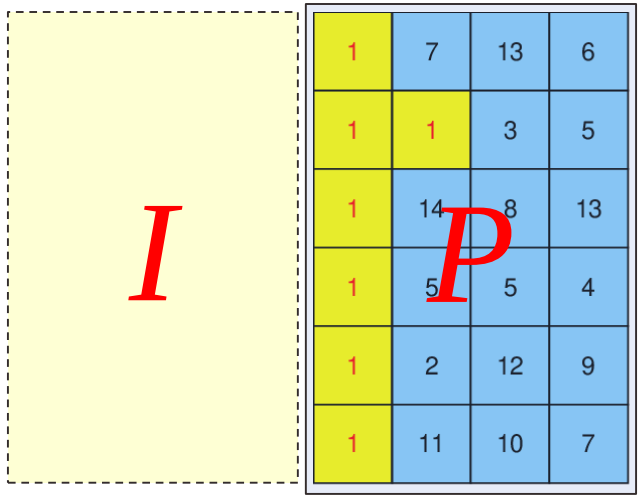


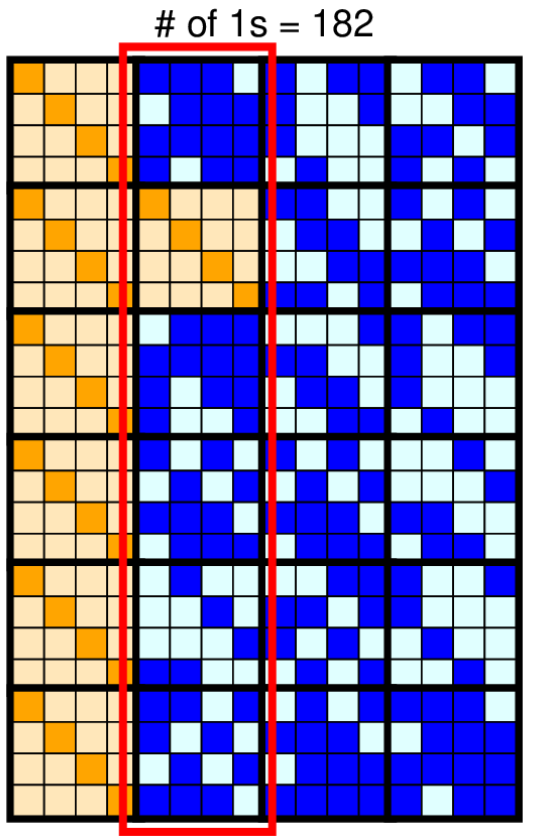


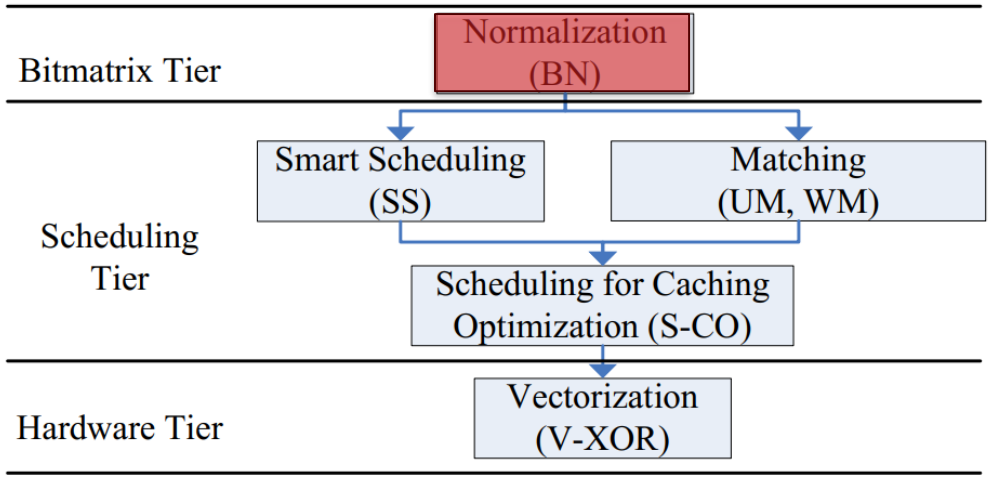
James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s



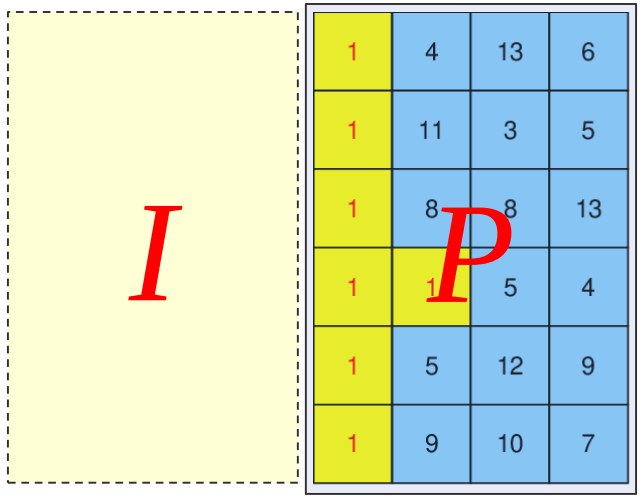


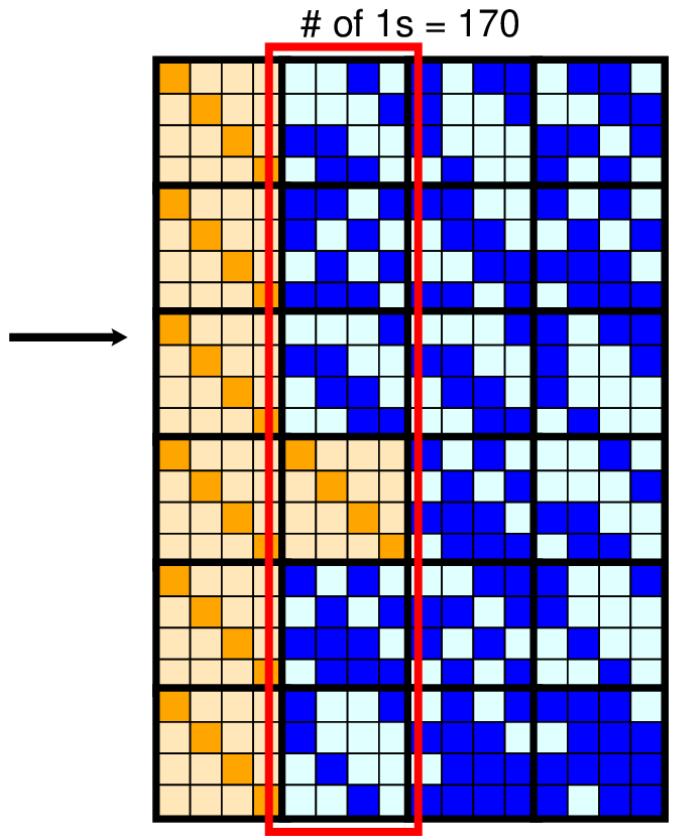


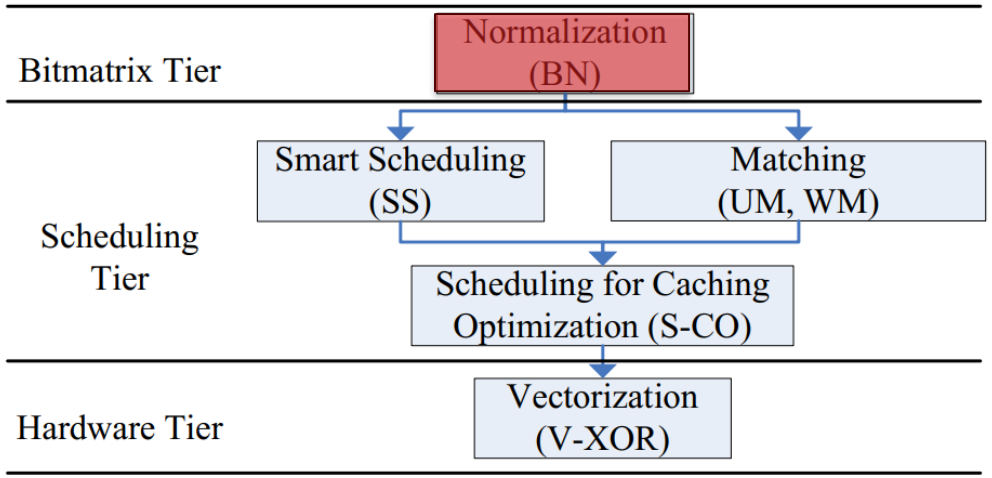
James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s



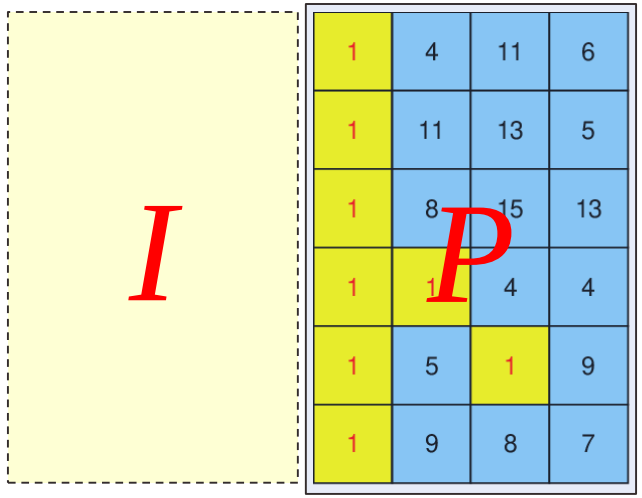


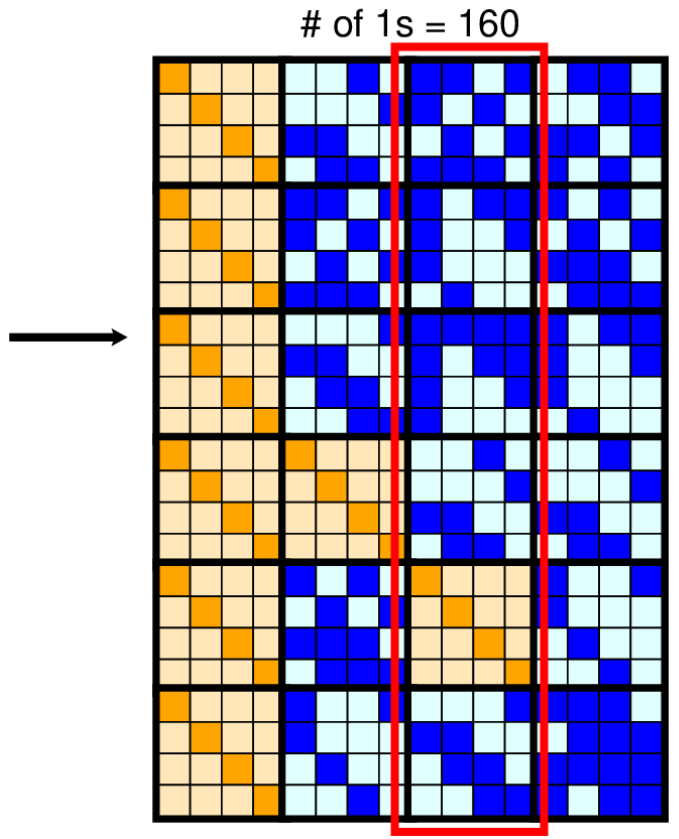


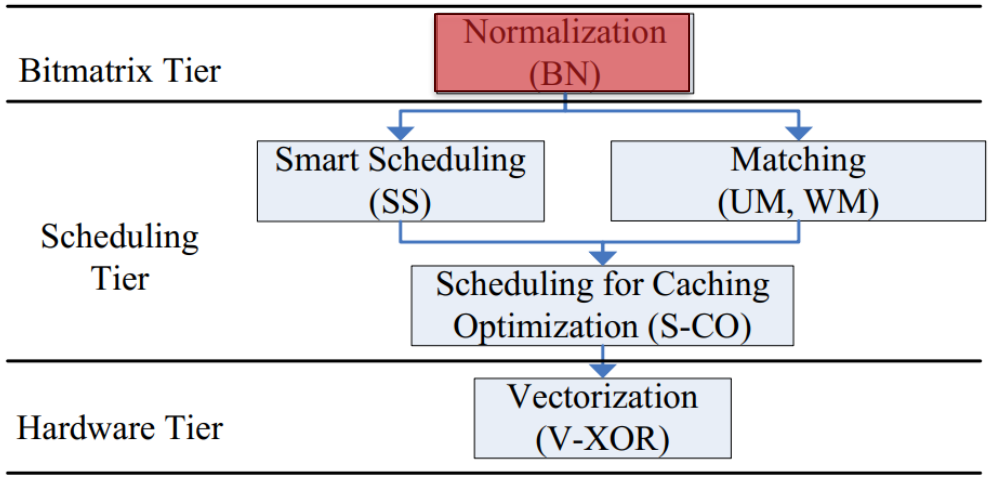
James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s



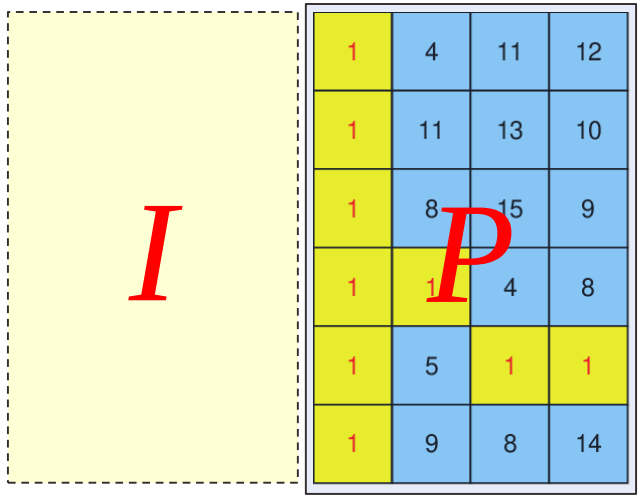


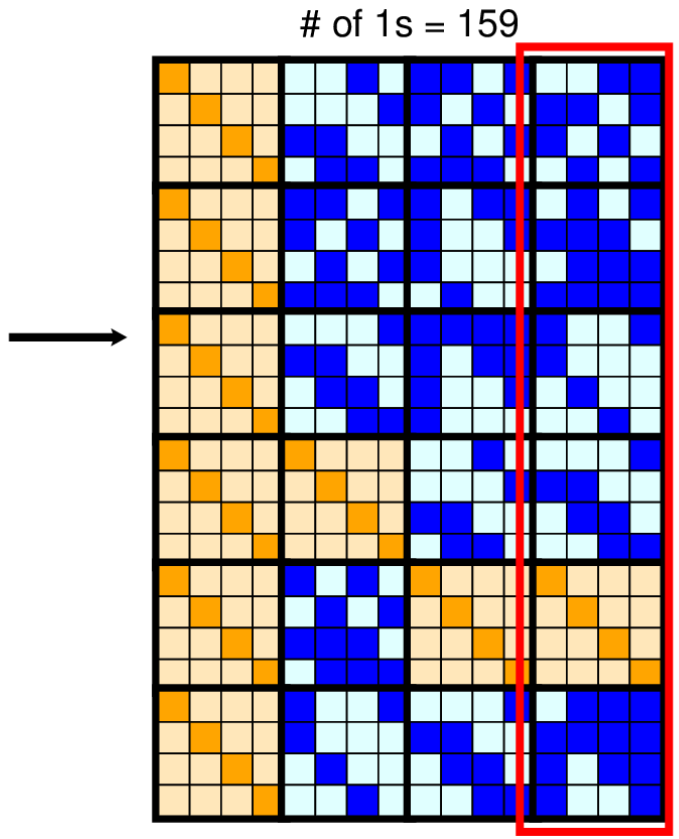


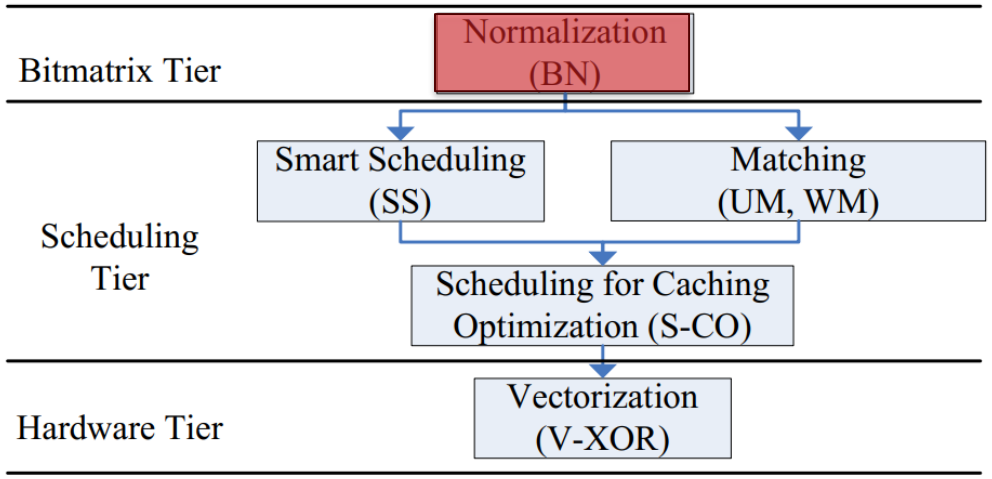
James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

Bitmatrix Normalization (BN)

- Normalize by each element, find less bit 1s



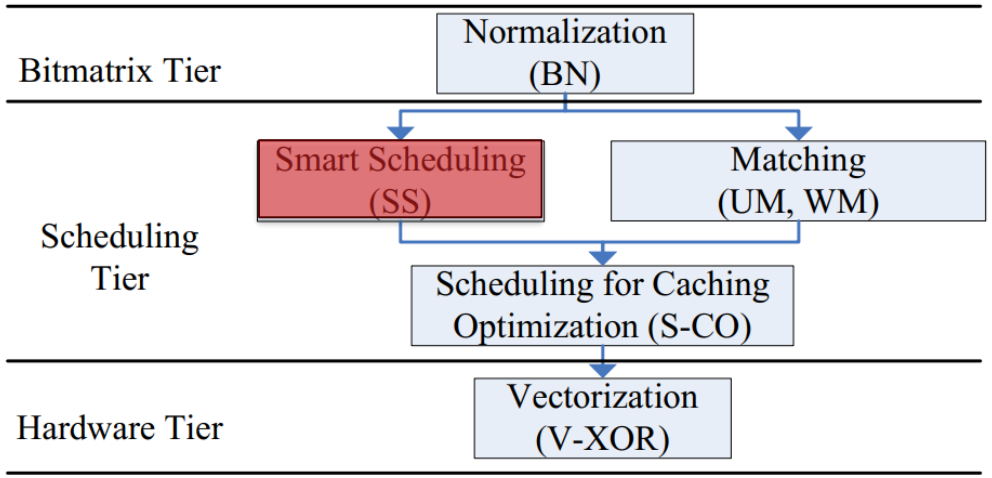




James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

Smart Scheduling (SS)

- Utilize both message and computed parities

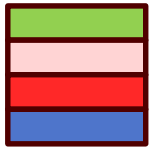


James S. Plank. The RAID-6 liberation code. In Proceedings of the 6th Usenix Conference on File and Storage Technologies, pages 97–110, 2008.

Smart Scheduling (SS)

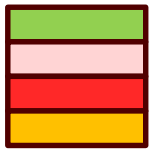
- Utilize both message and computed parities

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4$$

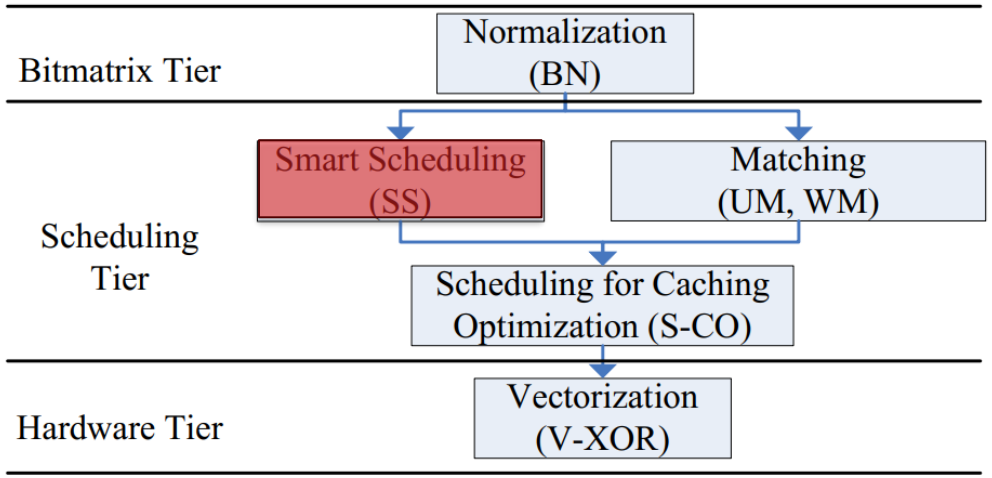


3 XORs

$$p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5$$



3 XORs



James S. Plank. The RAID-6 liberation code. In Proceedings of the 6th Usenix Conference on File and Storage Technologies, pages 97–110, 2008.

