Accelerating XOR-Based Erasure Coding using Program Optimization Techniques

Yuya Uezato

DWANGO, Co., Ltd.

SC21 St. Louis, MO | science & beyond.

“Accelerating Erasure Coding (EC)” means ...

Optimizing matrix multiplication over two finite fields:

$$\text{(for Standard EC)} \quad A \times B \quad \text{over } \mathbb{F}[2^8],$$

$$\text{(for XOR-based EC)} \quad C \times D \quad \text{over } \mathbb{F}[2].$$

- *Byte* Finite Field $\mathbb{F}[2^8]$ is a field with 256 elements.
- *Bit* Finite Field $\mathbb{F}[2] = \{0, 1\}$ is a field with the two bits.

"Accelerating Erasure Coding (EC)" means ...

Optimizin

- *Byte*
- *Bit Fi*

What we need to know about $\mathbb{F}[2^8]$ and $\mathbb{F}[2]$.

$\mathbb{F}[2^8]$ is a field with $2^8 = 256$ elements.

- ★ 1-byte (8-bits) data can be seen as an element of $\mathbb{F}[2^8]$.
- The definition is complex.
  (We will see it in the later page).

“Accelerating Erasure Coding (EC)” means ...

Optimizing

- *Byte* F
- *Bit* Fi

What we need to know about $\mathbb{F}[2^8]$ and $\mathbb{F}[2]$.

$\mathbb{F}[2^8]$ is a field with $2^8 = 256$ elements.

- ★ 1-byte (8-bits) data can be seen as an element of $\mathbb{F}[2^8]$.
- The definition is complex.
  (We will see it in the later page).

$\mathbb{F}[2] = \{0, 1\}$ is a field of bits.

- Its addition is XOR $\oplus$.
- Its multiplication is AND $\&$.
- 0 and 1 satisfy the following:

$$x \oplus 0 = 0 \oplus x = x, \quad y \& 1 = 1 \& y = y.$$

"Accelerating Erasure Coding (EC)" means ...

Optimizing matrix multiplication over two finite fields:

(for Standard EC) $A \times B$ over $\mathbb{F}[2^8]$,

(for XOR-based EC) $C \times D$ over $\mathbb{F}[2]$.

- *Byte* Finite Field $\mathbb{F}[2^8]$ is a field with 256 elements.
- *Bit* Finite Field $\mathbb{F}[2] = \{0, 1\}$ is a field with the two bits.

A is small. B is large.

“Accelerating Erasure Coding (EC)” means ...

Optimizing matrix multiplication over two finite fields:

$$\begin{array}{lll} \text{(for Standard EC)} & A \times B & \text{over } \mathbb{F}[2^8], \\ \text{(for XOR-based EC)} & C \times D & \text{over } \mathbb{F}[2]. \end{array}$$

- *Byte* Finite Field $\mathbb{F}[2^8]$ is a field with 256 elements.
- *Bit* Finite Field $\mathbb{F}[2] = \{0, 1\}$ is a field with the two bits.

A is small. B is large. In this talk, as an example, we consider:

$$\begin{array}{cc} & 10 \\ 14 & \boxed{A} \end{array} \times_{\mathbb{F}[2^8]} \begin{array}{cc} & \text{1M–4M} \\ 10 & \boxed{B} \end{array}$$

"Accelerating Erasure Coding (EC)" means ...

Optimizing matrix multiplication over two finite fields:

(for Standard EC) $A \times B$ over $\mathbb{F}[2^8]$,

(for XOR-based EC) $C \times D$ over $\mathbb{F}[2]$.

- *Byte* Finite Field $\mathbb{F}[2^8]$ is a field with 256 elements.
- *Bit* Finite Field $\mathbb{F}[2] = \{0, 1\}$ is a field with the two bits.

A is small. B is large. In this talk, as an example, we consider:

$$ {}_{14}\overset{10}{\boxed{A}} \times_{\mathbb{F}[2^8]} {}_{10}\overset{\text{1M–4M}}{\boxed{\quad B \quad}} $$

This setting comes from *erasure coding*.

What are Erasure Coding (EC) and XOR-Based EC

Erasure Coding: Method for Data Redundancy

Example (Building a streaming media server with criteria)

1. We have 14 nodes. Each node has a 20TB disk.
2. We can load data even if nodes ≤ 4 are down.
3. The total capacity of our server $=$ 200TB.
   - $14 \cdot 20 - 200 = 80$TB can be used for data redundancy.

Erasure Coding: Method for Data Redundancy

Example (Building a streaming media server with criteria)

1. We have 14 nodes. Each node has a 20TB disk.
2. We can load data even if nodes ≤ 4 are down.
3. The total capacity of our server = 200TB.
   - $14 \cdot 20 - 200 = 80$TB can be used for data redundancy.

For this criteria, we can employ Reed-Solomon EC $\mathbf{RS}(d = 10, p = 4)$.

- d: we can assume d-nodes are living. $(d = 14 - p = 10)$.
- p: we can permit nodes $\leq p$ go down. $(p = 4)$.

Erasure Coding: Method for Data Redundancy

Example (Building a streaming media server with criteria)

1. We have 14 nodes. Each node has a 20TB disk.
2. We can load data even if nodes ≤ 4 are down.
3. The total capacity of our server $=$ 200TB.
   - $14 \cdot 20 - 200 = 80$TB can be used for data redundancy.