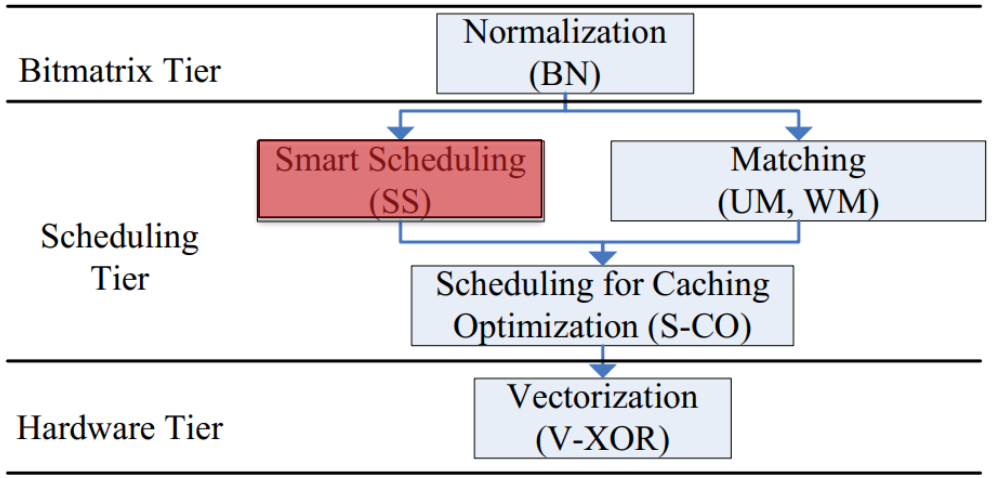
Smart Scheduling (SS)

- Utilize both message and computed parities

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4$$

3 XORs

3 XORs

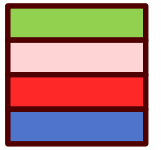


James S. Plank. The RAID-6 liberation code. In Proceedings of the 6th Usenix Conference on File and Storage Technologies, pages 97–110, 2008.

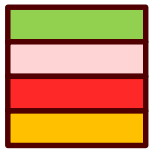
Smart Scheduling (SS)

- Utilize both message and computed parities

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4$$

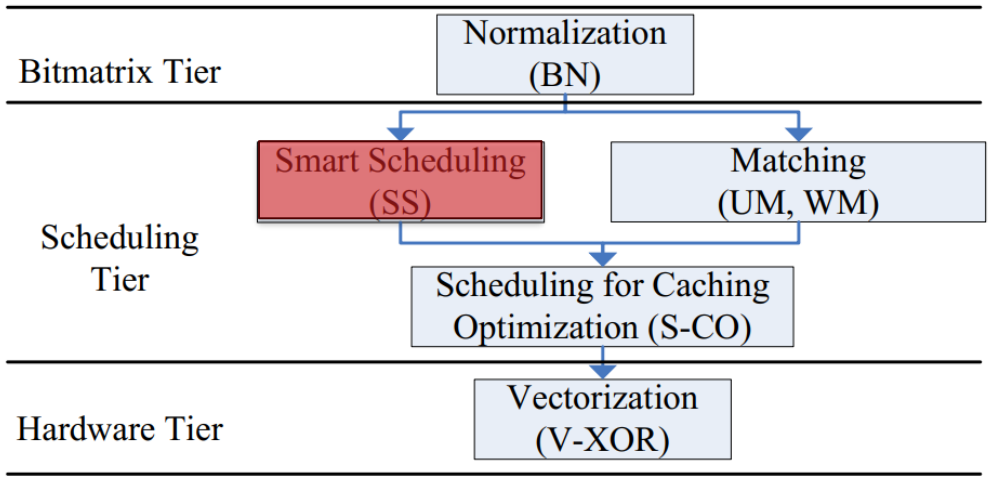


3 XORs



3 XORs

$$p_1 = p_0 \oplus u_4 \oplus u_5$$

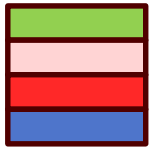


James S. Plank. The RAID-6 liberation code. In Proceedings of the 6th Usenix Conference on File and Storage Technologies, pages 97–110, 2008.

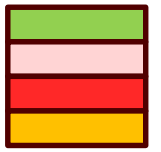
Smart Scheduling (SS)

- Utilize both message and computed parities

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4$$

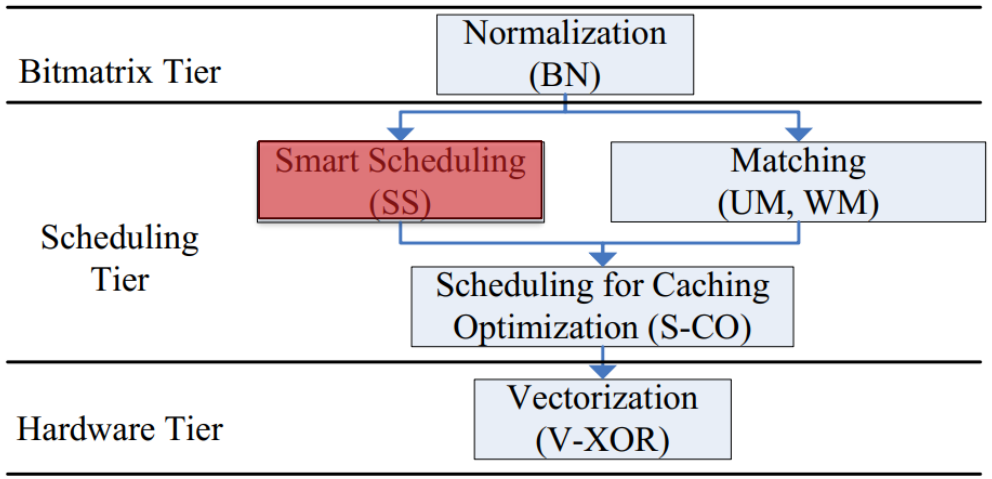


3 XORs



3 XORs

$$p_1 = p_0 \oplus u_4 \oplus u_5$$

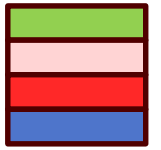


James S. Plank. The RAID-6 liberation code. In Proceedings of the 6th Usenix Conference on File and Storage Technologies, pages 97–110, 2008.

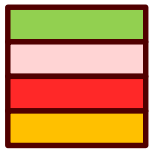
Smart Scheduling (SS)

- Utilize both message and computed parities

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4$$

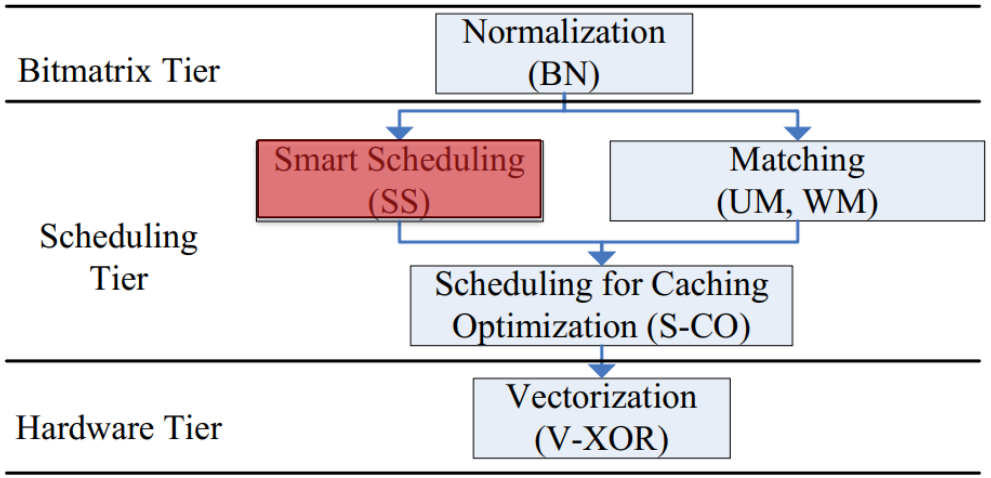


3 XORs



3 XORs

$$p_1 = p_0 \oplus u_4 \oplus u_5$$

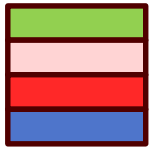


James S. Plank. The RAID-6 liberation code. In Proceedings of the 6th Usenix Conference on File and Storage Technologies, pages 97–110, 2008.

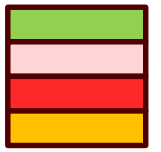
Smart Scheduling (SS)

- Utilize both message and computed parities

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4$$



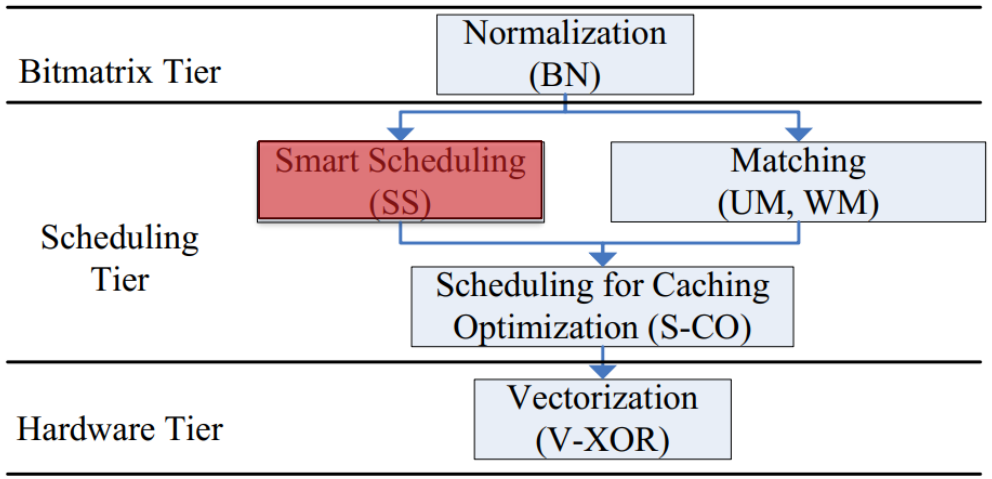
3 XORs



3 XORs

$$p_1 = p_0 \oplus u_4 \oplus u_5$$

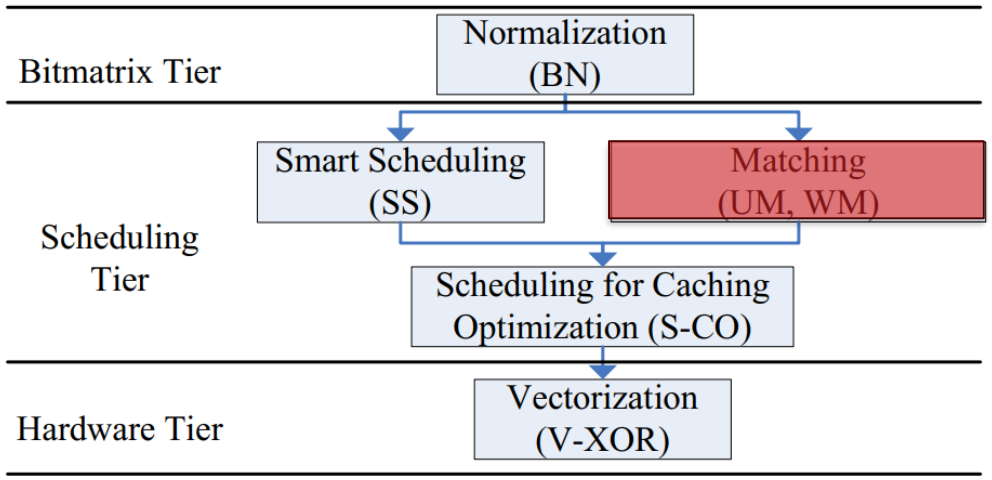
2 XORs

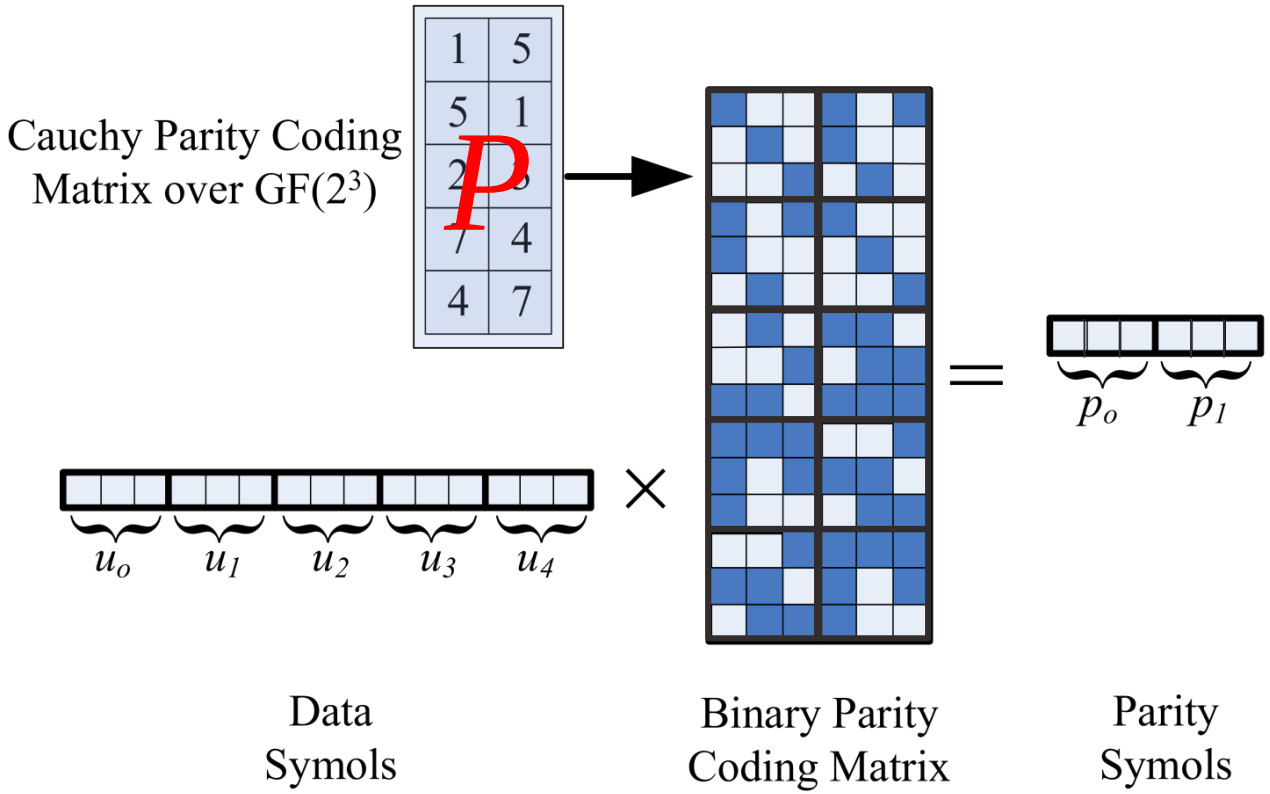


James S. Plank. The RAID-6 liberation code. In Proceedings of the 6th Usenix Conference on File and Storage Technologies, pages 97–110, 2008.

Matching (UM, WM)

- Common XORs of pair message bits
- Unweighted/weighted greedy algorithm

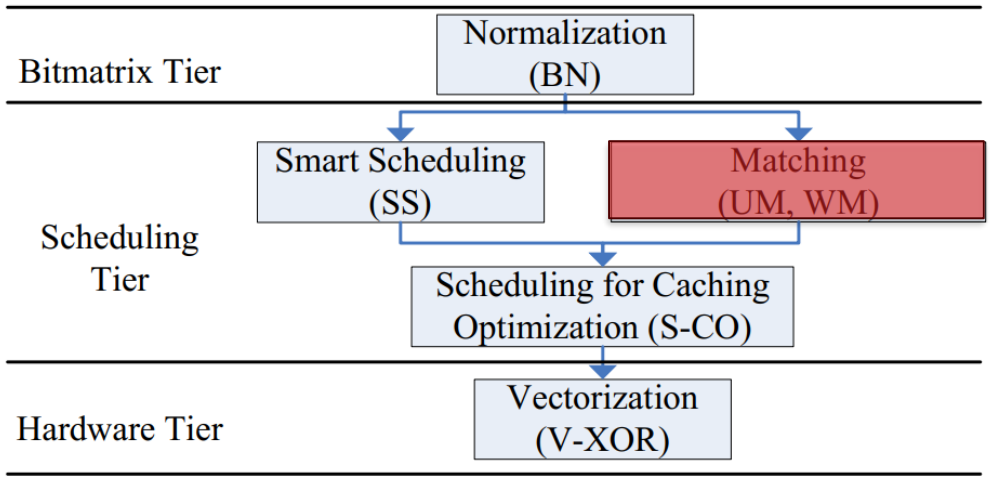


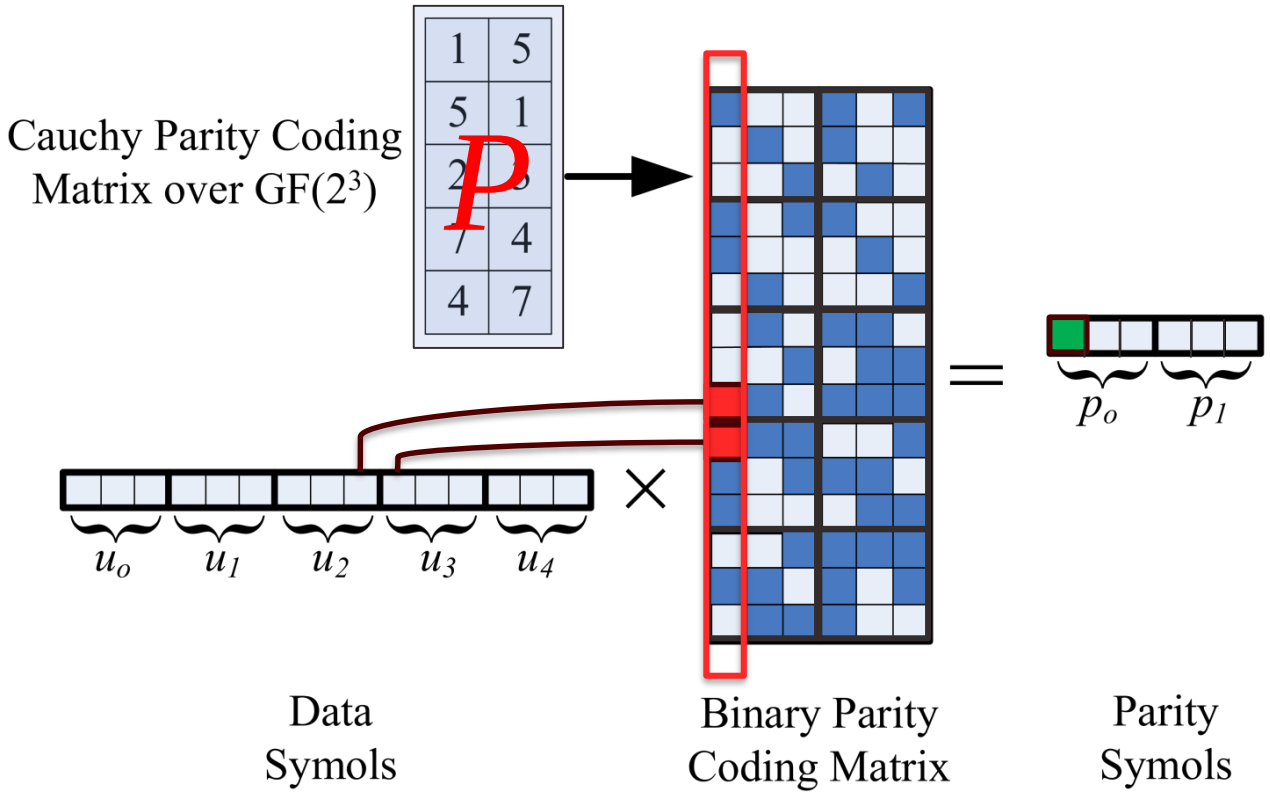


Cheng Huang, Jin Li, and Minghua Chen. On optimizing XOR-based codes for fault-tolerant storage applications. In Proceedings of Information Theory Workshop, pages 218–223, 2007.