For this criteria, we can employ Reed-Solomon EC $\mathbf{RS}(d = 10, p = 4)$.

- d: we can assume d-nodes are living. $(d = 14 - p = 10)$.
- p: we can permit nodes $\leq p$ go down. $(p = 4)$.

$$\underbrace{\boxed{\quad D \quad}}^{N\text{-bytes}} \xRightarrow{\text{encode}} \overset{\frac{N}{10}}{\boxed{f_1}} \; \overset{\frac{N}{10}}{\boxed{f_2}} \cdots \overset{\frac{N}{10}}{\boxed{f_{14}}} \,; \quad \text{put } \boxed{d_i} \text{ to node } n_i.$$

Erasure Coding: Method for Data Redundancy

Example (Building a streaming media server with criteria)

1. We have 14 nodes. Each node has a 20TB disk.
2. We can load data even if nodes ≤ 4 are down.
3. The total capacity of our server $=$ 200TB.
   - $14 \cdot 20 - 200 = 80$TB can be used for data redundancy.

For this criteria, we can employ Reed-Solomon EC $\mathbf{RS}(d = 10, p = 4)$.

- d: we can assume d-nodes are living. $(d = 14 - p = 10)$.
- p: we can permit nodes $\leq p$ go down. $(p = 4)$.

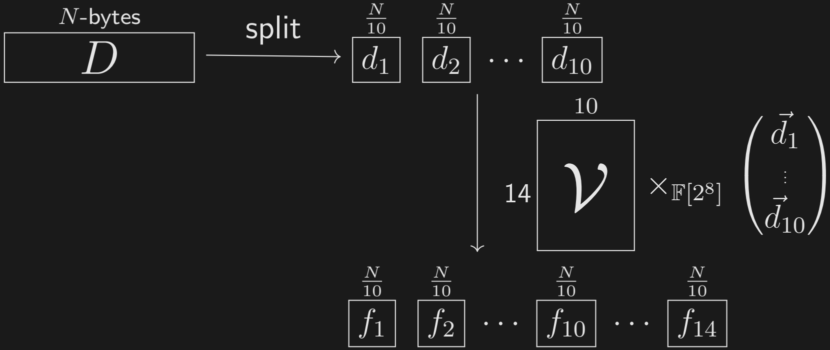
$$\overset{N\text{-bytes}}{\boxed{D}} \xRightarrow{\text{encode}} \overset{\frac{N}{10}}{\boxed{f_1}}\ \overset{\frac{N}{10}}{\boxed{f_2}} \cdots \overset{\frac{N}{10}}{\boxed{f_{14}}}\ ⨟ \quad \text{put } \boxed{d_i} \text{ to node } n_i.$$

$$\text{collect 10 } \boxed{d_i}\ ⨟ \quad \overset{\frac{N}{10}}{\boxed{f_{i_1}}}\ \overset{\frac{N}{10}}{\boxed{f_{i_2}}} \cdots \overset{\frac{N}{10}}{\boxed{f_{i_{10}}}} \xRightarrow{\text{decode}} \overset{N\text{-bytes}}{\boxed{D}}$$

How encoding and decoding are implemented in $\mathbf{RS}(10,4)$?

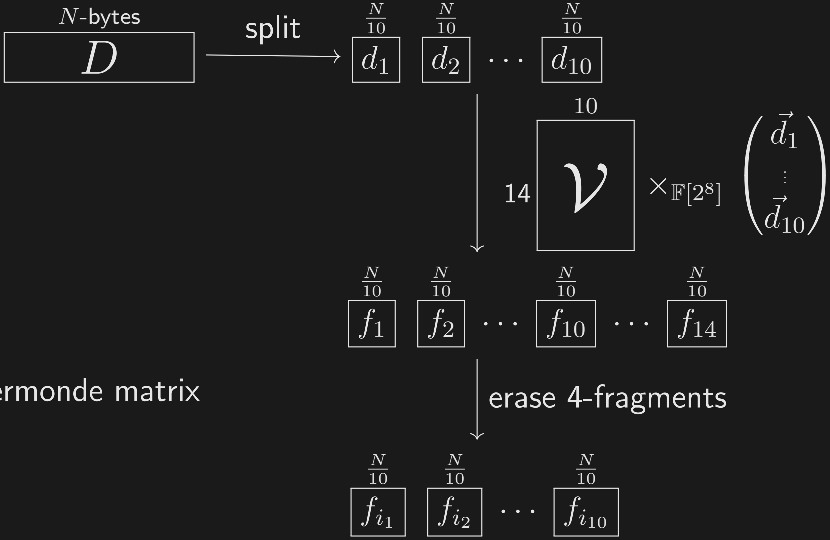
$$\overset{N\text{-bytes}}{\boxed{\quad D \quad}} \xrightarrow{\text{split}} \overset{\frac{N}{10}}{\boxed{d_1}}\ \overset{\frac{N}{10}}{\boxed{d_2}}\ \cdots\ \overset{\frac{N}{10}}{\boxed{d_{10}}}$$

How encoding and decoding are implemented in $\mathbf{RS}(10,4)$?