Matching (UM, WM)

- Common XORs of pair message bits
- Unweighted/weighted greedy algorithm

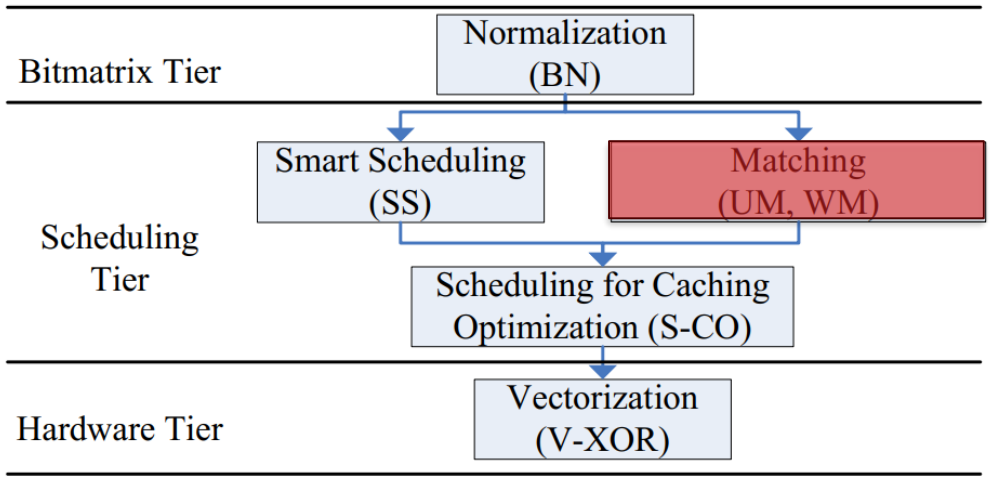


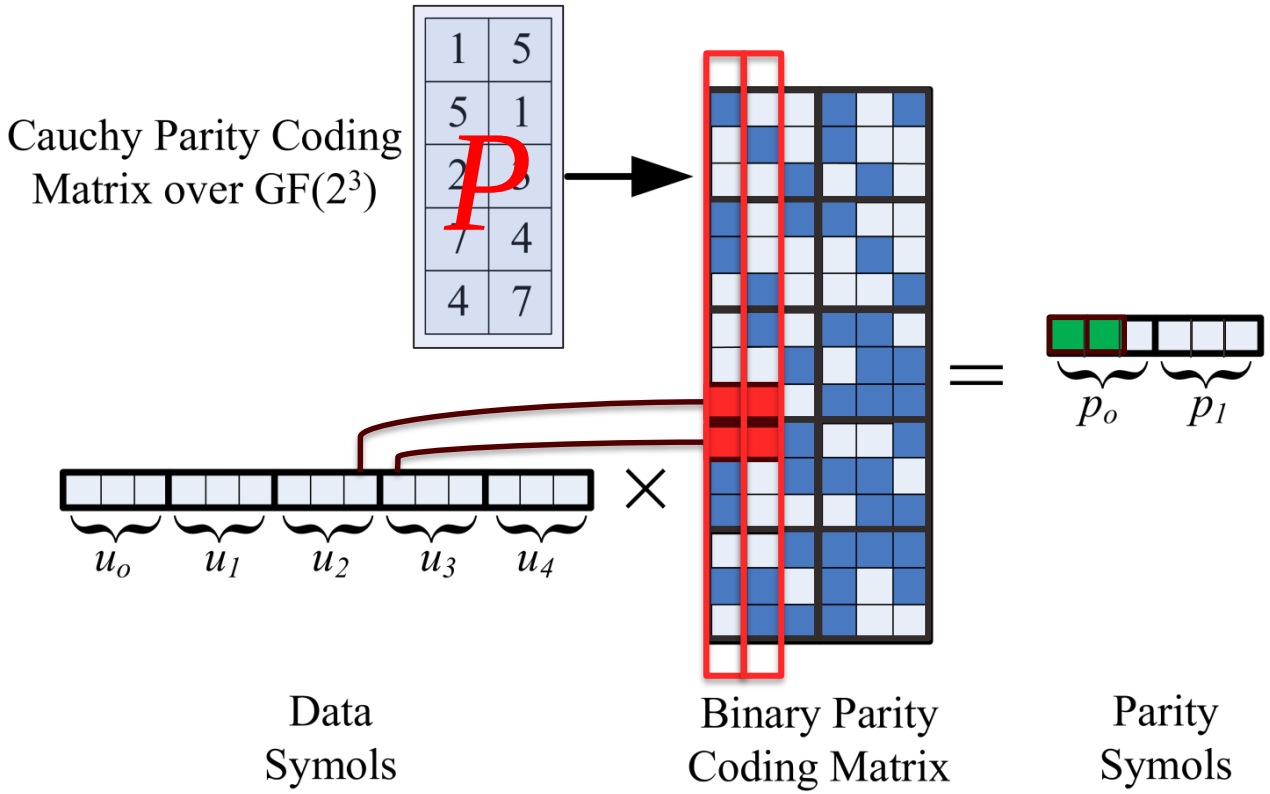


Cheng Huang, Jin Li, and Minghua Chen. On optimizing XOR-based codes for fault-tolerant storage applications. In Proceedings of Information Theory Workshop, pages 218–223, 2007.

Matching (UM, WM)

- Common XORs of pair message bits
- Unweighted/weighted greedy algorithm

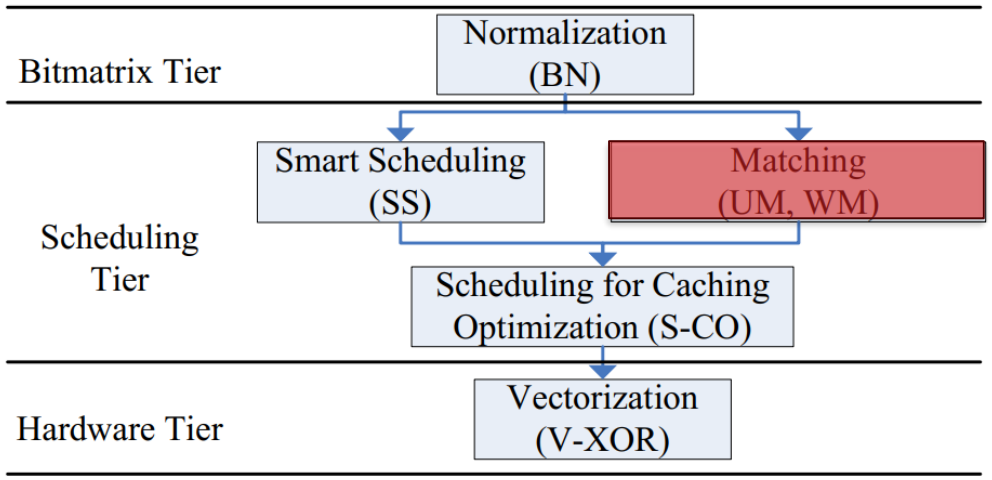


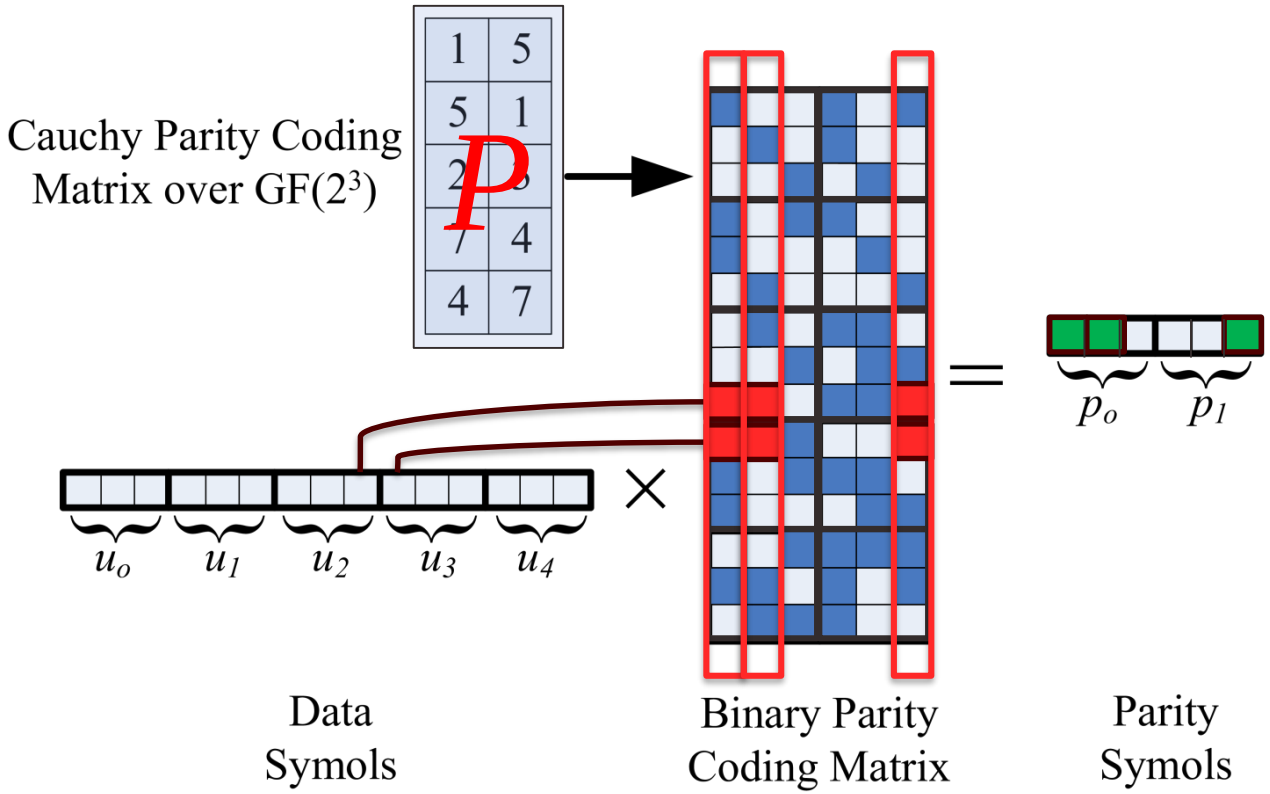


Cheng Huang, Jin Li, and Minghua Chen. On optimizing XOR-based codes for fault-tolerant storage applications. In Proceedings of Information Theory Workshop, pages 218–223, 2007.

Matching (UM, WM)

- Common XORs of pair message bits
- Unweighted/weighted greedy algorithm





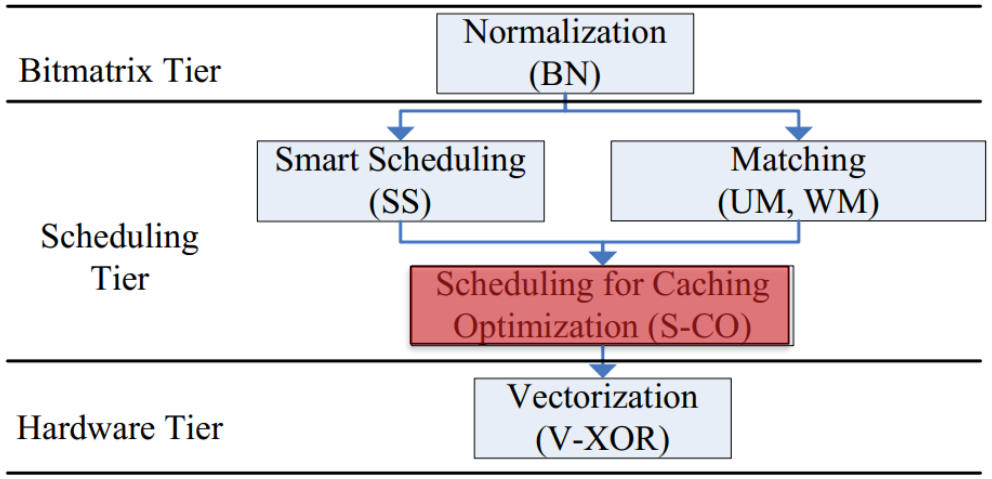
Cheng Huang, Jin Li, and Minghua Chen. On optimizing XOR-based codes for fault-tolerant storage applications. In Proceedings of Information Theory Workshop, pages 218–223, 2007.

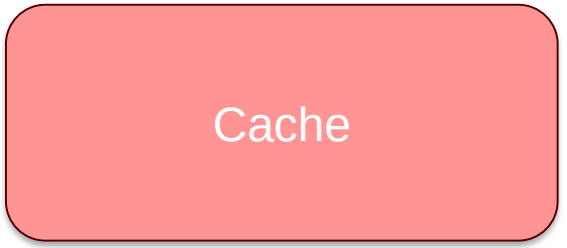
Scheduling - Cache Optimization (S-CO)

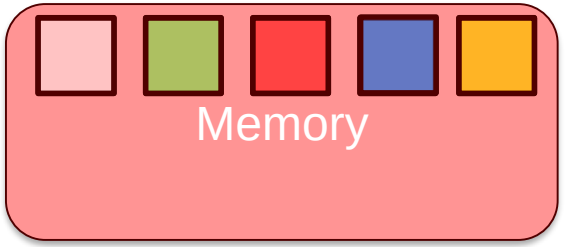
- Reduce cache miss penalty

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4$$

$$p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5$$







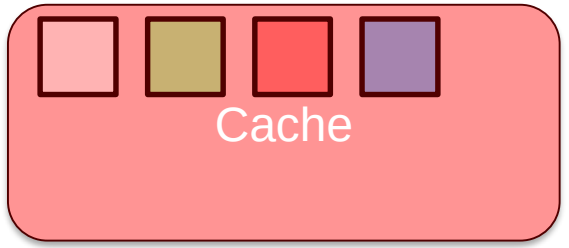
Jianqiang Luo, Mochan Shrestha, Lihao Xu, and James S. Plank. Efficient encoding schedules for XOR-based erasure codes. IEEE Transactions on Computers, 63(9):2259–2272, 2014.

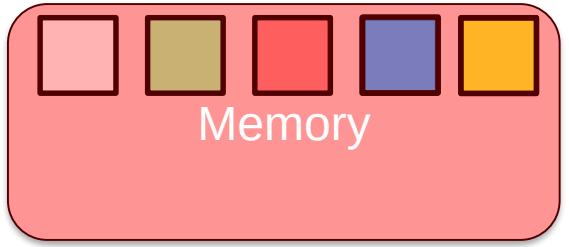
Scheduling - Cache Optimization (S-CO)

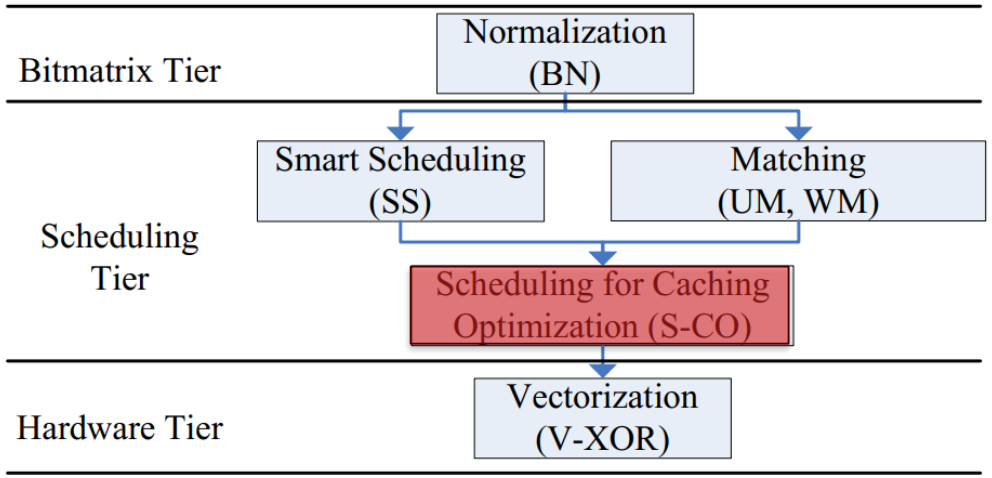
- Reduce cache miss penalty

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4$$

$$p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5$$







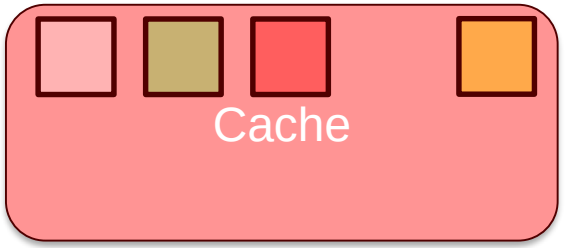
Jianqiang Luo, Mochan Shrestha, Lihao Xu, and James S. Plank. Efficient encoding schedules for XOR-based erasure codes. IEEE Transactions on Computers, 63(9):2259–2272, 2014.

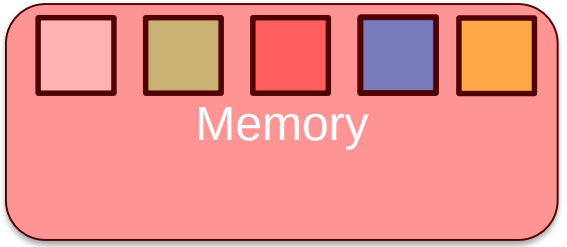
Scheduling - Cache Optimization (S-CO)

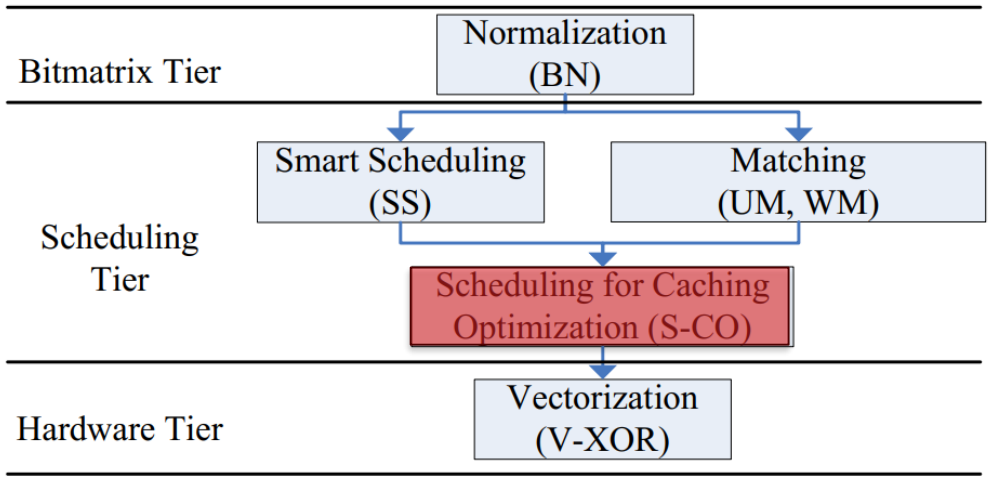
- Reduce cache miss penalty

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4$$

$$p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5$$







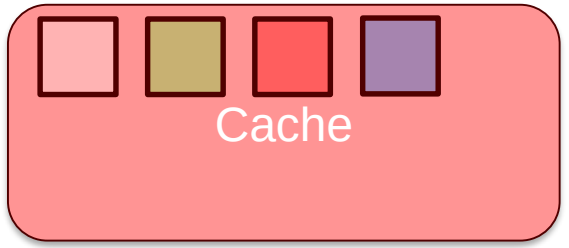
Jianqiang Luo, Mochan Shrestha, Lihao Xu, and James S. Plank. Efficient encoding schedules for XOR-based erasure codes. IEEE Transactions on Computers, 63(9):2259–2272, 2014.

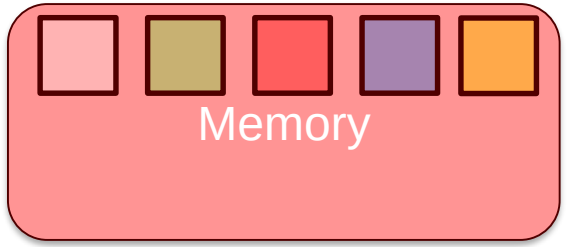
Scheduling - Cache Optimization (S-CO)

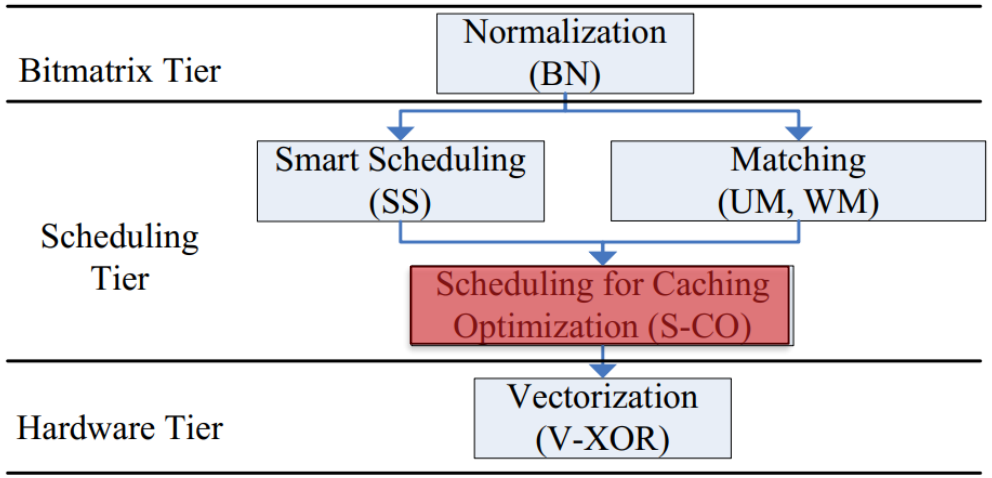
- Reduce cache miss penalty

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4$$

$$p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5$$



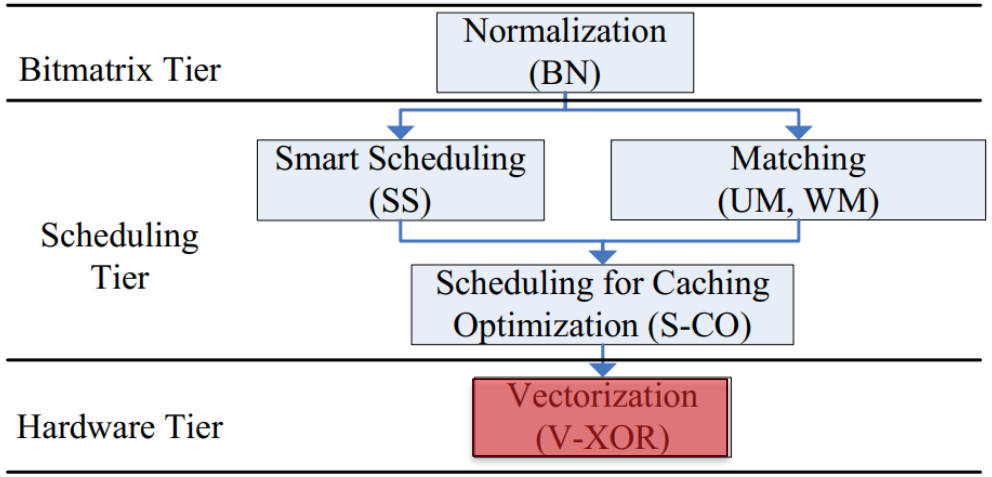




Jianqiang Luo, Mochan Shrestha, Lihao Xu, and James S. Plank. Efficient encoding schedules for XOR-based erasure codes. IEEE Transactions on Computers, 63(9):2259–2272, 2014.