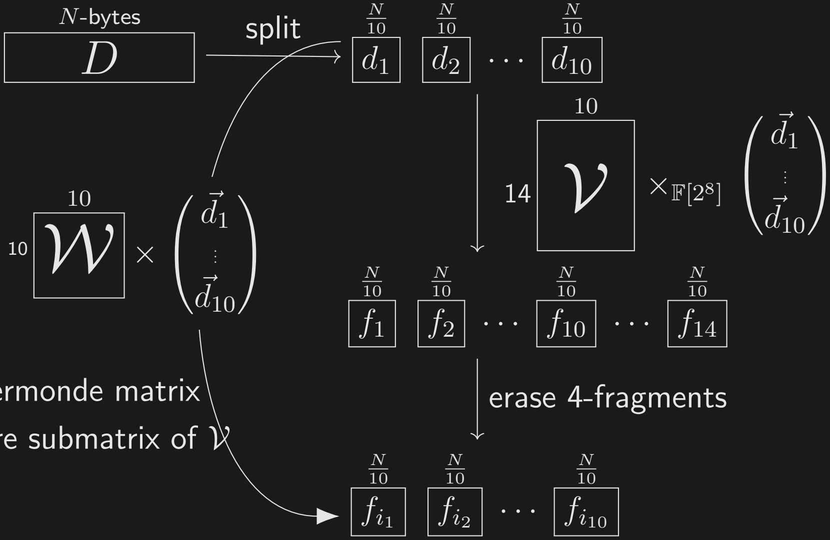
- $\mathcal{V}$: Vandermonde matrix

How encoding and decoding are implemented in $\mathbf{RS}(10, 4)$?



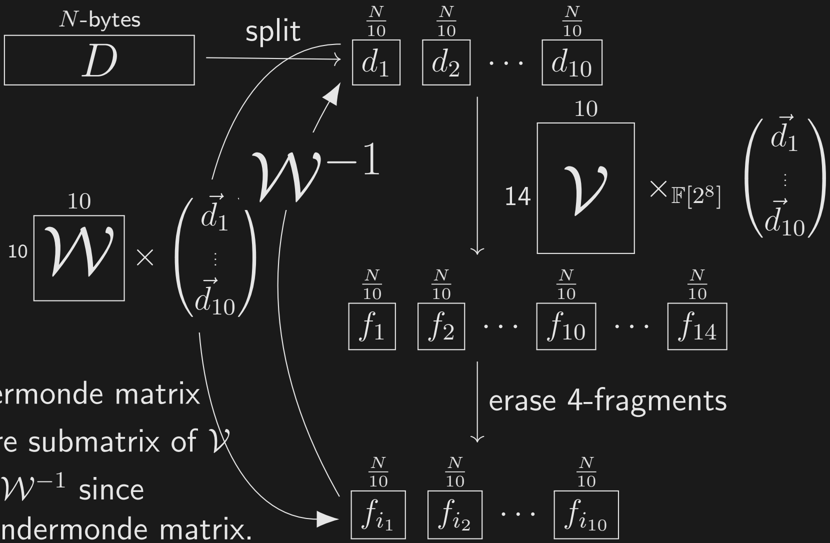
- $\mathcal{V}$: Vandermonde matrix

How encoding and decoding are implemented in $\mathbf{RS}(10, 4)$?



- $\mathcal{V}$: Vandermonde matrix
- $\mathcal{W}$: square submatrix of $\mathcal{V}$

How encoding and decoding are implemented in $\mathbf{RS}(10, 4)$?



- $\mathcal{V}$: Vandermonde matrix
- $\mathcal{W}$: square submatrix of $\mathcal{V}$
- We have $\mathcal{W}^{-1}$ since
  $\mathcal{V}$ is a Vandermonde matrix.

How encoding and decoding are implemented in $\mathbf{RS}(10,4)$?

N-bytes

D $\xrightarrow{\text{split}}$ $\overset{\frac{N}{10}}{\boxed{d_1}}$ $\overset{\frac{N}{10}}{\boxed{d_2}}$ $\cdots$ $\overset{\frac{N}{10}}{\boxed{d_{10}}}$

How large is N in a real application?

In my company, D is a short video whose size is 10MB–40MB:

- The size of 10 secs videos of 1080p & 30fps $\sim 12\text{MB}$.
- The size of 5 secs videos of 4K & 30fps $\sim 35\text{MB}$.

$\boxed{f_1}$ $\boxed{f_2}$ $\cdots$ $\boxed{f_{10}}$ $\cdots$ $\boxed{f_{14}}$

erase 4-fragments

$\overset{\frac{N}{10}}{\boxed{f_{i_1}}}$ $\overset{\frac{N}{10}}{\boxed{f_{i_2}}}$ $\cdots$ $\overset{\frac{N}{10}}{\boxed{f_{i_{10}}}}$

- $\mathcal{V}$: Vandermonde matrix
- $\mathcal{W}$: square submatrix of $\mathcal{V}$
- We have $\mathcal{W}^{-1}$ since $\mathcal{V}$ is a Vandermonde matrix.

Optimizing $\mathcal{V} \times_{\mathbb{F}[2^8]} D$

Q. What is the heaviest operation on $\mathcal{V} \times_{\mathbb{F}[2^8]} D$?

A. Multiplication of $\mathbb{F}[2^8]$:

Optimizing $\mathcal{V} \times_{\mathbb{F}[2^8]} D$

Q. What is the heaviest operation on $\mathcal{V} \times_{\mathbb{F}[2^8]} D$?
A. Multiplication of $\mathbb{F}[2^8]$:

- Internally, $p \in \mathbb{F}[2^8]$ is a 7-degree polynomial over $\mathbb{F}[2]$:

$$b_7 x^7 + b_6 x^6 + \cdots + b_1 x + b_0 \qquad \text{where } b_i \in \mathbb{F}[2].$$

Optimizing $\mathcal{V} \times_{\mathbb{F}[2^8]} D$

Q. What is the heaviest operation on $\mathcal{V} \times_{\mathbb{F}[2^8]} D$?
A. Multiplication of $\mathbb{F}[2^8]$:

- Internally, $p \in \mathbb{F}[2^8]$ is a 7-degree polynomial over $\mathbb{F}[2]$:

$$b_7 x^7 + b_6 x^6 + \cdots + b_1 x + b_0 \qquad \text{where } b_i \in \mathbb{F}[2].$$

- $p_1 + p_2$ of $\mathbb{F}[2^8]$ is the polynomial addition.
  Easy because just componentwise XOR:

$$(b_7 \oplus b'_7)x^7 + (b_6 \oplus b'_6)x^6 + \cdots + (b_0 \oplus b'_0).$$

Optimizing $\mathcal{V} \times_{\mathbb{F}[2^8]} D$

Q. What is the heaviest operation on $\mathcal{V} \times_{\mathbb{F}[2^8]} D$?
A. Multiplication of $\mathbb{F}[2^8]$:

- Internally, $p \in \mathbb{F}[2^8]$ is a 7-degree polynomial over $\mathbb{F}[2]$:

$$b_7 x^7 + b_6 x^6 + \cdots + b_1 x + b_0 \qquad \text{where } b_i \in \mathbb{F}[2].$$

- $p_1 + p_2$ of $\mathbb{F}[2^8]$ is the polynomial addition.
  Easy because just componentwise XOR:

$$(b_7 \oplus b'_7)x^7 + (b_6 \oplus b'_6)x^6 + \cdots + (b_0 \oplus b'_0).$$

- On the other hand, $p_1 \cdot p_2$ of $\mathbb{F}[2^8]$ is CPU-heavy and slow:
  1. We do the 7-degree polynomial multiplication $p_1 \times p_2$.
  2. We take the modulo by a special polynomial $(p_1 \times p_2) \bmod p$.

Optimizing $\mathcal{V} \times_{\mathbb{F}[2^8]} D$

Q. What is the heaviest operation on $\mathcal{V} \times_{\mathbb{F}[2^8]} D$?
A. Multiplication of $\mathbb{F}[2^8]$:

- Internally, $p \in \mathbb{F}[2^8]$ is a 7-degree polynomial over $\mathbb{F}[2]$:

$$b_7 x^7 + b_6 x^6 + \cdots + b_1 x + b_0 \qquad \text{where } b_i \in \mathbb{F}[2].$$

- $p_1 + p_2$ of $\mathbb{F}[2^8]$ is the polynomial addition.
  Easy because just componentwise XOR:

$$(b_7 \oplus b'_7)x^7 + (b_6 \oplus b'_6)x^6 + \cdots + (b_0 \oplus b'_0).$$

- On the other hand, $p_1 \cdot p_2$ of $\mathbb{F}[2^8]$ is CPU-heavy and slow:
  1. We do the 7-degree polynomial multiplication $p_1 \times p_2$.
  2. We take the modulo by a special polynomial $(p_1 \times p_2) \bmod p$.

XOR-based EC is one way to vanish $\cdot$ of $\mathbb{F}[2^8]$.

XOR-based EC: From $\mathbb{F}[2^8]$ to BitMatrix ($\mathbb{F}[2]$-Matrix)

- 1-byte and 8-bits are isomorphic: $x \in \mathbb{F}[2^8] \;\cong\; \widetilde{x} \in \begin{array}{cc} & 1 \\ 8 & \boxed{\mathbb{F}[2]} \end{array}$.

XOR-based EC: From $\mathbb{F}[2^8]$ to BitMatrix ($\mathbb{F}[2]$-Matrix)

- 1-byte and 8-bits are isomorphic: $x \in \mathbb{F}[2^8] \;\cong\; \widetilde{x} \in \begin{array}{cc} & 1 \\ 8 & \boxed{\mathbb{F}[2]} \end{array}$.
- There is an injective ring homomorphism $\mathcal{B} : \mathbb{F}[2^8] \to \begin{array}{cc} & 8 \\ 8 & \boxed{\;\mathbb{F}[2]\;} \end{array}$ I.e.,

XOR-based EC: From $\mathbb{F}[2^8]$ to BitMatrix ($\mathbb{F}[2]$-Matrix)

- 1-byte and 8-bits are isomorphic: $x \in \mathbb{F}[2^8] \;\cong\; \widetilde{x} \in 8 \overset{1}{\boxed{\mathbb{F}[2]}}$.
- There is an injective ring homomorphism $\mathcal{B} : \mathbb{F}[2^8] \to 8 \overset{8}{\boxed{\;\mathbb{F}[2]\;}}$ I.e.,

$$\forall x, y \in \mathbb{F}[2^8].\; \begin{cases} x + y = \mathcal{B}^{-1}(\mathcal{B}(x) + \mathcal{B}(y)), \\ \;x \cdot y \;= \mathcal{B}^{-1}(\mathcal{B}(x) \times \mathcal{B}(y)) \end{cases}$$

XOR-based EC: From $\mathbb{F}[2^8]$ to BitMatrix ($\mathbb{F}[2]$-Matrix)