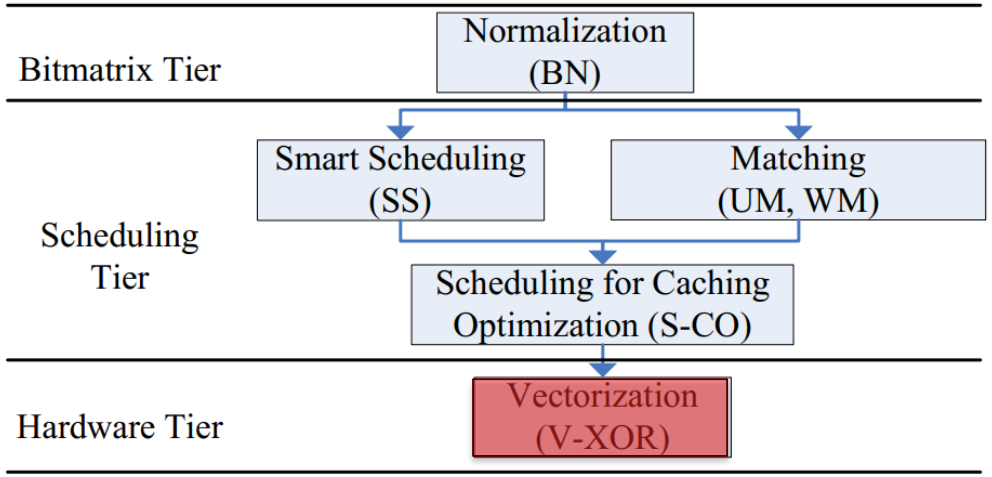
Vectorization

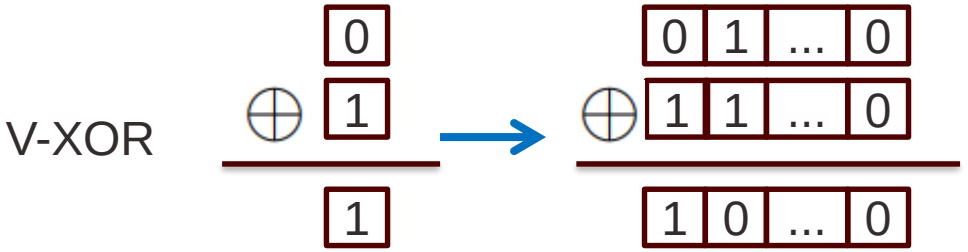
- Using SIMD ISA perform multiple bits operation
  - SSE: 128 bits
  - AVX2: 256 bits
  - AVX512: 512 bits



Vectorization

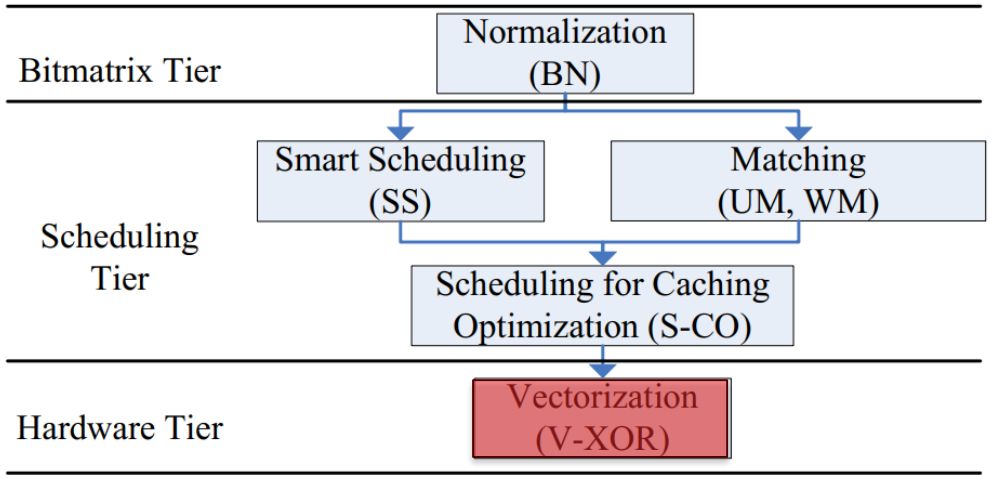
- Using SIMD ISA perform multiple bits operation
  - SSE: 128 bits
  - AVX2: 256 bits
  - AVX512: 512 bits

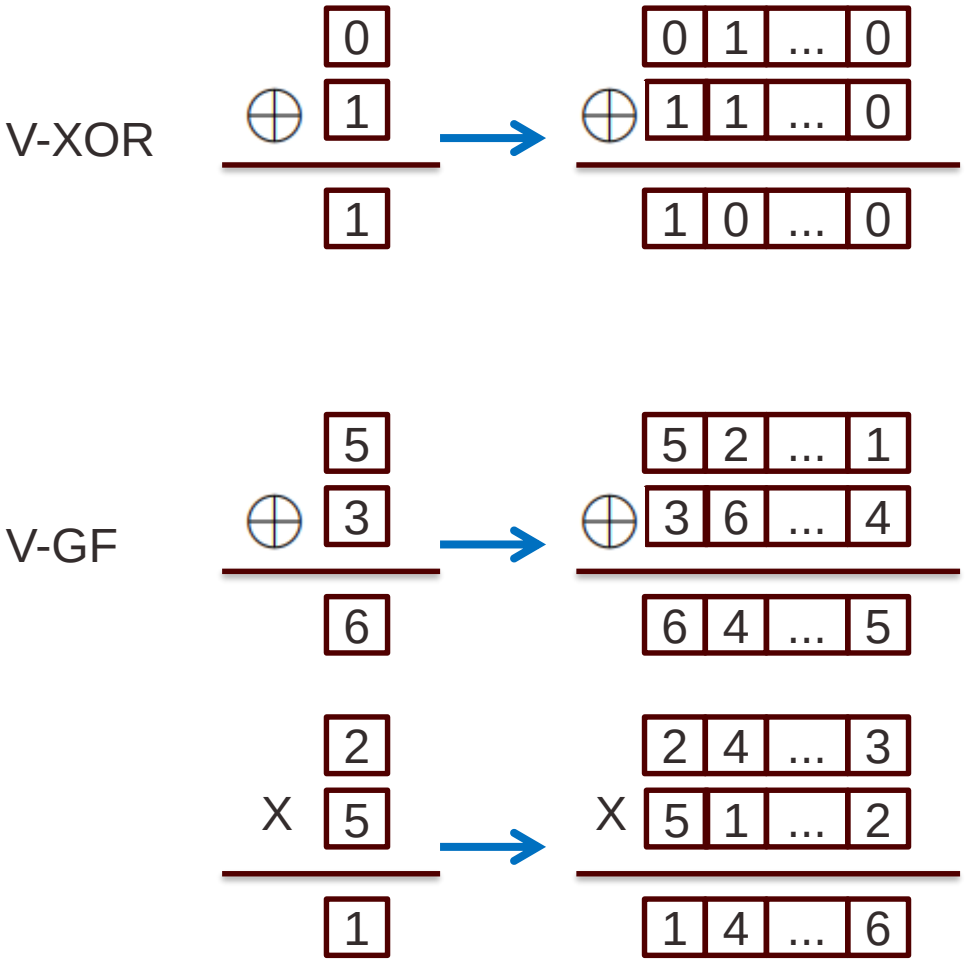




Vectorization

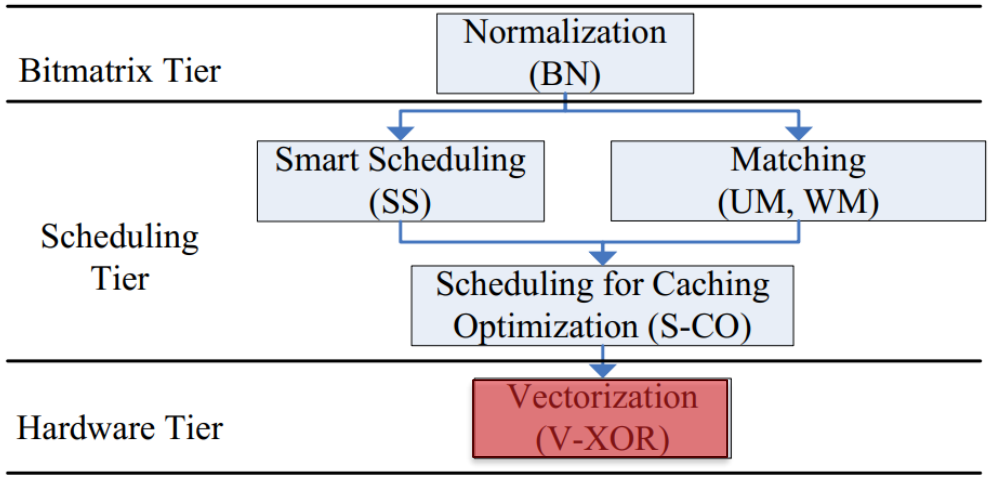
- Using SIMD ISA perform multiple bits operation
  - SSE: 128 bits
  - AVX2: 256 bits
  - AVX512: 512 bits

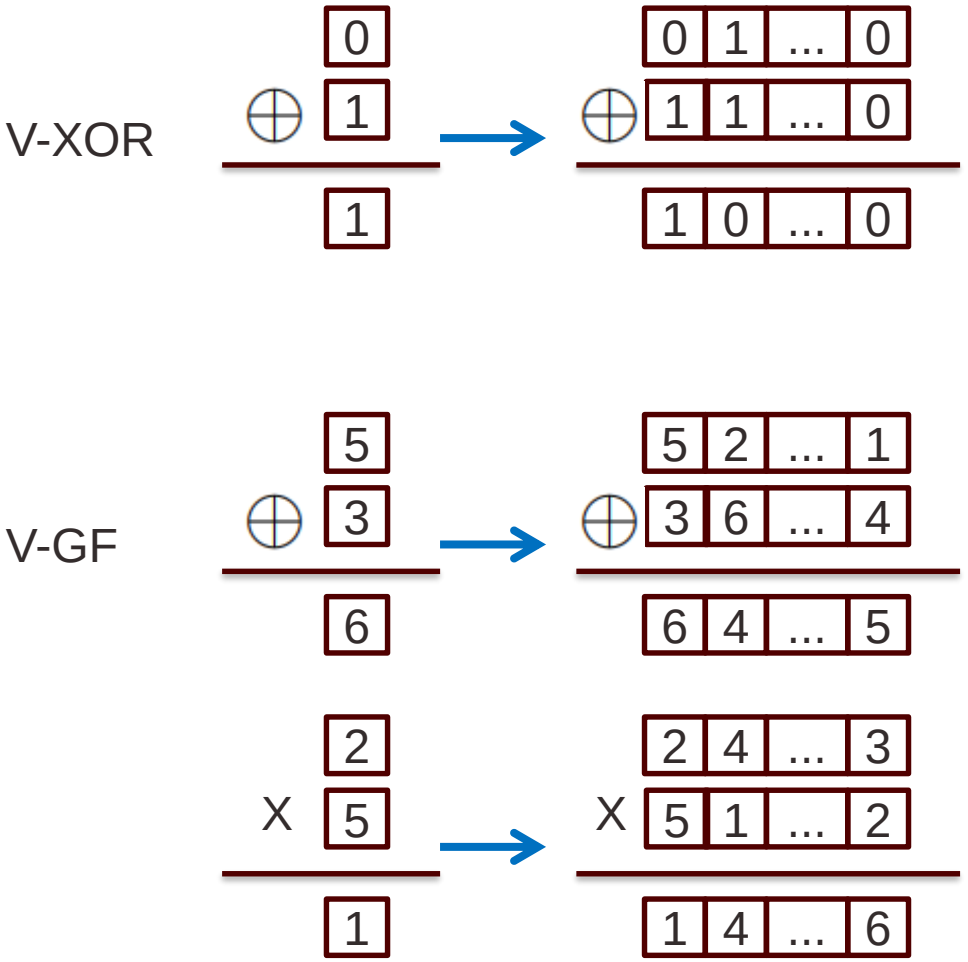




Vectorization

- Using SIMD ISA perform multiple bits operation
  - SSE: 128 bits
  - AVX2: 256 bits
  - AVX512: 512 bits

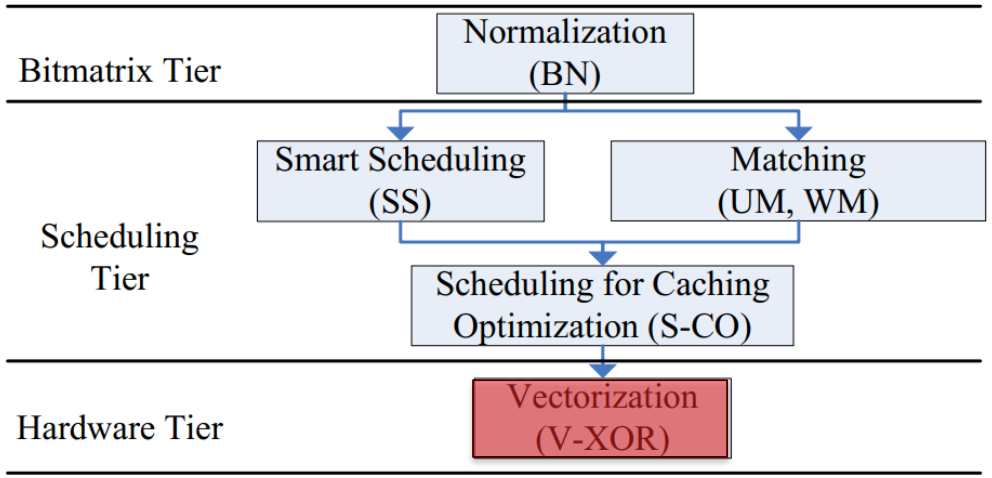


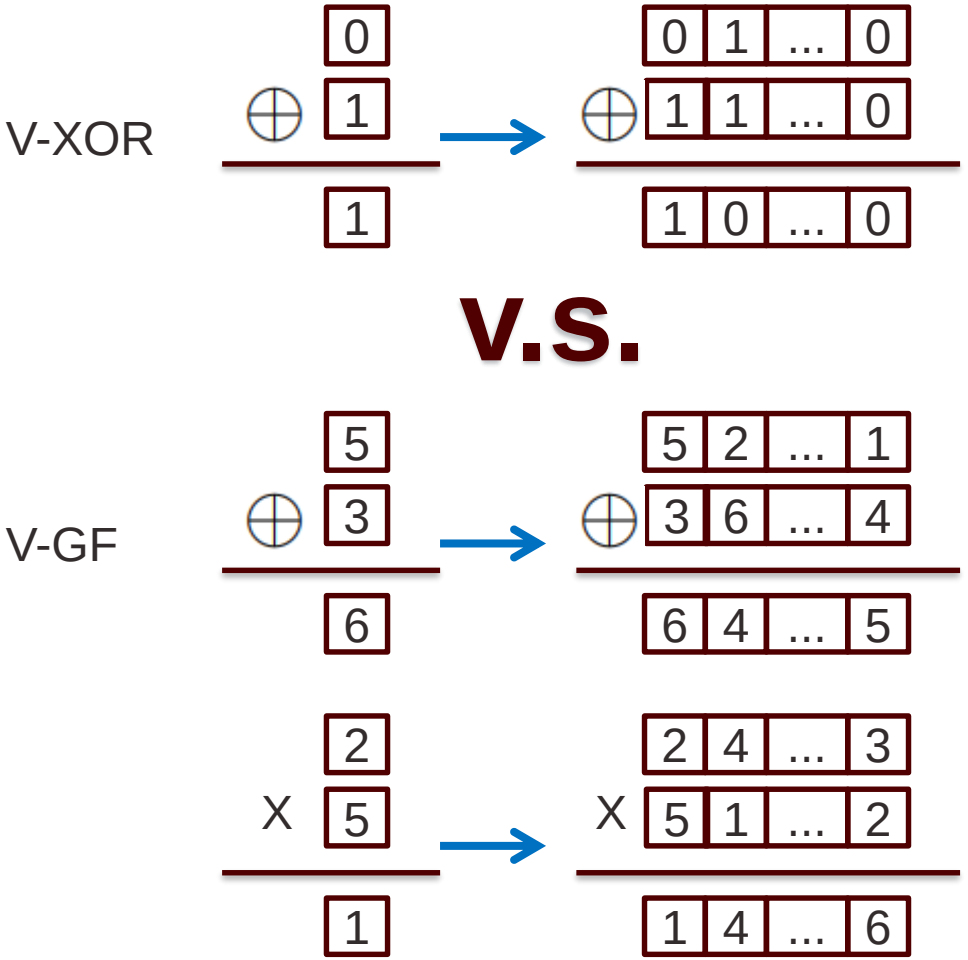


Jerasure 2.0

Vectorization

- Using SIMD ISA perform multiple bits operation
  - SSE: 128 bits
  - AVX2: 256 bits
  - AVX512: 512 bits

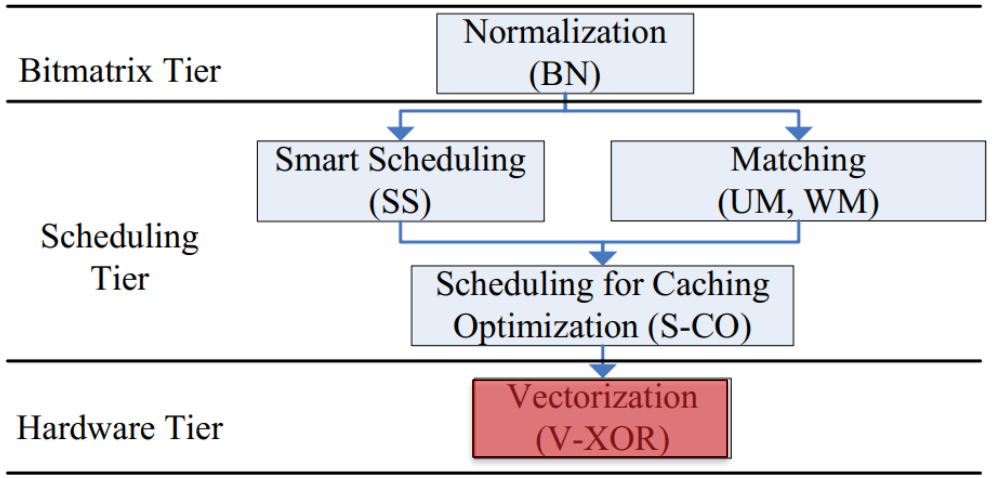


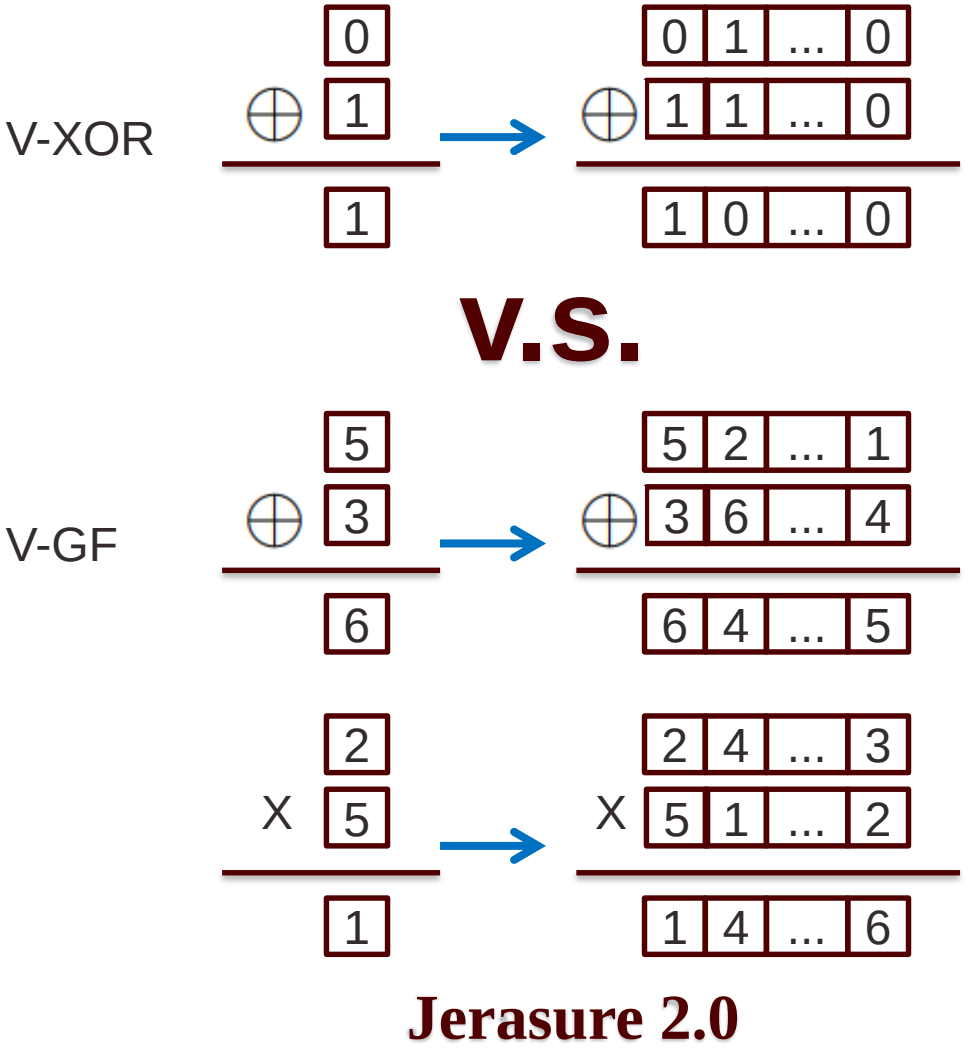


Jerasure 2.0

Vectorization

- Using SIMD ISA perform multiple bits operation
  - SSE: 128 bits
  - AVX2: 256 bits
  - AVX512: 512 bits





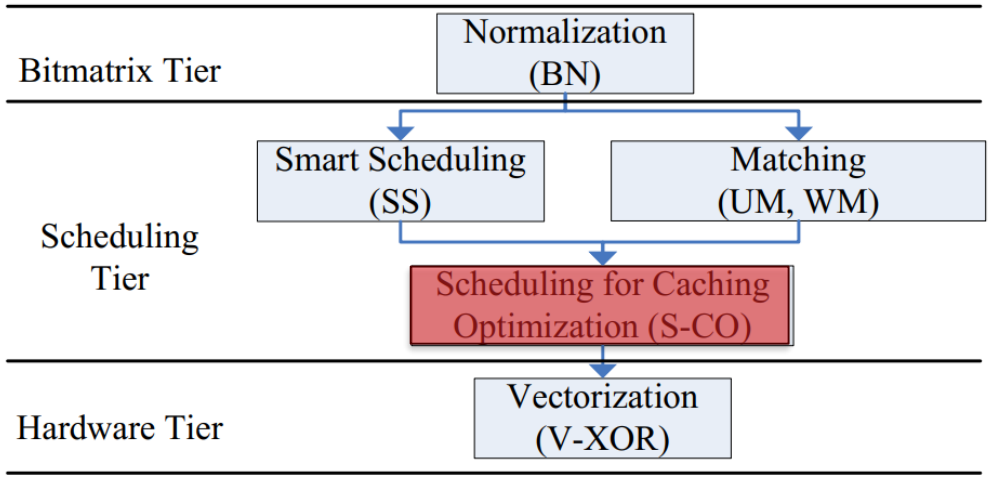
Jerasure 2.0

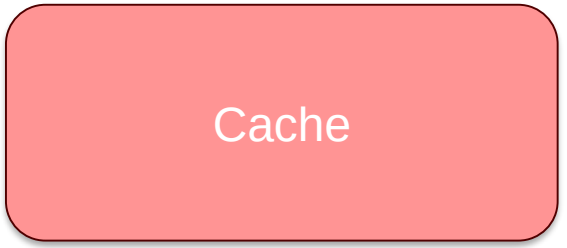
Scheduling - Cache Optimization (S-CO)

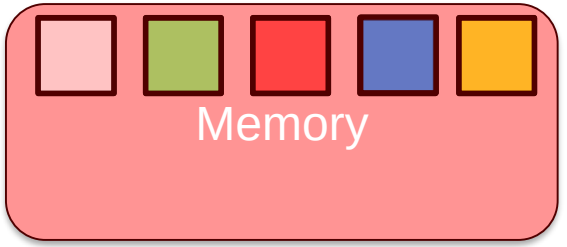
- Reduce cache miss penalty

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4$$

$$p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5$$







Jianqiang Luo, Mochan Shrestha, Lihao Xu, and James S. Plank. Efficient encoding schedules for XOR-based erasure codes. IEEE Transactions on Computers, 63(9):2259–2272, 2014.