- 1-byte and 8-bits are isomorphic: $x \in \mathbb{F}[2^8] \;\cong\; \widetilde{x} \in {}_{8}\overset{1}{\boxed{\mathbb{F}[2]}}$.
- There is an injective ring homomorphism $\mathcal{B} : \mathbb{F}[2^8] \to {}_{8}\overset{8}{\boxed{\;\mathbb{F}[2]\;}}$ I.e.,

$$\forall x, y \in \mathbb{F}[2^8].\; \begin{cases} x + y = \mathcal{B}^{-1}(\mathcal{B}(x) + \mathcal{B}(y)), \\ \;x \cdot y \;= \mathcal{B}^{-1}(\mathcal{B}(x) \times \mathcal{B}(y)) \end{cases}$$

Prop: Emulate $\mathcal{W}^{-1} \times (\mathcal{W} \times D) = D$ in the $\mathbb{F}[2]$ world

$$\mathcal{B}(\mathcal{W}^{-1}) \overset{\mathbb{F}[2]}{\times} (\mathcal{B}(\mathcal{W}) \overset{\mathbb{F}[2]}{\times} \widetilde{D}) = \mathcal{B}(\mathcal{W}^{-1} \times \mathcal{W}) \times \widetilde{D} = \widetilde{D}.$$

XOR-based EC: From $\mathbb{F}[2^8]$ to BitMatrix ($\mathbb{F}[2]$-Matrix)

- 1-byte and 8-bits are isomorphic: $x \in \mathbb{F}[2^8] \;\cong\; \widetilde{x} \in \begin{array}{cc} & 1 \\ 8 & \boxed{\mathbb{F}[2]} \end{array}$.
- There is an injective ring homomorphism $\mathcal{B} : \mathbb{F}[2^8] \to \begin{array}{cc} & 8 \\ 8 & \boxed{\;\mathbb{F}[2]\;} \end{array}$

$$\begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \end{pmatrix} \times_{\mathbb{F}[2^8]} \begin{pmatrix} d_1 & \cdots \\ d_2 & \cdots \end{pmatrix} = \begin{pmatrix} x_1 \cdot d_1 + x_2 \cdot d_2 & \cdots \\ x_3 \cdot d_1 + x_4 \cdot d_4 & \cdots \end{pmatrix}$$

$$\Downarrow$$

$$\begin{pmatrix} 1 & 1 & 0 & 1 & \cdots \\ 0 & 0 & 1 & 1 & \cdots \\ 0 & 1 & 1 & 0 & \cdots \\ 1 & 0 & 0 & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \times_{\mathbb{F}[2]} \begin{pmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vec{x}_3 \\ \vec{x}_4 \\ \vdots \end{pmatrix} = \begin{pmatrix} \vec{x}_1 \oplus \vec{x}_2 \oplus \vec{x}_4 \oplus \cdots \\ \vec{x}_3 \oplus \vec{x}_4 \oplus \cdots \\ \vec{x}_2 \oplus \vec{x}_3 \oplus \cdots \\ \vec{x}_1 \oplus \vec{x}_4 \oplus \cdots \\ \vdots \end{pmatrix}$$

$\oplus$ is byte-array XOR.

Comparing MM over $\mathbb{F}[2^8]$ and MM over $\mathbb{F}[2]$ for Encoding

Trade-off in Matrix Multiplication	$\mathbf{RS}(10, 4)$ by $\mathbb{F}[2^8]$	$\mathbf{RS}(10, 4)$ by $\mathbb{F}[2]$
Number of Core Operation	$\mathcal{V}$: $14 \overset{10}{\boxed{\mathbb{F}[2^8]}}$	$\mathcal{B}(\mathcal{V})$: $112 \overset{80}{\boxed{\mathbb{F}[2]}}$
Speed of Core Operation	$+$ of $\mathbb{F}[2^8]$ is fast $\cdot$ of $\mathbb{F}[2^8]$ is slow	bytevec-XOR $\oplus$ is fast (SIMDable)

Comparing MM over $\mathbb{F}[2^8]$ and MM over $\mathbb{F}[2]$ for Encoding

Trade-off in Matrix Multiplication	$\mathbf{RS}(10,4)$ by $\mathbb{F}[2^8]$	$\mathbf{RS}(10,4)$ by $\mathbb{F}[2]$
Number of Core Operation	$\mathcal{V}$: $14 \overbrace{\boxed{\mathbb{F}[2^8]}}^{10}$	$\mathcal{B}(\mathcal{V})$: $112 \overbrace{\boxed{\mathbb{F}[2]}}^{80}$
Speed of Core Operation	$+$ of $\mathbb{F}[2^8]$ is fast $\cdot$ of $\mathbb{F}[2^8]$ is slow	bytevec-XOR $\oplus$ is fast (SIMDable)

Encoding Throughput Comparison (on Intel CPU):

GB/s	ISA-L♣ $\mathbb{F}[2^8]$	State-of-the-art♠ $\mathbb{F}[2]$	
RS(10, 4)	6.79	4.94	
RS(10, 3)	6.78	6.15	
RS(9, 3)	7.31	6.17	

♣ ISA-L: Intel's EC library `https://github.com/intel/isa-l`

♠ T. Zhou & C. Tian. 2020. *Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques.*