Scheduling - Cache Optimization (S-CO)

- Reduce cache miss penalty

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4$$

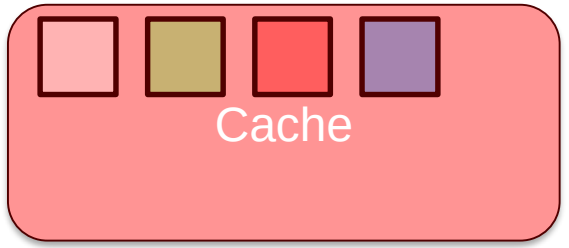
$$p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5$$

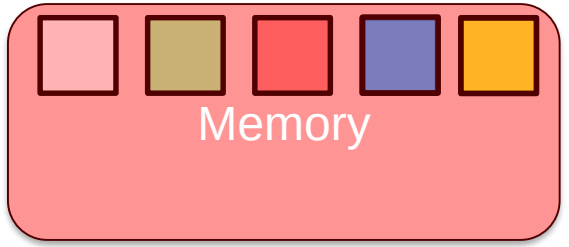
$u_0 \rightarrow p_0, p_1, p_{2\ldots}$

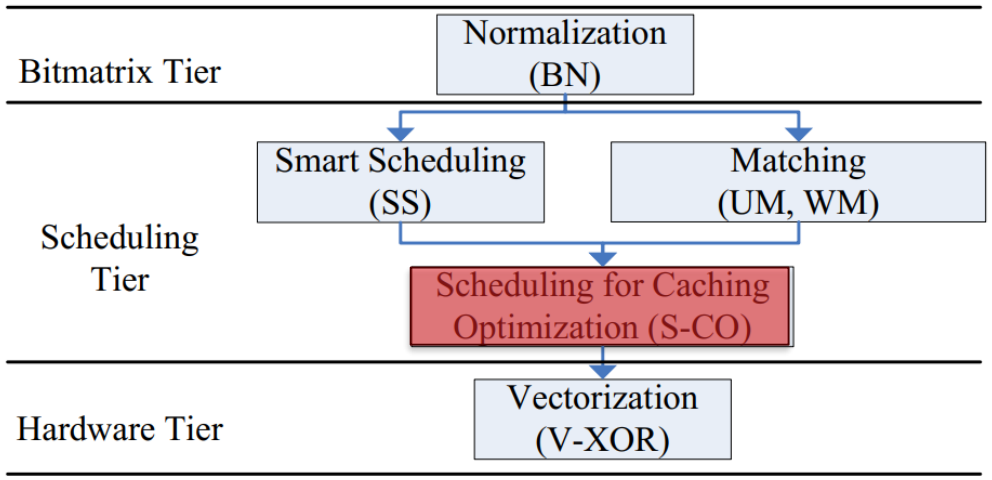
$u_1 \rightarrow p_2, p_3, p_{5\ldots}$

$u_2 \rightarrow p_0, p_1, p_{4\ldots}$

$u_3 \rightarrow p_2, p_6, p_{8\ldots}$







Jianqiang Luo, Mochan Shrestha, Lihao Xu, and James S. Plank. Efficient encoding schedules for XOR-based erasure codes. IEEE Transactions on Computers, 63(9):2259–2272, 2014.

Question to Answer

- Most effective technique(s)?
- Utilize together?
- Components to be optimized?

Individual Techniques

Table 1: Performance improvements by individual techniques

(n,k,w)	The number of XOR's in the baseline code	Reduction in the number of XOR's				Throughput increase	
		BN	SS	UM	WM	S-CO	V-XOR
(8,6,4)	104	42.31%	17.31%	31.73%	31.73%	2.14%	98.44%
(8,6,8)	362	53.31%	33.70%	34.53%	34.81%	0.52%	126.36%
(9,6,4)	152	32.89%	19.74%	32.89%	32.89%	3.54%	90.48%
(9,6,8)	549	44.63%	29.14%	38.25%	38.25%	4.51%	152.87%
(10,6,4)	200	27.50%	22.00%	36.00%	36.00%	0.60%	91.30%
(10,6,8)	736	40.90%	28.80%	38.59%	38.59%	1.23%	130.71%
(12,8,4)	256	23.44%	23.44%	35.94%	35.94%	9.21%	118.26%
(12,8,8)	1028	36.38%	24.81%	39.79%	39.79%	0.10%	156.88%
(16,10,4)	496	18.95%	21.77%	37.70%	37.70%	8.28%	138.93%
(16,10,8)	1920	30.16%	21.98%	40.89%	40.89%	5.00%	179.00%
Average over all tested cases		35.05%	24.27%	36.63%	36.66%	4.81%	130.4%

Individual Techniques

XOR-based Vectorization

Table 1: Performance improvements by individual techniques

(n,k,w)	The number of XOR's in the baseline code	Reduction in the number of XOR's				Throughput increase	
		BN	SS	UM	WM	S-CO	V-XOR
(8,6,4)	104	42.31%	17.31%	31.73%	31.73%	2.14%	98.44%
(8,6,8)	362	53.31%	33.70%	34.53%	34.81%	0.52%	126.36%
(9,6,4)	152	32.89%	19.74%	32.89%	32.89%	3.54%	90.48%
(9,6,8)	549	44.63%	29.14%	38.25%	38.25%	4.51%	152.87%
(10,6,4)	200	27.50%	22.00%	36.00%	36.00%	0.60%	91.30%
(10,6,8)	736	40.90%	28.80%	38.59%	38.59%	1.23%	130.71%
(12,8,4)	256	23.44%	23.44%	35.94%	35.94%	9.21%	118.26%
(12,8,8)	1028	36.38%	24.81%	39.79%	39.79%	0.10%	156.88%
(16,10,4)	496	18.95%	21.77%	37.70%	37.70%	8.28%	138.93%
(16,10,8)	1920	30.16%	21.98%	40.89%	40.89%	5.00%	179.00%
Average over all tested cases		35.05%	24.27%	36.63%	36.66%	4.81%	130.4%

Individual Techniques

XOR-based Vectorization

Table 1: Performance improvements by individual techniques

(n,k,w)	The number of XOR's in the baseline code	Reduction in the number of XOR's				Throughput increase	
		BN	SS	UM	WM	S-CO	V-XOR
(8,6,4)	104	42.31%	17.31%	31.73%	31.73%	2.14%	98.44%
(8,6,8)	362	53.31%	33.70%	34.53%	34.81%	0.52%	126.36%
(9,6,4)	152	32.89%	19.74%	32.89%	32.89%	3.54%	90.48%
(9,6,8)	549	44.63%	29.14%	38.25%	38.25%	4.51%	152.87%
(10,6,4)	200	27.50%	22.00%	36.00%	36.00%	0.60%	91.30%
(10,6,8)	736	40.90%	28.80%	38.59%	38.59%	1.23%	130.71%
(12,8,4)	256	23.44%	23.44%	35.94%	35.94%	9.21%	118.26%
(12,8,8)	1028	36.38%	24.81%	39.79%	39.79%	0.10%	156.88%
(16,10,4)	496	18.95%	21.77%	37.70%	37.70%	8.28%	138.93%
(16,10,8)	1920	30.16%	21.98%	40.89%	40.89%	5.00%	179.00%
Average over all tested cases		35.05%	24.27%	36.63%	36.66%	4.81%	130.4%

Bitmatrix Normalization

Unweighted Matching

Weighted Matching

Individual Techniques

XOR-based Vectorization

Table 1: Performance improvements by individual techniques

(n,k,w)	The number of XOR's in the baseline code	Reduction in the number of XOR's				Throughput increase	
		BN	SS	UM	WM	S-CO	V-XOR
(8,6,4)	104	42.31%	17.31%	31.73%	31.73%	2.14%	98.44%
(8,6,8)	362	53.31%	33.70%	34.53%	34.81%	0.52%	126.36%
(9,6,4)	152	32.89%	19.74%	32.89%	32.89%	3.54%	90.48%
(9,6,8)	549	44.63%	29.14%	38.25%	38.25%	4.51%	152.87%
(10,6,4)	200	27.50%	22.00%	36.00%	36.00%	0.60%	91.30%
(10,6,8)	736	40.90%	28.80%	38.59%	38.59%	1.23%	130.71%
(12,8,4)	256	23.44%	23.44%	35.94%	35.94%	9.21%	118.26%
(12,8,8)	1028	36.38%	24.81%	39.79%	39.79%	0.10%	156.88%
(16,10,4)	496	18.95%	21.77%	37.70%	37.70%	8.28%	138.93%
(16,10,8)	1920	30.16%	21.98%	40.89%	40.89%	5.00%	179.00%
Average over all tested cases		35.05%	24.27%	36.63%	36.66%	4.81%	130.4%

Bitmatrix Normalization

Smart Scheduling

Unweighted Matching

Weighted Matching

Individual Techniques

Cache Optimization

XOR-based Vectorization

Table 1: Performance improvements by individual techniques

(n,k,w)	The number of XOR's in the baseline code	Reduction in the number of XOR's				Throughput increase	
		BN	SS	UM	WM	S-CO	V-XOR
(8,6,4)	104	42.31%	17.31%	31.73%	31.73%	2.14%	98.44%
(8,6,8)	362	53.31%	33.70%	34.53%	34.81%	0.52%	126.36%
(9,6,4)	152	32.89%	19.74%	32.89%	32.89%	3.54%	90.48%
(9,6,8)	549	44.63%	29.14%	38.25%	38.25%	4.51%	152.87%
(10,6,4)	200	27.50%	22.00%	36.00%	36.00%	0.60%	91.30%
(10,6,8)	736	40.90%	28.80%	38.59%	38.59%	1.23%	130.71%
(12,8,4)	256	23.44%	23.44%	35.94%	35.94%	9.21%	118.26%
(12,8,8)	1028	36.38%	24.81%	39.79%	39.79%	0.10%	156.88%
(16,10,4)	496	18.95%	21.77%	37.70%	37.70%	8.28%	138.93%
(16,10,8)	1920	30.16%	21.98%	40.89%	40.89%	5.00%	179.00%
Average over all tested cases		35.05%	24.27%	36.63%	36.66%	4.81%	130.4%

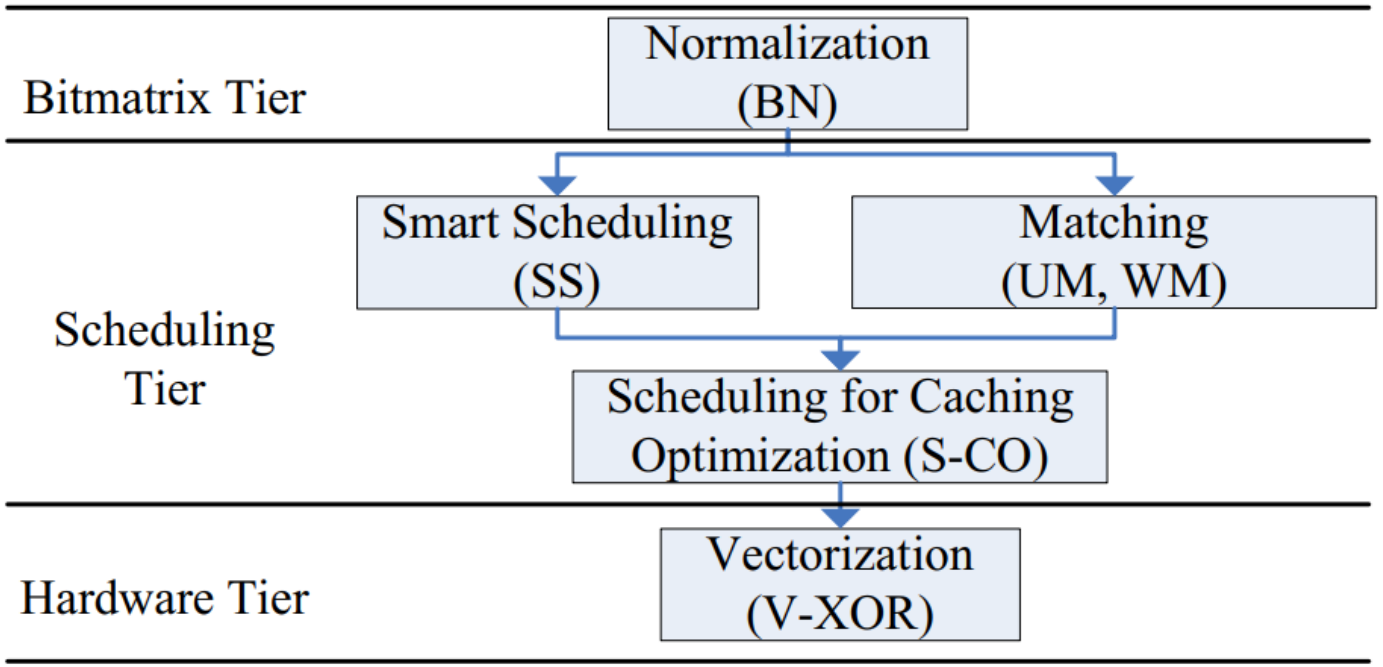
Bitmatrix Normalization

Smart Scheduling

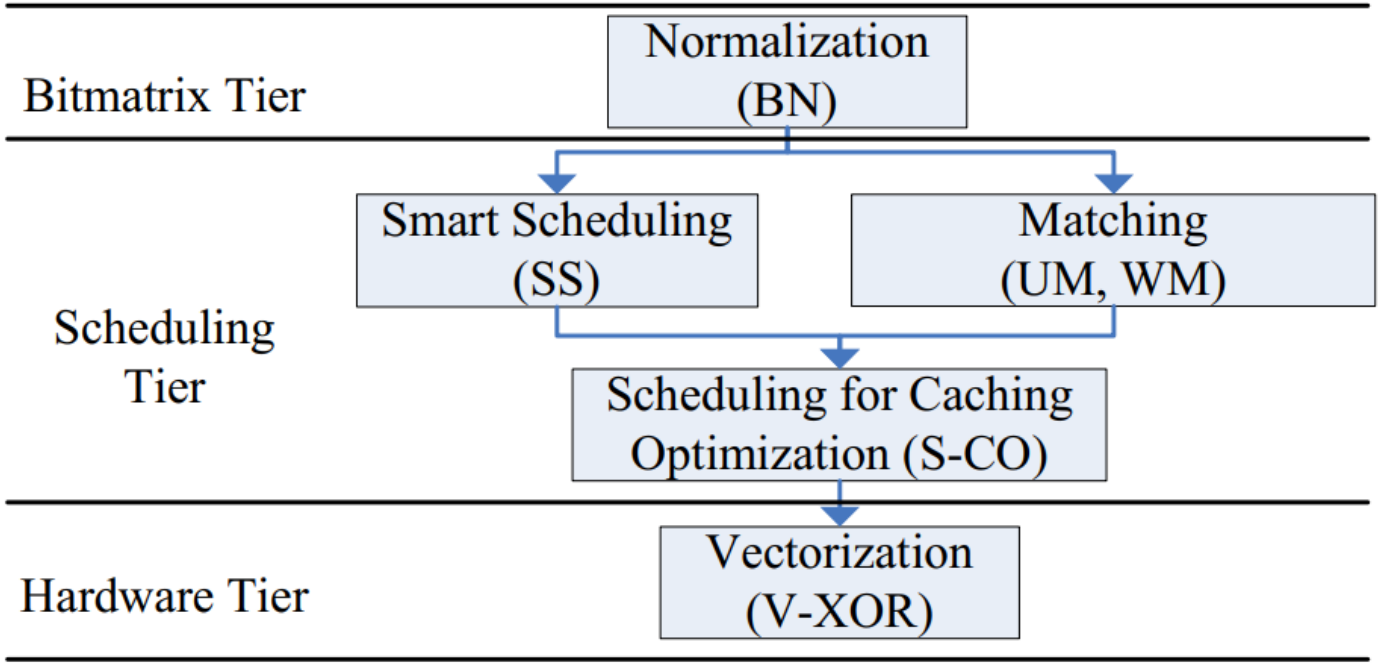
Unweighted Matching

Weighted Matching

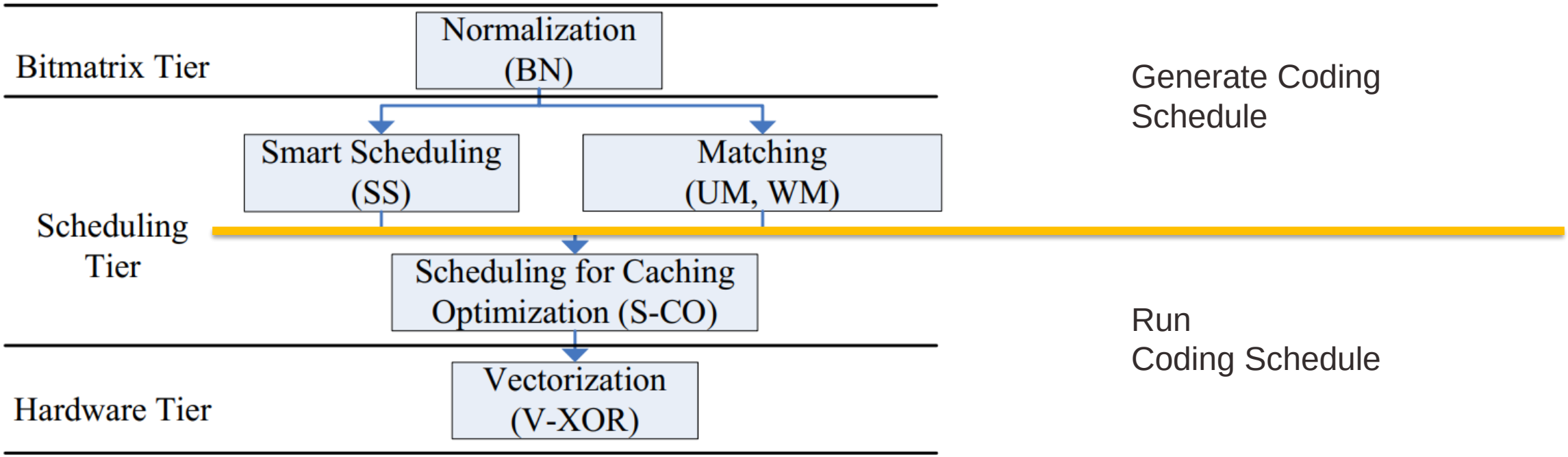
Optimization Tiers



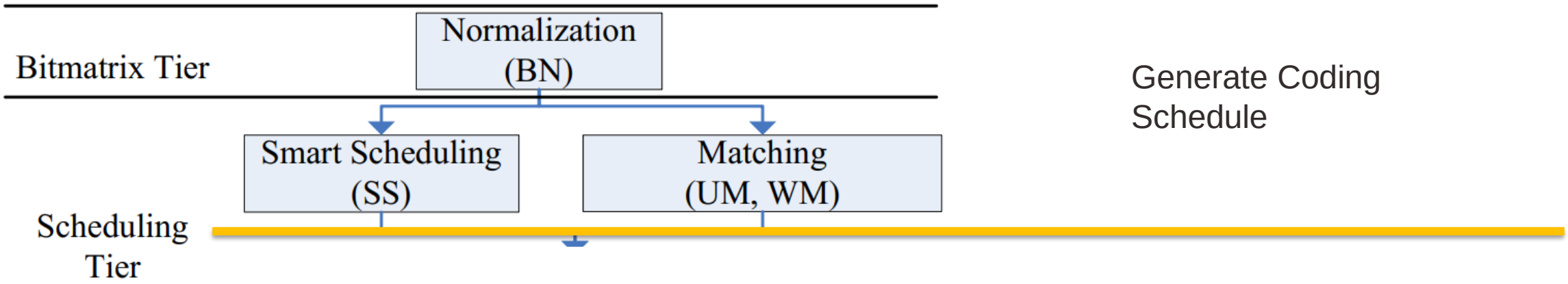
Optimization Tiers



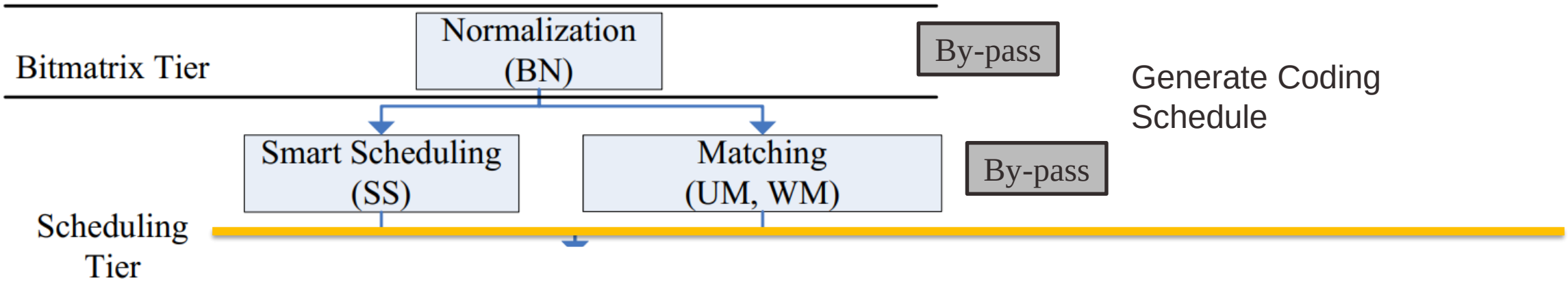
Optimization Tiers



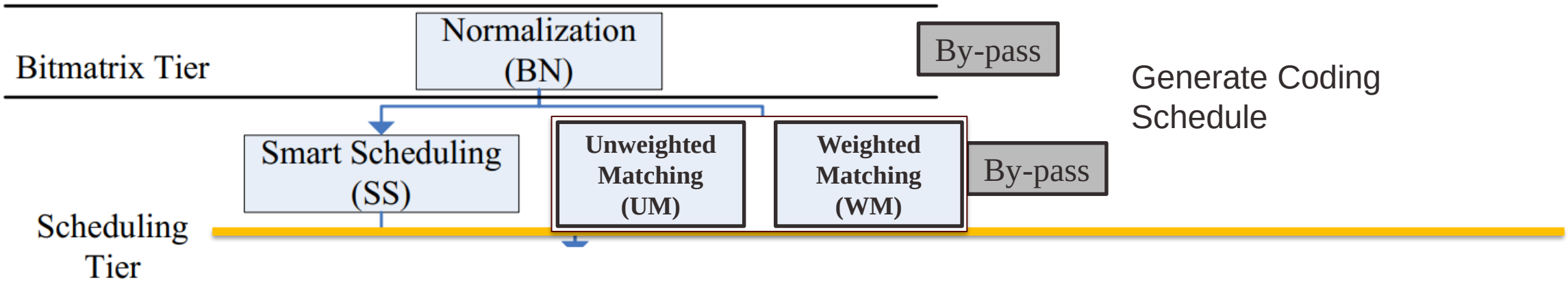
Optimization Tiers



Optimization Tiers



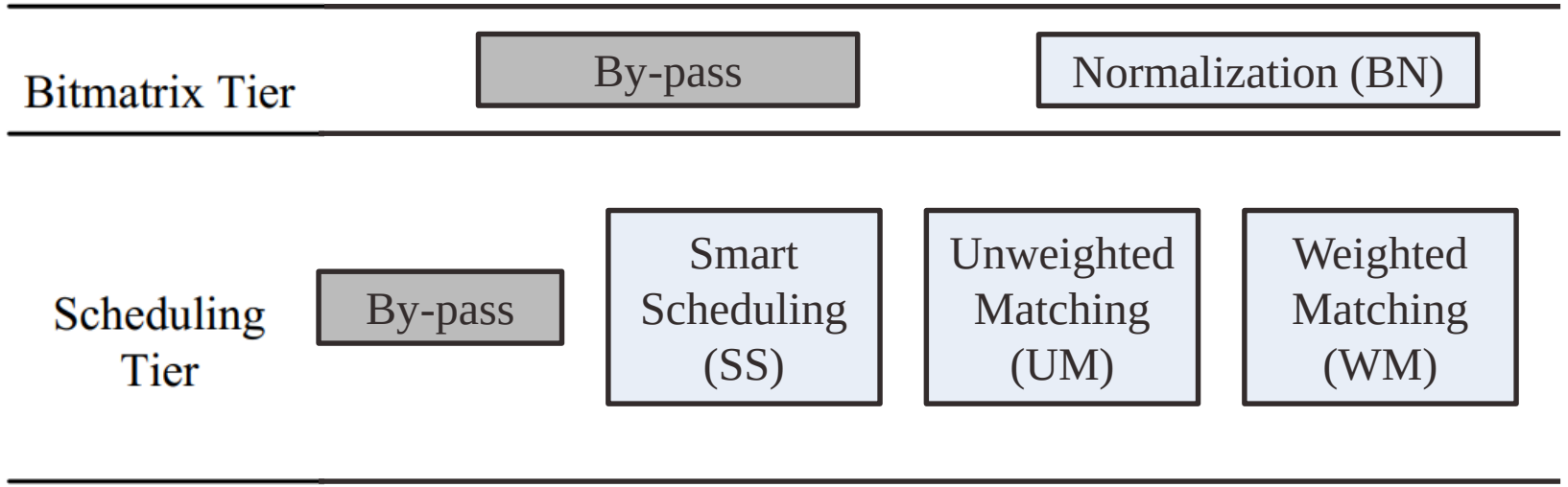
Optimization Tiers



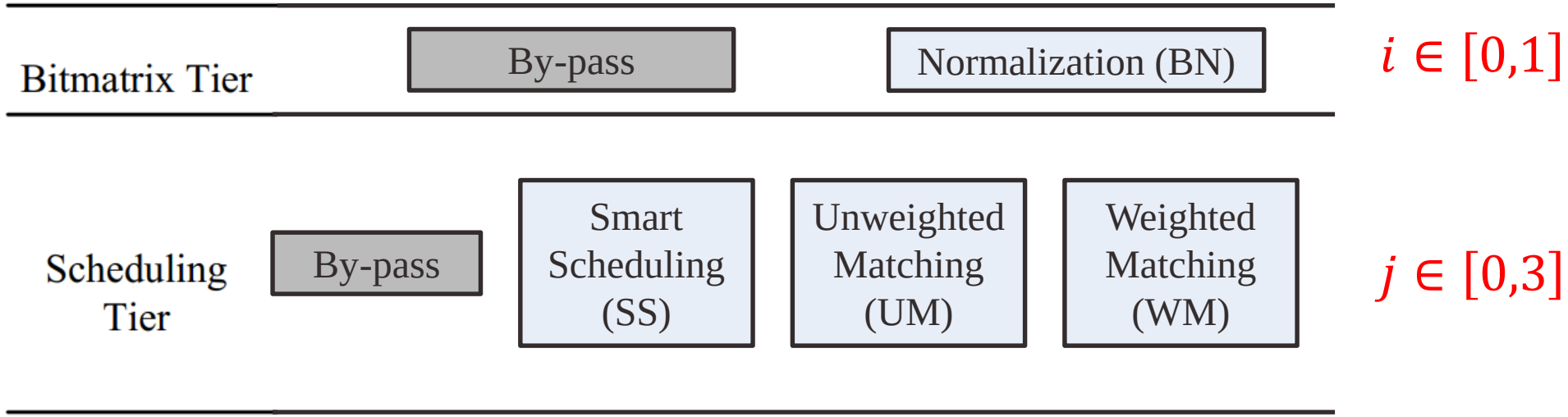
Question to Answer

- Most effective technique(s)?
- Utilize together?
- Components to be optimized?

Combinations (i,j)-strategy



Combinations (i,j)-strategy

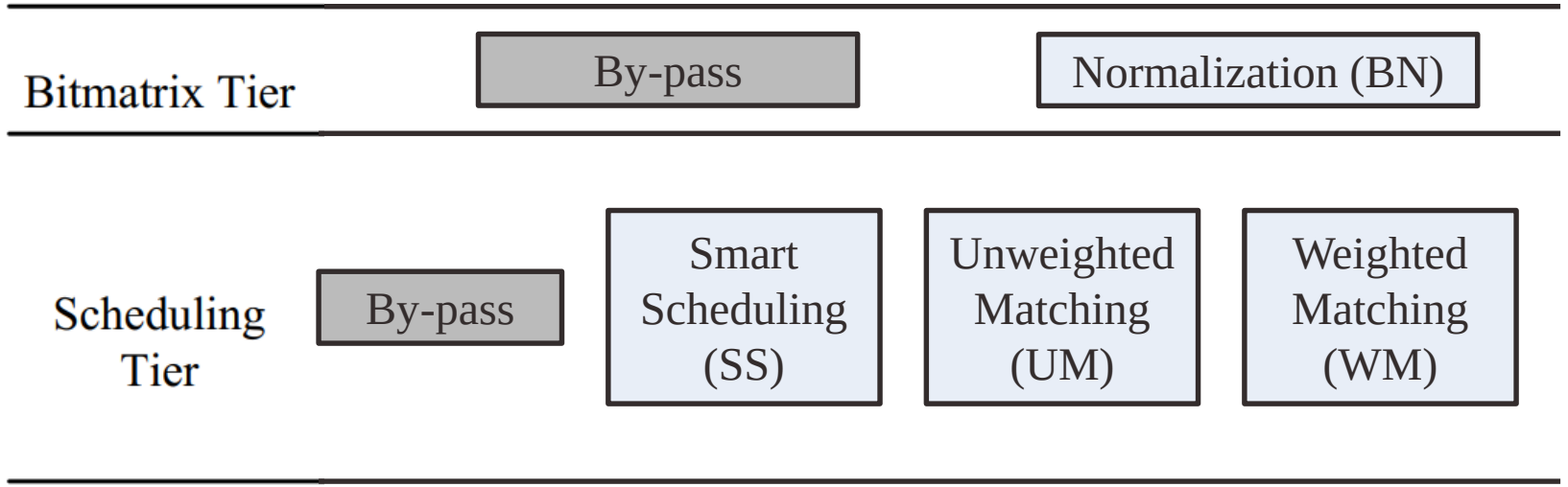


$i \in [0,1]$

$j \in [0,3]$

Combinations (i,j)-strategy

Example: strategy-(1,1)

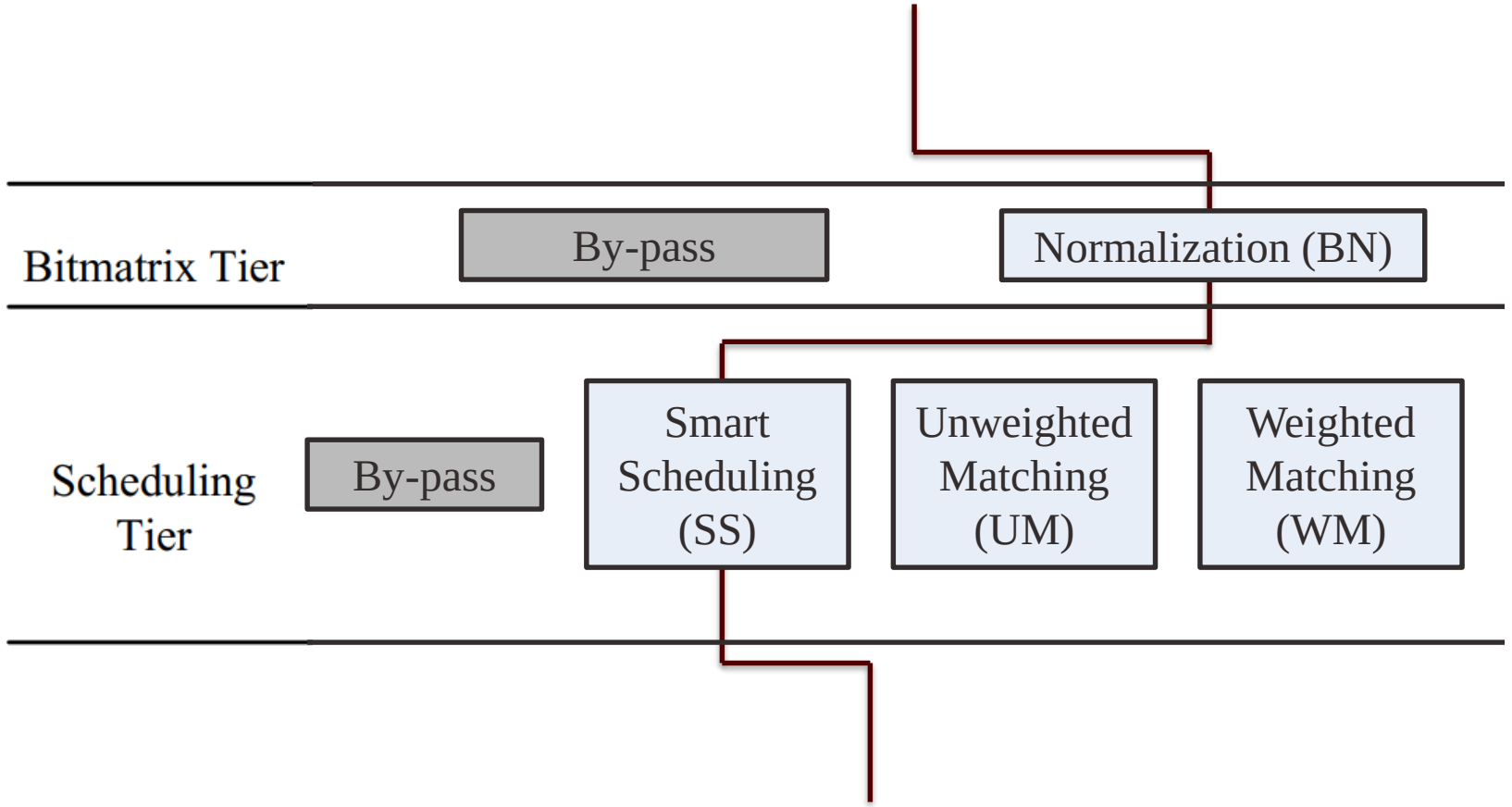


$i \in [0,1]$

$j \in [0,3]$

Combinations (i,j)-strategy

Example: strategy-(1,1)



$i \in [0,1]$

$j \in [0,3]$

Contents

- Motivation
- Background and Review

- Evaluating Individual Techniques
- Find the Best Strategy under Optimized Bitmatrix
- Proposed Coding Procedure and Evaluation

- Conclusion

Question to Answer

- Most effective technique(s)?
- Utilize together?
- Components to be optimized?

Choice of Cauchy Matrix

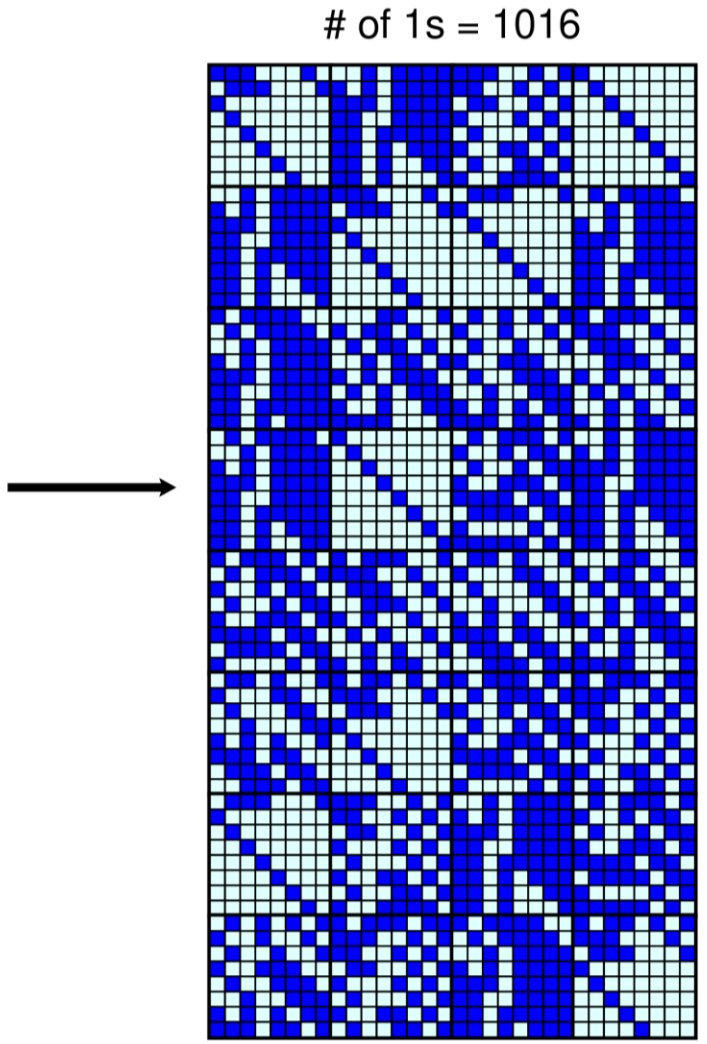
- Different X,Y array yields different Cauchy Matrix

Choice of Cauchy Matrix

- Different *X*,*Y* array yields different Cauchy Matrix

X = 6 1 7 3
Y = 2 5 10 0 8 9 4 11

71	244	167	1
244	71	142	122
61	152	170	157
122	1	186	244
93	157	150	152
150	173	93	221
142	167	244	186
170	221	61	173

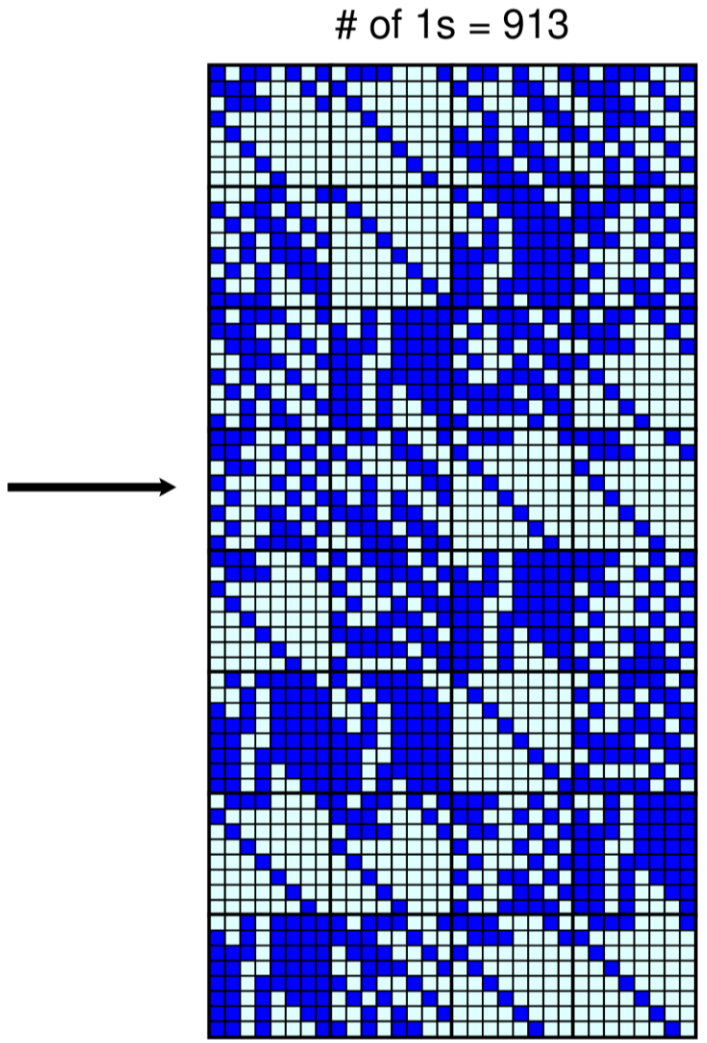


Choice of Cauchy Matrix

- Different *X*,*Y* array yields different Cauchy Matrix

X = 1 11 6 0
Y = 9 10 8 4 5 7 3 2

173	142	150	157
152	1	61	221
157	244	93	173
167	150	142	71
71	93	244	167
122	61	1	186
142	173	167	244
244	157	71	142