Comparing MM over $\mathbb{F}[2^8]$ and MM over $\mathbb{F}[2]$ for Encoding

Trade-off in Matrix Multiplication	$\mathbf{RS}(10,4)$ by $\mathbb{F}[2^8]$	$\mathbf{RS}(10,4)$ by $\mathbb{F}[2]$
Number of Core Operation	$\mathcal{V} : 14 \overset{10}{\boxed{\mathbb{F}[2^8]}}$	$\mathcal{B}(\mathcal{V}) : 112 \overset{80}{\boxed{\mathbb{F}[2]}}$
Speed of Core Operation	$+$ of $\mathbb{F}[2^8]$ is fast $\cdot$ of $\mathbb{F}[2^8]$ is slow	bytevec-XOR $\oplus$ is fast (SIMDable)

Encoding Throughput Comparison (on Intel CPU):

GB/s	ISA-L♣ $\mathbb{F}[2^8]$	State-of-the-art♠ $\mathbb{F}[2]$	Ours(New!) $\mathbb{F}[2]$
RS(10, 4)	6.79	4.94	8.92
RS(10, 3)	6.78	6.15	11.78
RS(9, 3)	7.31	6.17	11.97

♣ ISA-L: Intel's EC library `https://github.com/intel/isa-l`

♠ T. Zhou & C. Tian. 2020. *Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques.*

Our Contribution:

Optimizing Bitmatrix Multiplication

as

Program Optimization Problem

MM over $\mathbb{F}[2]$ = Running Straight Line Program

We identify bitmatrix multiplication as *straight line program* (SLP):

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} \times_{\mathbb{F}[2]} \begin{pmatrix} \vec{a} \\ \vec{b} \\ \vec{c} \\ \vec{d} \end{pmatrix}$$

$$\Downarrow$$

$$\begin{array}{l} P(a, b, c, d) \\ \hline v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow a \oplus b \oplus c; \\ v_3 \leftarrow b \oplus c \oplus d; \\ \textsf{return}(v_1, v_2, v_3) \end{array}$$

MM over $\mathbb{F}[2]$ = Running Straight Line Program

We identify bitmatrix multiplication as *straight line program* (SLP):

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} \times_{\mathbb{F}[2]} \begin{pmatrix} \vec{a} \\ \vec{b} \\ \vec{c} \\ \vec{d} \end{pmatrix} = \begin{pmatrix} \vec{a} \oplus \vec{b} \\ \vec{a} \oplus \vec{b} \oplus \vec{c} \\ \vec{b} \oplus \vec{c} \oplus \vec{d} \end{pmatrix}$$

$$\Downarrow$$

$$\frac{P(a, b, c, d)}{\begin{array}{l} v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow a \oplus b \oplus c; \\ v_3 \leftarrow b \oplus c \oplus d; \\ \textsf{return}(v_1, v_2, v_3) \end{array}}$$

$$\begin{aligned} [\![P]\!] &= \textsf{return}(v_1, v_2, v_3) \\ &= \langle a \oplus b, \\ &\quad\; a \oplus b \oplus c, \\ &\quad\; b \oplus c \oplus d \rangle \end{aligned}$$

MM over $\mathbb{F}[2]$ = Running Straight Line Program

We identify bitmatrix multiplication as *straight line program* (SLP):

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} \times_{\mathbb{F}[2]} \begin{pmatrix} \vec{a} \\ \vec{b} \\ \vec{c} \\ \vec{d} \end{pmatrix} = \begin{pmatrix} \vec{a} \oplus \vec{b} \\ \vec{a} \oplus \vec{b} \oplus \vec{c} \\ \vec{b} \oplus \vec{c} \oplus \vec{d} \end{pmatrix}$$

$$\Downarrow$$

$$\frac{P(a, b, c, d)}{\begin{array}{l} v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow a \oplus b \oplus c; \\ v_3 \leftarrow b \oplus c \oplus d; \\ \text{return}(v_1, v_2, v_3) \end{array}}$$

$$\begin{aligned} [\![P]\!] &= \text{return}(v_1, v_2, v_3) \\ &= \langle a \oplus b, \\ &\quad\; a \oplus b \oplus c, \\ &\quad\; b \oplus c \oplus d \rangle \end{aligned}$$

★ "Bitmatrix as SLP" is not a new idea (See. Boyar+ 2008)

MM over $\mathbb{F}[2]$ = Running Straight Line Program

We identify bitmatrix multiplication as *straight line program* (SLP):

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} \times_{\mathbb{F}[2]} \begin{pmatrix} \vec{a} \\ \vec{b} \\ \vec{c} \\ \vec{d} \end{pmatrix} = \begin{pmatrix} \vec{a} \oplus \vec{b} \\ \vec{a} \oplus \vec{b} \oplus \vec{c} \\ \vec{b} \oplus \vec{c} \oplus \vec{d} \end{pmatrix}$$

$$\Downarrow$$

$$\frac{P(a, b, c, d)}{\begin{array}{l} v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow a \oplus b \oplus c; \\ v_3 \leftarrow b \oplus c \oplus d; \\ \text{return}(v_1, v_2, v_3) \end{array}}$$

$$\begin{aligned} [\![P]\!] &= \text{return}(v_1, v_2, v_3) \\ &= \langle a \oplus b, \\ &\quad\ a \oplus b \oplus c, \\ &\quad\ b \oplus c \oplus d \rangle \end{aligned}$$

- ★ "Bitmatrix as SLP" is not a new idea (See. Boyar+ 2008)
- ▶ SLP only allow assignments with one kind *binary* operator $\oplus$.
- ▶ SLP do not have functions, `if`-branchings, and `while`-loop, etc.

XOR Optimization: Reducing XORs

Optimization Metric $\#_{\oplus}(\_)$: the number of XORs.

$P \quad \#_{\oplus} = 8$		Q
$v_1 \leftarrow a \oplus b;$ $v_2 \leftarrow a \oplus b \oplus c;$ $v_3 \leftarrow a \oplus b \oplus c \oplus d;$ $v_4 \leftarrow b \oplus c \oplus d;$ return(v_1, v_2, v_3, v_4)	$\Rightarrow$	

XOR Optimization: Reducing XORs

Optimization Metric $\#_{\oplus}(\_)$: the number of XORs.

$P \quad \#_{\oplus} = 8$		Q
$v_1 \leftarrow a \oplus b;$	$\Longrightarrow$	$v_1 \leftarrow a \oplus b;$
$v_2 \leftarrow a \oplus b \oplus c;$		
$v_3 \leftarrow a \oplus b \oplus c \oplus d;$		
$v_4 \leftarrow b \oplus c \oplus d;$		
return(v_1, v_2, v_3, v_4)		

XOR Optimization: Reducing XORs

Optimization Metric $\#_{\oplus}(\_)$: the number of XORs.

$P \quad \#_{\oplus} = 8$		Q
$v_1 \leftarrow a \oplus b;$		$v_1 \leftarrow a \oplus b;$
$v_2 \leftarrow a \oplus b \oplus c;$		$v_2 \leftarrow v_1 \oplus c;$
$v_3 \leftarrow a \oplus b \oplus c \oplus d;$	$\Longrightarrow$	
$v_4 \leftarrow b \oplus c \oplus d;$		
return(v_1, v_2, v_3, v_4)		

XOR Optimization: Reducing XORs

Optimization Metric $\#_{\oplus}(\_)$: the number of XORs.

$P \quad \#_{\oplus} = 8$		Q
$v_1 \leftarrow a \oplus b;$		$v_1 \leftarrow a \oplus b;$
$v_2 \leftarrow a \oplus b \oplus c;$	$\Longrightarrow$	$v_2 \leftarrow v_1 \oplus c;$
$v_3 \leftarrow a \oplus b \oplus c \oplus d;$		$v_3 \leftarrow v_2 \oplus d;$
$v_4 \leftarrow b \oplus c \oplus d;$		
return(v_1, v_2, v_3, v_4)		

XOR Optimization: Reducing XORs

Optimization Metric $\#_{\oplus}(\_)$: the number of XORs.

$P \quad \#_{\oplus} = 8$		$Q \quad \#_{\oplus} = 4$
$v_1 \leftarrow a \oplus b;$		$v_1 \leftarrow a \oplus b;$
$v_2 \leftarrow a \oplus b \oplus c;$		$v_2 \leftarrow v_1 \oplus c;$
$v_3 \leftarrow a \oplus b \oplus c \oplus d;$	$\Longrightarrow$	$v_3 \leftarrow v_2 \oplus d;$
$v_4 \leftarrow b \oplus c \oplus d;$		$v_4 \leftarrow v_3 \oplus a;$
		$\because (a \oplus b \oplus c \oplus d) \oplus a = b \oplus c \oplus d.$
return(v_1, v_2, v_3, v_4)		return(v_1, v_2, v_3, v_4)

XOR Optimization: Reducing XORs

Optimization Metric $\#_\oplus(\_)$: the number of XORs.

$$\begin{array}{l}
P \quad \#_\oplus = 8 \\
\hline
v_1 \leftarrow a \oplus b; \\
v_2 \leftarrow a \oplus b \oplus c; \\
v_3 \leftarrow a \oplus b \oplus c \oplus d; \\
v_4 \leftarrow b \oplus c \oplus d; \\
\\
\text{return}(v_1, v_2, v_3, v_4)
\end{array}
\implies
\begin{array}{l}
Q \quad \#_\oplus = 4 \\
\hline
v_1 \leftarrow a \oplus b; \\
v_2 \leftarrow v_1 \oplus c; \\
v_3 \leftarrow v_2 \oplus d; \\
v_4 \leftarrow v_3 \oplus a; \\
\quad \because (a \oplus b \oplus c \oplus d) \oplus a = b \oplus c \oplus d. \\
\text{return}(v_1, v_2, v_3, v_4)
\end{array}$$

- P and Q are equivalent: $[\![P]\!] = [\![Q]\!]$.
- Intuitively, Q $(\#_\oplus(Q) = 4)$ runs faster than P $(\#_\oplus(P) = 8)$.

XOR Optimization: Reducing XORs

Optimization Metric $\#_\oplus(\_)$: the number of XORs.

$P \quad \#_\oplus = 8$		$Q \quad \#_\oplus = 4$
$v_1 \leftarrow a \oplus b;$		$v_1 \leftarrow a \oplus b;$
$v_2 \leftarrow a \oplus b \oplus c;$		$v_2 \leftarrow v_1 \oplus c;$
$v_3 \leftarrow a \oplus b \oplus c \oplus d;$	$\Longrightarrow$	$v_3 \leftarrow v_2 \oplus d;$
$v_4 \leftarrow b \oplus c \oplus d;$		$v_4 \leftarrow v_3 \oplus a;$
		$\because (a \oplus b \oplus c \oplus d) \oplus a = b \oplus c \oplus d.$
return(v_1, v_2, v_3, v_4)		return(v_1, v_2, v_3, v_4)

- P and Q are equivalent: $[\![P]\!] = [\![Q]\!]$.
- Intuitively, Q ($\#_\oplus(Q) = 4$) runs faster than P ($\#_\oplus(P) = 8$).