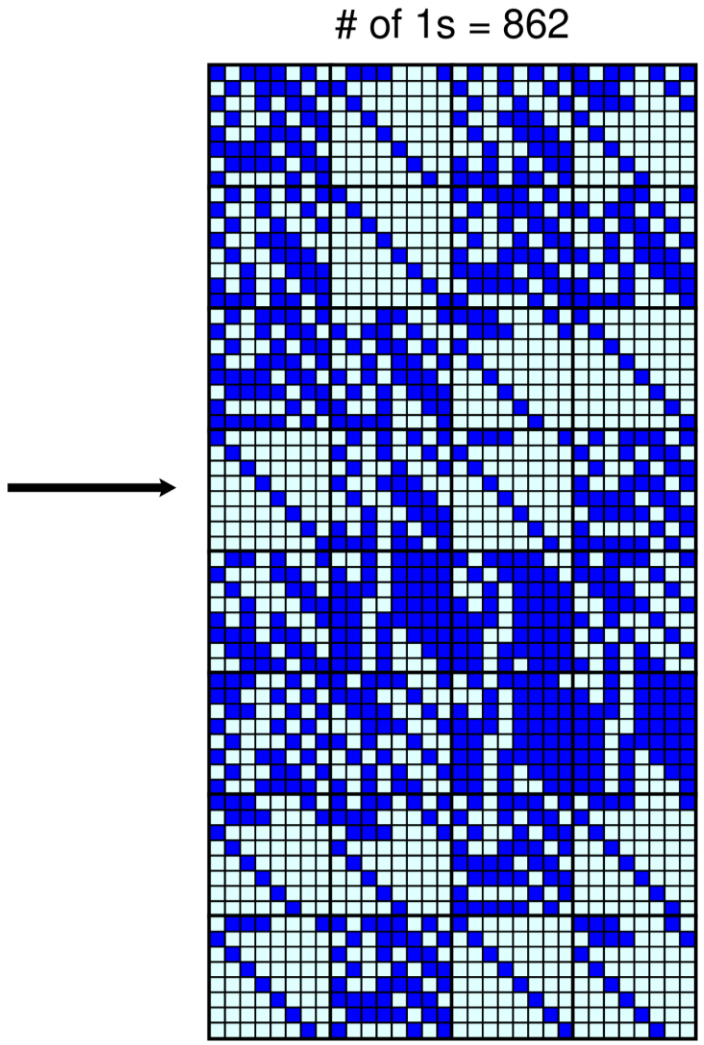


Choice of Cauchy Matrix

- Different *X*,*Y* array yields different Cauchy Matrix

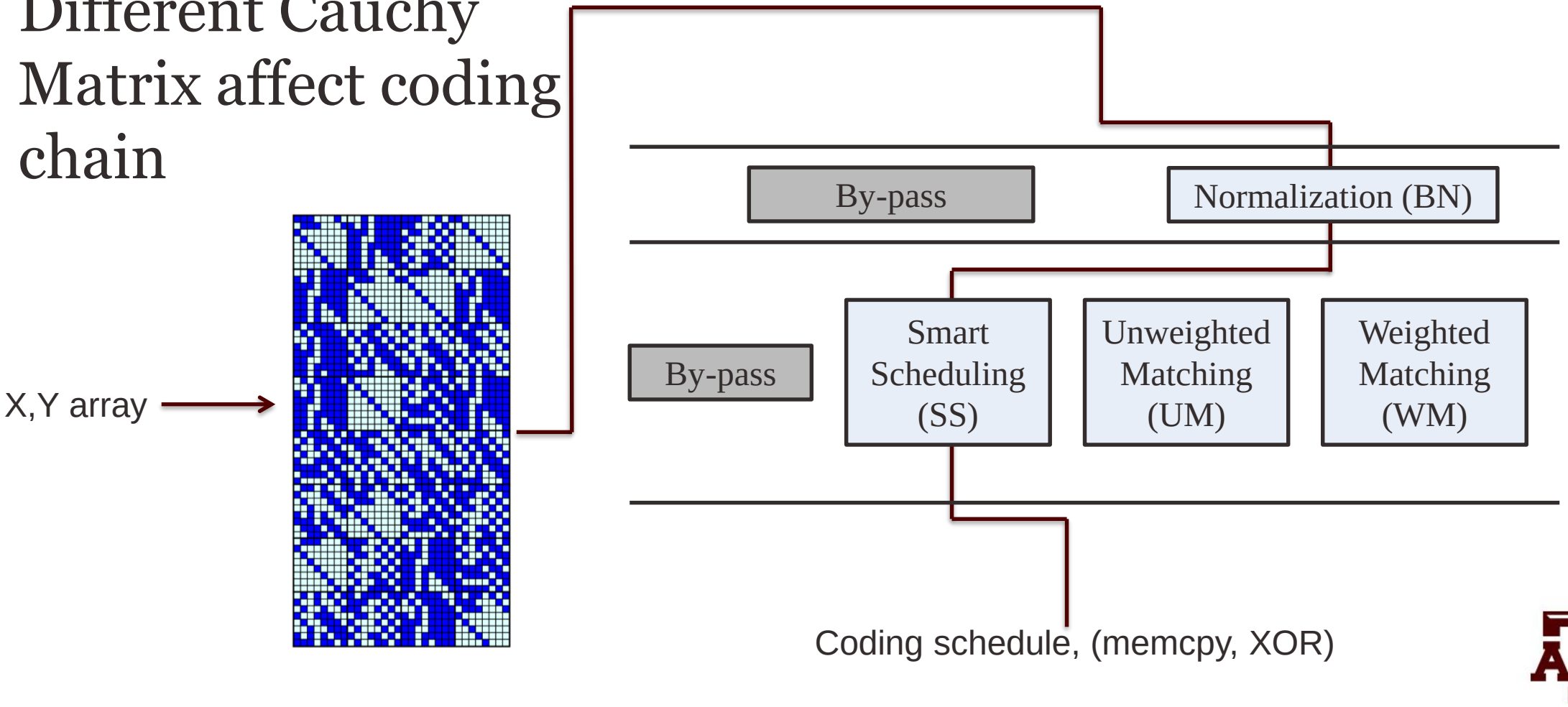
X = 5 9 6 3
Y = 11 8 2 4 10 0 1 7

93	142	170	173
170	1	93	152
186	152	71	1
1	170	142	186
150	244	61	157
167	157	122	244
71	173	186	142
142	93	1	71



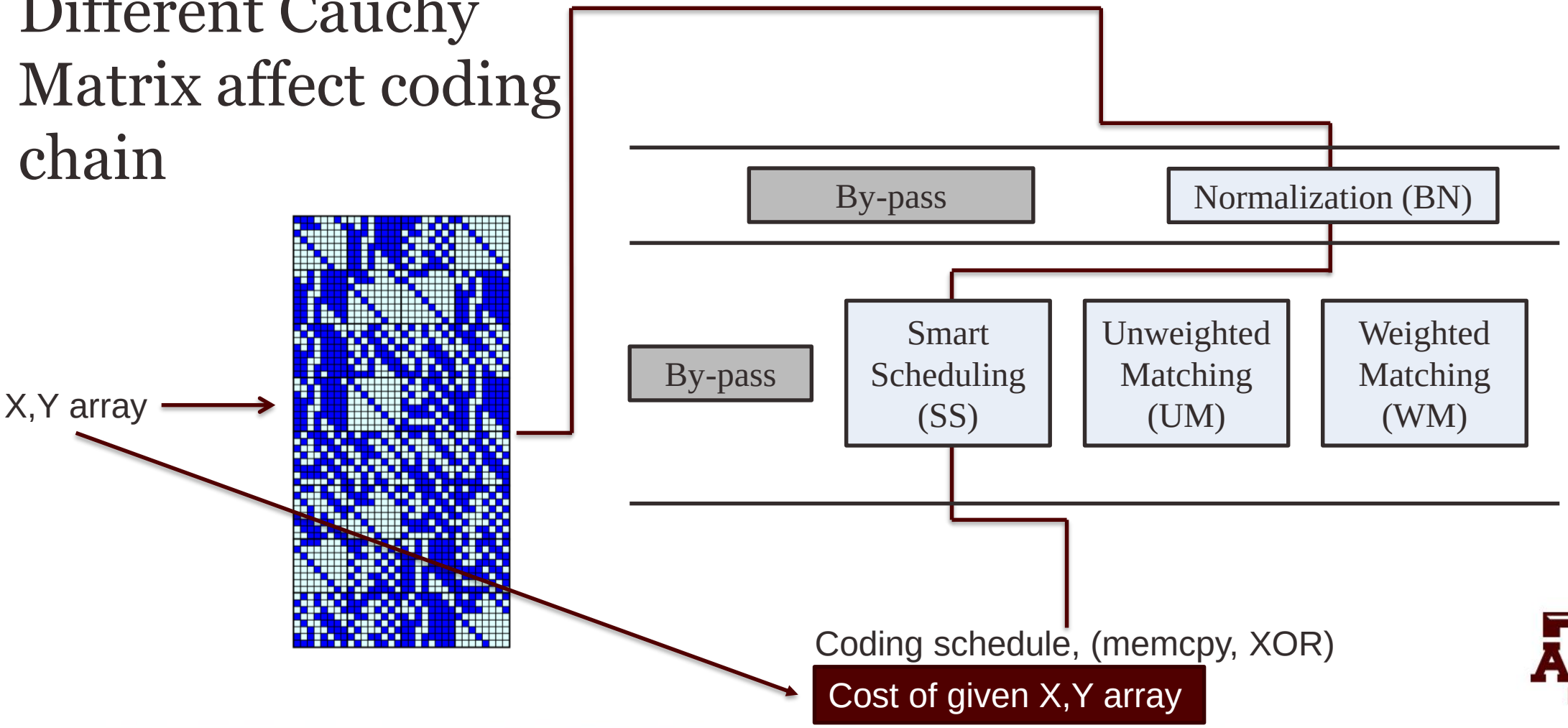
Choice of Cauchy Matrix

- Different Cauchy Matrix affect coding chain



Choice of Cauchy Matrix

- Different Cauchy Matrix affect coding chain



Bitmatrix Optimization

- Individual bitmatrix optimization for all (i,j)-strategies
- Simulated Annealing and Genetic Algorithm
- # of operations as cost

[no_opt by SA by GA], # of ops in schedule

(i,j)	$(n,k,w)=(8,6,4)$			$(n,k,w)=(9,6,4)$			$(n,k,w)=(10,6,4)$			$(n,k,w)=(12,8,4)$			$(n,k,w)=(16,10,4)$		
(0,0)	112	**77**	**77**	164	122	**117**	216	173	**162**	272	**232**	**232**	520	**462**	464
(0,1)	94	70	**68**	134	102	**100**	172	137	**134**	212	190	**188**	412	**368**	**368**
(0,2)	90	72	**68**	127	103	**101**	164	134	**132**	204	186	**180**	376	**344**	345
(0,3)	90	72	**68**	127	102	**101**	164	**132**	**132**	204	184	**182**	376	**343**	345
(1,0)	68	**58**	**58**	114	99	**98**	161	142	**141**	212	**198**	**198**	426	**419**	**419**
(1,1)	64	**58**	**58**	99	90	**89**	138	**120**	**120**	189	174	**171**	365	**352**	**352**
(1,2)	64	58	**57**	98	**87**	**87**	133	**118**	**118**	176	165	**164**	326	317	**316**
(1,3)	64	58	**57**	98	90	**87**	132	**118**	**118**	175	**164**	**164**	326	**316**	**316**

(i,j)	$(n,k,w)=(8,6,8)$			$(n,k,w)=(9,6,8)$			$(n,k,w)=(10,6,8)$			$(n,k,w)=(12,8,8)$			$(n,k,w)=(16,10,8)$		
(0,0)	378	291	**247**	573	467	**425**	768	611	**582**	1060	880	**841**	1968	1685	**1681**
(0,1)	256	234	**225**	413	393	**349**	556	518	**479**	805	740	**684**	1546	1432	**1374**
(0,2)	286	227	**217**	408	346	**321**	532	474	**450**	726	636	**591**	1304	1180	**1170**
(0,3)	286	216	**197**	408	357	**329**	532	460	**450**	726	636	**627**	1304	1180	**1170**
(1,0)	185	151	**142**	328	286	**262**	467	434	**419**	686	613	**563**	1389	1293	**1246**
(1,1)	164	155	**133**	285	270	**230**	411	383	**371**	593	557	**544**	1264	1184	**1133**
(1,2)	167	143	**134**	272	244	**225**	377	343	**335**	520	487	**462**	998	951	**922**
(1,3)	167	144	**130**	273	249	**233**	377	**337**	**337**	520	473	**467**	995	941	**932**

[no_opt by SA by GA], # of ops in schedule

(i,j)	$(n,k,w)=(8,6,4)$			$(n,k,w)=(9,6,4)$			$(n,k,w)=(10,6,4)$			$(n,k,w)=(12,8,4)$			$(n,k,w)=(16,10,4)$		
(0,0)	112	**77**	**77**	164	122	**117**	216	173	**162**	272	**232**	**232**	520	**462**	464
(0,1)	94	70	**68**	134	102	**100**	172	137	**134**	212	190	**188**	412	**368**	**368**
(0,2)	90	72	**68**	127	103	**101**	164	134	**132**	204	186	**180**	376	**344**	345
(0,3)	90	72	**68**	127	102	**101**	164	**132**	**132**	204	184	**182**	376	**343**	345
(1,0)	68	**58**	**58**	114	99	**98**	161	142	**141**	212	**198**	**198**	426	**419**	**419**
(1,1)	64	**58**	**58**	99	90	**89**	138	**120**	**120**	189	174	**171**	365	**352**	**352**
(1,2)	64	58	**57**	98	**87**	**87**	133	**118**	**118**	176	165	**164**	326	317	**316**
(1,3)	64	58	**57**	98	90	**87**	132	**118**	**118**	175	**164**	**164**	326	**316**	**316**

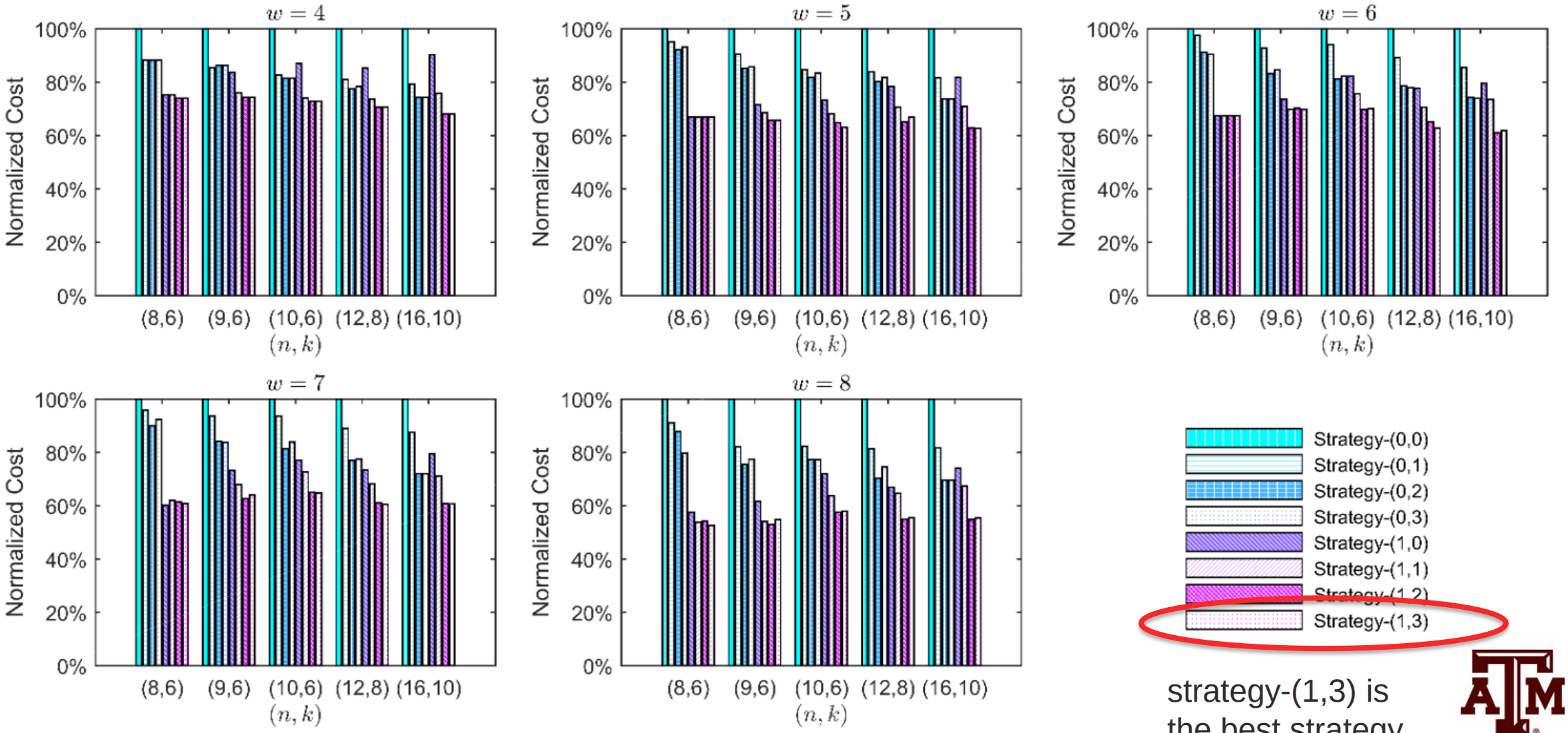
(i,j)	$(n,k,w)=(8,6,8)$			$(n,k,w)=(9,6,8)$			$(n,k,w)=(10,6,8)$			$(n,k,w)=(12,8,8)$			$(n,k,w)=(16,10,8)$		
(0,0)	378	291	**247**	573	467	**425**	768	611	**582**	1060	880	**841**	1968	1685	**1681**
(0,1)	256	234	**225**	413	393	**349**	556	518	**479**	805	740	**684**	1546	1432	**1374**
(0,2)	286	227	**217**	408	346	**321**	532	474	**450**	726	636	**591**	1304	1180	**1170**
(0,3)	286	216	**197**	408	357	**329**	532	460	**450**	726	636	**627**	1304	1180	**1170**
(1,0)	185	151	**142**	328	286	**262**	467	434	**419**	686	613	**563**	1389	1293	**1246**
(1,1)	164	155	**133**	285	270	**230**	411	383	**371**	593	557	**544**	1264	1184	**1133**
(1,2)	167	143	**134**	272	244	**225**	377	343	**335**	520	487	**462**	998	951	**922**
(1,3)	167	144	**130**	273	249	**233**	377	**337**	**337**	520	473	**467**	995	941	**932**

[no_opt by SA by GA], # of ops in schedule

(i,j)	$(n,k,w)=(8,6,4)$			$(n,k,w)=(9,6,4)$			$(n,k,w)=(10,6,4)$			$(n,k,w)=(12,8,4)$			$(n,k,w)=(16,10,4)$		
(0,0)	112	**77**	**77**	164	122	**117**	216	173	**162**	272	**232**	**232**	520	**462**	464
(0,1)	94	70	**68**	134	102	**100**	172	137	**134**	212	190	**188**	412	**368**	**368**
(0,2)	90	72	**68**	127	103	**101**	164	134	**132**	204	186	**180**	376	**344**	345
(0,3)	90	72	**68**	127	102	**101**	164	**132**	**132**	204	184	**182**	376	**343**	345
(1,0)	68	**58**	**58**	114	99	**98**	161	142	**141**	212	**198**	**198**	426	**419**	**419**
(1,1)	64	**58**	**58**	99	90	**89**	138	**120**	**120**	189	174	**171**	365	**352**	**352**
(1,2)	64	58	**57**	98	**87**	**87**	133	**118**	**118**	176	165	**164**	326	317	**316**
(1,3)	64	58	**57**	98	90	**87**	132	**118**	**118**	175	**164**	**164**	326	**316**	**316**

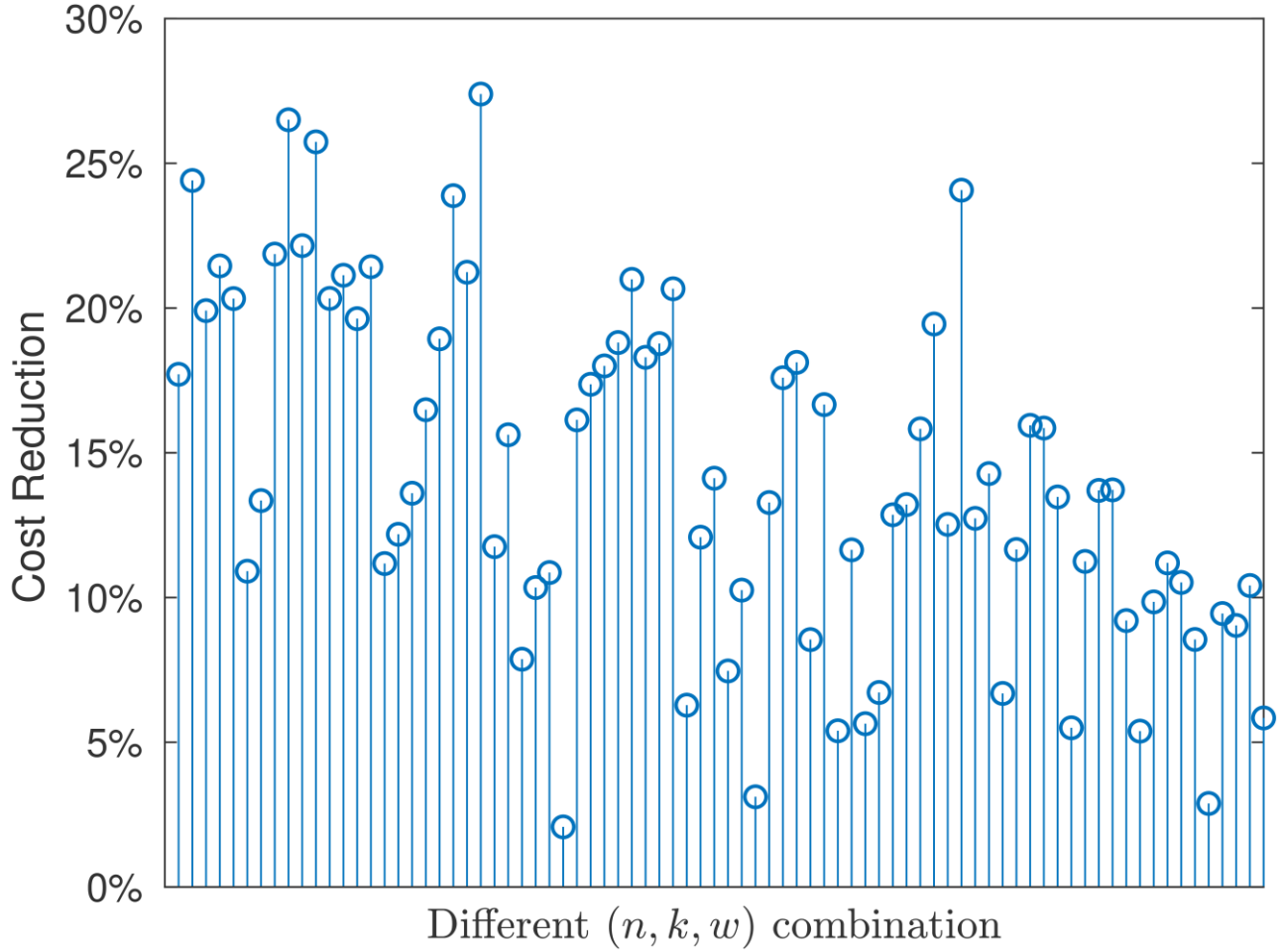
(i,j)	$(n,k,w)=(8,6,8)$			$(n,k,w)=(9,6,8)$			$(n,k,w)=(10,6,8)$			$(n,k,w)=(12,8,8)$			$(n,k,w)=(16,10,8)$		
(0,0)	378	291	**247**	573	467	**425**	768	611	**582**	1060	880	**841**	1968	1685	**1681**
(0,1)	256	234	**225**	413	393	**349**	556	518	**479**	805	740	**684**	1546	1432	**1374**
(0,2)	286	227	**217**	408	346	**321**	532	474	**450**	726	636	**591**	1304	1180	**1170**
(0,3)	286	216	**197**	408	357	**329**	532	460	**450**	726	636	**627**	1304	1180	**1170**
(1,0)	185	151	**142**	328	286	**262**	467	434	**419**	686	613	**563**	1389	1293	**1246**
(1,1)	164	155	**133**	285	270	**230**	411	383	**371**	593	557	**544**	1264	1184	**1133**
(1,2)	167	143	**134**	272	244	**225**	377	343	**335**	520	487	**462**	998	951	**922**
(1,3)	167	144	**130**	273	249	**233**	377	**337**	**337**	520	473	**467**	995	941	**932**

Strategies-(i,j) with Optimized Bitmatrices



strategy-(1,3) is the best strategy

Optimized Bitmatrix Reduced Costs



Cost Function Improvement

- Memcpy is 1.5x faster than XOR
- Cost functions:
  - Total # of XORs in schedule
  - Total # of ops in schedule
  - Weighted sum of ops in schedule

Table 4: Encoding throughput (GB/s) using bitmatrices obtained by the genetic algorithm under different cost functions

(n,k,w)	Cost Function		
	# of XOR	# of XOR and copying	Weighted
(8,6,4)	4.64	4.66	**4.68**
(8,6,8)	4.30	**4.35**	4.32
(9,6,4)	3.72	3.73	**3.80**
(9,6,8)	3.28	3.28	**3.44**
(10,6,4)	2.33	2.51	**2.52**
(10,6,8)	1.99	1.97	**2.11**
(12,8,4)	2.96	3.11	**3.16**
(12,8,8)	2.54	2.56	**2.58**
(16,10,4)	2.29	2.29	**2.32**
(16,10,8)	1.71	1.72	**1.74**

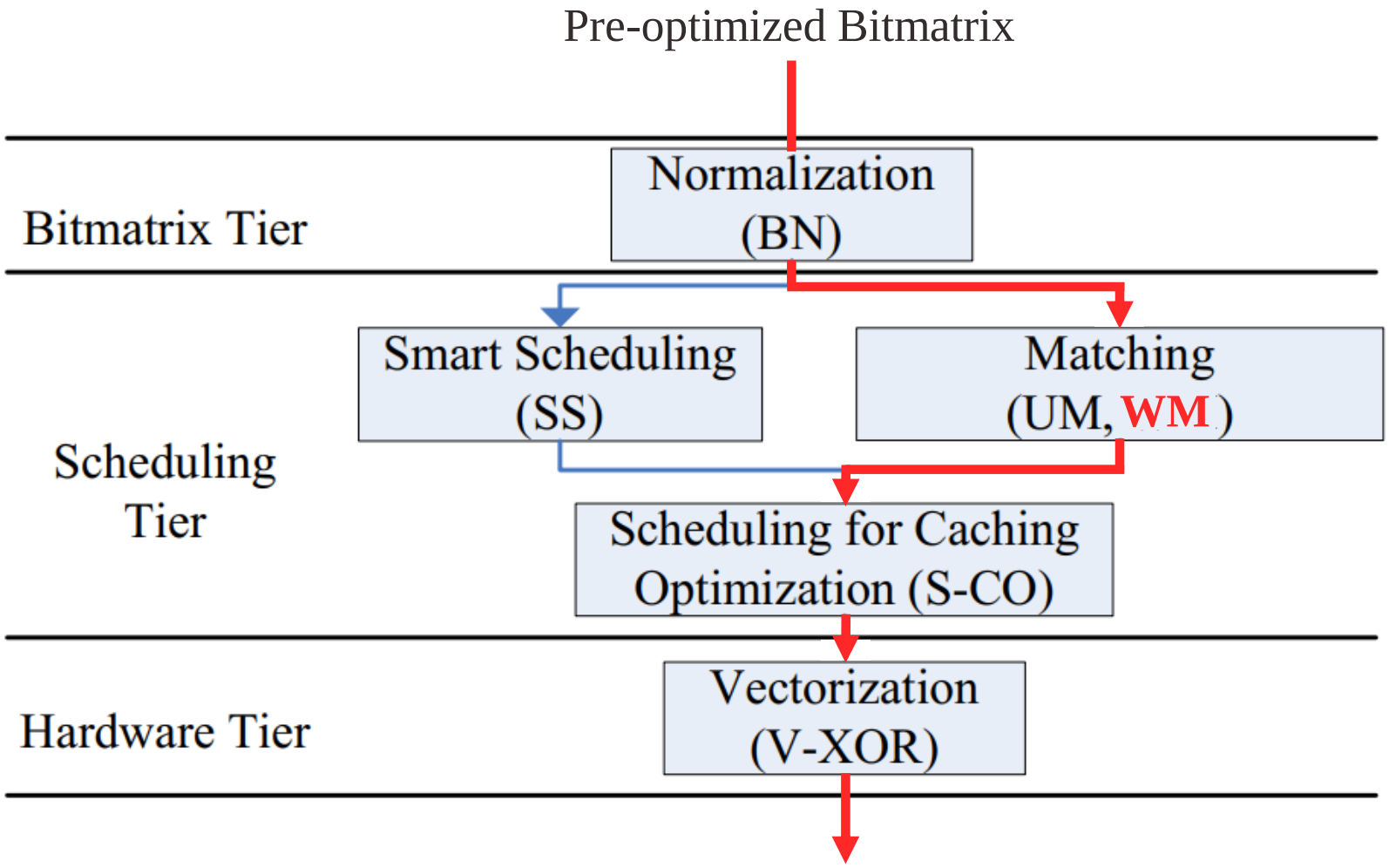
Contents

- Motivation
- Background and Review

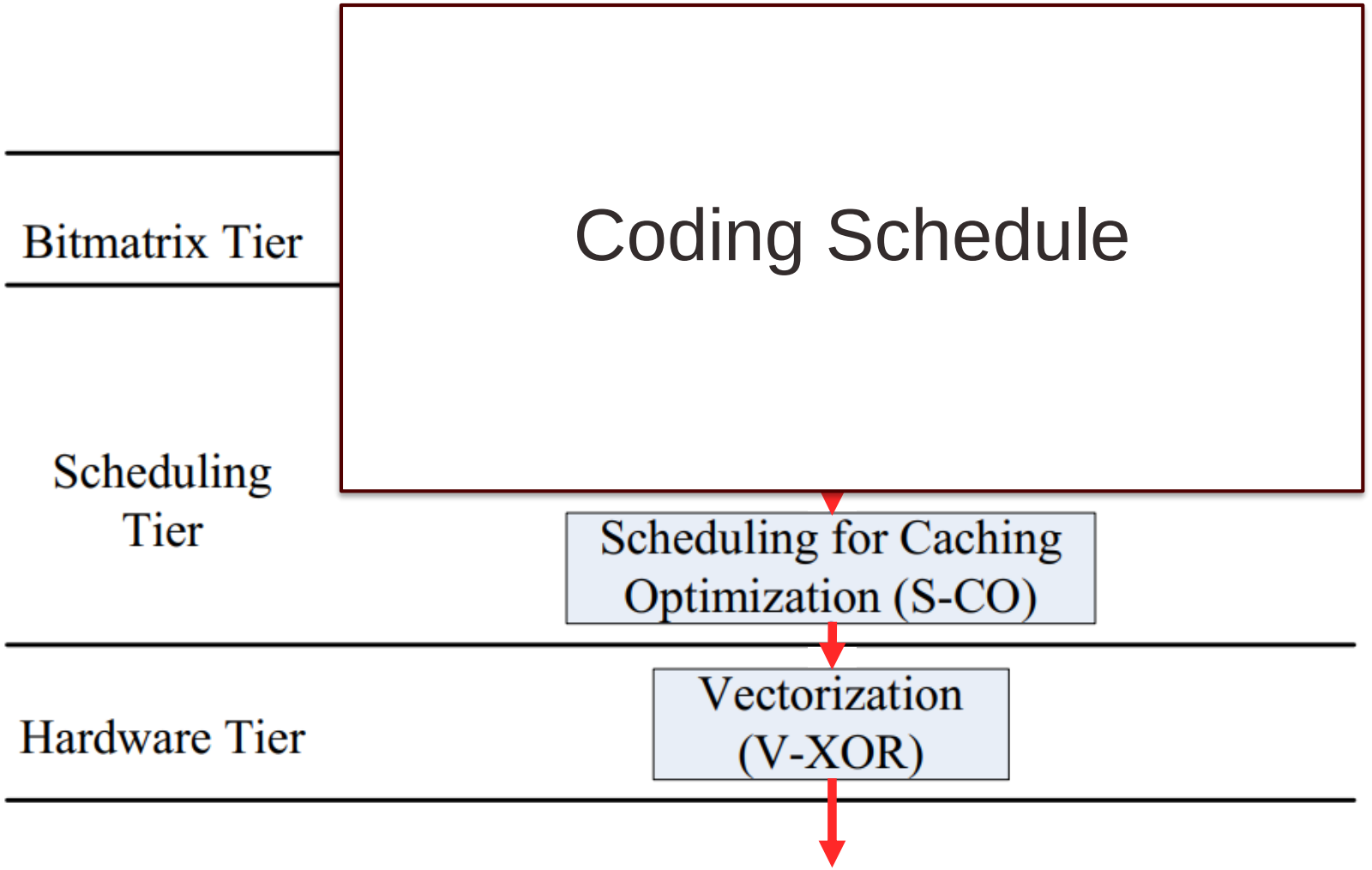
- Evaluating Individual Techniques
- Find the Best Strategy under Optimized Bitmatrix
- Proposed Coding Procedure and Evaluation

- Conclusion

Proposed Coding Procedure



Proposed Coding Procedure



Testing Setup

- Ryzen 1700X @ 3.4Ghz (8C/16T)
- 16GB DDR4
- Ubuntu 18.04 64-bit, GCC 7.3.0

- Jerasure 1.2A/ Jerasure 2.0:
  - XOR-based Cauchy RS code (BN, SS applied)
  - GF-based RS code
  - Raid-6
- Other codes:
  - EVENODD
  - RDP
  - STAR

Encoding v.s. Efficient RS/CRS code

Table 5: Encoding throughput (GB/s) for methods that allow general *(n, k)* parameters and $w = 8$

(n,k)	Proposed	Vectorized XOR-based CRS code	Vectorized GF-based RS code [16]
(7,5)	4.64	**4.73**	2.52
(8,6)	5.21	**5.22**	2.70
(9,7)	5.32	**5.45**	2.74
(10,8)	5.36	**5.59**	2.77
(12,10)	5.72	**5.88**	2.81
(8,5)	**3.19**	2.75	1.76
(9,6)	**3.49**	2.84	1.77
(10,7)	**3.67**	2.79	1.80
(11,8)	**3.72**	2.92	1.82
(13,10)	**3.82**	3.10	1.84
(10,6)	**2.55**	2.15	1.31
(11,7)	**2.75**	2.17	1.32
(12,8)	**2.86**	2.20	1.35
(14,10)	**2.86**	2.19	1.40
(15,10)	**2.30**	1.79	1.11
(16,10)	**1.96**	1.48	0.92

Outperform vectorized XOR-based CRS code for $m>2$

Encoding v.s. Efficient RS/CRS code

Table 5: Encoding throughput (GB/s) for methods that allow general *(n, k)* parameters and $w = 8$

(n,k)	Proposed	Vectorized XOR-based CRS code	Vectorized GF-based RS code [16]
(7,5)	4.64	**4.73**	2.52
(8,6)	5.21	**5.22**	2.70
(9,7)	5.32	**5.45**	2.74
(10,8)	5.36	**5.59**	2.77
(12,10)	5.72	**5.88**	2.81
(8,5)	**3.19**	2.75	1.76
(9,6)	**3.49**	2.84	1.77
(10,7)	**3.67**	2.79	1.80
(11,8)	**3.72**	2.92	1.82
(13,10)	**3.82**	3.10	1.84
(10,6)	**2.55**	2.15	1.31
(11,7)	**2.75**	2.17	1.32
(12,8)	**2.86**	2.20	1.35
(14,10)	**2.86**	2.19	1.40
(15,10)	**2.30**	1.79	1.11
(16,10)	**1.96**	1.48	0.92

Outperform vectorized XOR-based CRS code for $m>2$

Outperform vectorized GF-based RS code with big margin (~ 2x faster)

Encoding v.s. Three Parities Codes

Table 6: Encoding throughputs (GB/s): Three parities

(n,k,w)	Proposed	Vectorized XOR-based CRS code	STAR code [8]	Quantcast QFS [13]
(8,5,4)	**3.59**	3.25	2.97	2.92
(8,5,8)	**3.19**	2.75	2.97	2.92
(9,6,4)	3.52	**3.72**	3.42	3.04
(9,6,8)	**3.49**	2.84	3.42	3.04
(10,7,4)	**4.15**	3.86	3.76	3.25
(10,7,8)	3.67	2.79	**3.76**	3.25
(11,8,4)	**4.36**	4.13	3.94	3.27
(11,8,8)	3.72	2.92	**3.94**	3.27
(13,10,4)	**4.51**	4.08	4.37	3.41
(13,10,8)	3.82	3.10	**4.37**	3.41

Similar or better performance than specially designed codes

Encoding v.s. Two Parities Codes

Table 7: Encoding throughputs (GB/s): Two parities

(n,k,w)	Proposed	Vectorized XOR-based CRS code	Vectorized Raid-6	EVEN ODD [2]	RDP [6]
(7,5,4)	**5.16**	4.85	n/a	4.28	4.37
(7,5,8)	4.64	**4.73**	2.18	4.28	4.37
(8,6,4)	4.67	**5.22**	n/a	4.83	4.95
(8,6,8)	5.21	**5.22**	2.15	4.83	4.95
(9,7,4)	**5.77**	5.59	n/a	5.20	5.32
(9,7,8)	5.32	**5.45**	2.16	5.20	5.32
(10,8,4)	**5.90**	5.23	n/a	5.50	5.69
(10,8,8)	5.36	5.59	2.17	5.50	**5.69**
(12,10,4)	6.23	6.00	n/a	6.02	**6.24**
(12,10,8)	5.72	5.88	2.17	6.02	**6.24**

Similar or better performance than specially designed codes

Overall Encoding Improvement

Reference codes or methods	Improvement by proposed code
General (n,k) Codes	
GF-based RS code w/o vectorization	552.27%
XOR-based CRS code w/o vectorization	53.65%
Vectorized GF-based RS code [16]	99.82%
Vectorized XOR-based CRS code	14.98%
Three Parities Codes	
STAR [8]	5.59%
Quancast-QFS [13]	21.68%
Two Parities Codes	
Raid-6 w/o vectorization	206.88%
Vectorized Raid-6	142.07%
RDP [6]	5.85%
EVENODD [2]	8.79%

Decoding

Decoding

- Direct read in most cases

Decoding

- Direct read in most cases
- Degraded read (decoding) only if **systematic** nodes fail

Decoding

- Direct read in most cases
- Degraded read (decoding) only if **systematic** nodes fail
  - Single disk failure rate : 1%
  - In (10,6) erasure code storage system
    - 1 systematic node failure: ~ 0.055
    - 2 systematic node failure: ~ 0.0014
    - 3 systematic node failure: ~ 1.8×10^{-5}
    - 4 systematic node failure: ~ 1.4×10^{-7}

Decoding

- Direct read in most cases
- Degraded read (decoding) only if **systematic** nodes fail
  - Single disk failure rate : 1%
  - In (10,6) erasure code storage system
    - 1 systematic node failure: ~ 0.055
    - 2 systematic node failure: ~ 0.0014
    - 3 systematic node failure: ~ 1.8×10^{-5}
    - 4 systematic node failure: ~ 1.4×10^{-7}
- Decoding performance should be viewed as secondary importance

Decoding Throughput

Table 9: Decoding throughput (GB/s) for methods that allow general (n,k) parameters and $w = 8$

(n,k)	Proposed	Vectorized XOR-based CRS code	Vectorized GF-based RS code [16]
(7,5)	3.87	**4.56**	2.58
(8,6)	**5.45**	4.86	2.67
(9,7)	4.46	**5.06**	2.70
(10,8)	4.89	**5.11**	2.75
(12,10)	4.45	**5.52**	2.79
(8,5)	**3.04**	2.11	1.71
(9,6)	**2.94**	2.20	1.74
(10,7)	**3.28**	2.29	1.76
(11,8)	**3.08**	2.31	1.71
(13,10)	**3.21**	2.37	1.88
(10,6)	**2.38**	1.80	1.31
(11,7)	**2.35**	1.85	1.32
(12,8)	**2.54**	1.87	1.33
(14,10)	**2.47**	1.89	1.38
(15,10)	**2.00**	1.48	1.09
(16,10)	**1.77**	1.30	0.91

Table 10: Decoding throughputs (GB/s): Three parities

(n,k,w)	Proposed	Vectorized XOR-based CRS code	STAR code [8]	Quantcast QFS [13]
(8,5,4)	**4.28**	3.08	3.20	1.77
(8,5,8)	**3.04**	2.11	3.20	1.77
(9,6,4)	**4.13**	3.41	3.23	1.74
(9,6,8)	**2.94**	2.20	3.23	1.74
(10,7,4)	**4.55**	3.53	3.52	1.77
(10,7,8)	**3.28**	2.29	3.52	1.77
(11,8,4)	**4.70**	3.78	3.13	1.68
(11,8,8)	**3.08**	2.31	3.13	1.68
(13,10,4)	**4.86**	3.71	3.50	1.71
(13,10,8)	3.21	2.37	**3.50**	1.71

Table 11: Decoding throughputs (GB/s): Two parities

(n,k,w)	Proposed	Vectorized XOR-based CRS code	Vectorized Raid-6	EVEN ODD [2]	RDP [6]
(7,5,4)	5.52	4.85	n/a	6.66	**7.28**
(7,5,8)	3.87	4.65	2.64	6.66	**7.28**
(8,6,4)	5.43	5.14	n/a	7.42	**8.00**
(8,6,8)	5.45	4.86	2.67	7.42	**8.00**
(9,7,4)	6.03	5.37	n/a	7.65	**8.13**
(9,7,8)	4.46	5.06	2.73	7.65	**8.13**
(10,8,4)	5.88	5.73	n/a	7.93	**8.44**
(10,8,8)	4.89	5.11	2.77	7.93	**8.44**
(12,10,4)	6.23	5.89	n/a	7.49	**9.10**
(12,10,8)	4.45	5.52	2.81	7.49	**9.10**

Contents

- Motivation
- Background and Review

- Individual Techniques and Combinations
- Bitmatrix Optimization
- Proposed Coding Procedure and Evaluation

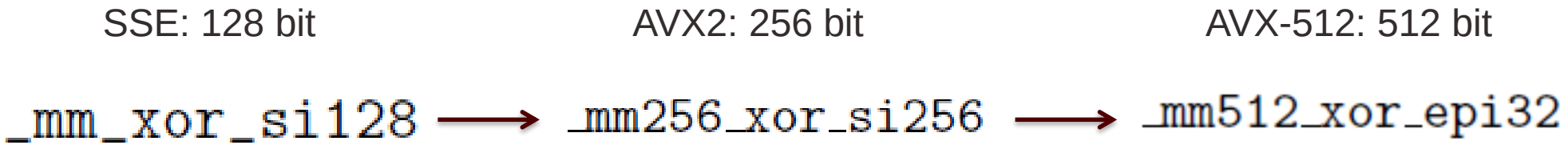
- Conclusion

Conclusion

- Comprehensive study of acceleration techniques
- Combine existing techniques and jointly optimize the bitmatrix
- Proposed approach outperforms most existing approaches in encoding throughput.

Conclusion : Key Finding

- Vectorization at XOR-level is much better choice than vectorization of finite field operations
  - Higher throughput (~ 2x faster)
  - Easy migration to newer SIMD ISA



Thank you!

Questions?

Tianli Zhou: zhoutl1106@tamu.edu

Chao Tian: chao.tian@tamu.edu

Source code:

https://bitbucket.org/zhoutl1106/zerasure.git

https://github.com/zhoutl1106/zerasure.git