Question. For a given SLP P,
can we quickly find the most efficient equivalent SLP Q?

XOR Optimization: Reducing XORs

Optimization Metric $\#_\oplus(\_)$: the number of XORs.

$P \quad \#_\oplus = 8$		$Q \quad \#_\oplus = 4$
$v_1 \leftarrow a \oplus b;$		$v_1 \leftarrow a \oplus b;$
$v_2 \leftarrow a \oplus b \oplus c;$		$v_2 \leftarrow v_1 \oplus c;$
$v_3 \leftarrow a \oplus b \oplus c \oplus d;$	$\Longrightarrow$	$v_3 \leftarrow v_2 \oplus d;$
$v_4 \leftarrow b \oplus c \oplus d;$		$v_4 \leftarrow v_3 \oplus a;$
		$\because (a \oplus b \oplus c \oplus d) \oplus a = b \oplus c \oplus d.$
return(v_1, v_2, v_3, v_4)		return(v_1, v_2, v_3, v_4)

- P and Q are equivalent: $[\![P]\!] = [\![Q]\!]$.
- Intuitively, Q ($\#_\oplus(Q) = 4$) runs faster than P ($\#_\oplus(P) = 8$).

Theorem (Boyar+ 2013)

Unless **P** $=$ **NP**, *for a given SLP P, in polynomial time, we cannot find Q such that $[\![P]\!] = [\![Q]\!]$ and minimizes $\#_\oplus(Q)$.*

Our Heuristic: Grammar Compression Algorithm RePair

Originally, RePair is an algorithm to compress context-free grammars.
We use it identifying SLPs as commutative CFGs.

- Larsson & Moffat. 1999. *Offline dictionary-based compression*
- Paar. 1997. *Optimized arithmetic for Reed-Solomon encoders*

Our Heuristic: Grammar Compression Algorithm RePair

Originally, RePair is an algorithm to compress context-free grammars.
We use it identifying SLPs as commutative CFGs.

- Larsson & Moffat. 1999. *Offline dictionary-based compression*
- Paar. 1997. *Optimized arithmetic for Reed-Solomon encoders*

RePair = **Re**peat Pair. The key operation is Pair:

$$\begin{aligned} v_1 &\leftarrow a \oplus b; \\ v_2 &\leftarrow a \oplus b \oplus c; \\ v_3 &\leftarrow a \oplus b \oplus c \oplus d; \\ v_4 &\leftarrow b \oplus c \oplus d; \\ &\qquad\qquad \#_{\oplus} = 8 \end{aligned} \quad \xRightarrow{\textsc{Pair}(a,\, c)} \quad \begin{aligned} t_1 &\leftarrow a \oplus c; \\ \hline v_1 &\leftarrow a \oplus b; \\ v_2 &\leftarrow t_1 \oplus b; \\ v_3 &\leftarrow t_1 \oplus b \oplus d; \\ v_4 &\leftarrow b \oplus c \oplus d; \end{aligned} \quad \#_{\oplus} = 7$$

Our Heuristic: Grammar Compression Algorithm RePair

Originally, RePair is an algorithm to compress context-free grammars.
We use it identifying SLPs as commutative CFGs.

- Larsson & Moffat. 1999. *Offline dictionary-based compression*
- Paar. 1997. *Optimized arithmetic for Reed-Solomon encoders*

RePair = **Re**peat Pair. The key operation is Pair:

$$\begin{array}{l} v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow a \oplus b \oplus c; \\ v_3 \leftarrow a \oplus b \oplus c \oplus d; \\ v_4 \leftarrow b \oplus c \oplus d; \\ \qquad\qquad \#_\oplus = 8 \end{array} \xRightarrow{\textsc{Pair}(a,\, c)} \begin{array}{l} t_1 \leftarrow a \oplus c; \\ \hline v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow t_1 \oplus b; \\ v_3 \leftarrow t_1 \oplus b \oplus d; \\ v_4 \leftarrow b \oplus c \oplus d; \end{array} \quad \#_\oplus = 7$$

How do we choose a pair of terms to do pairing?

Our Heuristic: Grammar Compression Algorithm RePair

Originally, RePair is an algorithm to compress context-free grammars.
We use it identifying SLPs as commutative CFGs.

- Larsson & Moffat. 1999. *Offline dictionary-based compression*
- Paar. 1997. *Optimized arithmetic for Reed-Solomon encoders*

RePair = **Re**peat Pair. The key operation is Pair:

$$\begin{aligned} v_1 &\leftarrow a \oplus b; \\ v_2 &\leftarrow a \oplus b \oplus c; \\ v_3 &\leftarrow a \oplus b \oplus c \oplus d; \\ v_4 &\leftarrow b \oplus c \oplus d; \end{aligned} \quad \#_\oplus = 8 \quad \xRightarrow{\textsc{Pair}(a,\, c)} \quad \begin{aligned} t_1 &\leftarrow a \oplus c; \\ \hline v_1 &\leftarrow a \oplus b; \\ v_2 &\leftarrow t_1 \oplus b; \\ v_3 &\leftarrow t_1 \oplus b \oplus d; \\ v_4 &\leftarrow b \oplus c \oplus d; \end{aligned} \quad \#_\oplus = 7$$

How do we choose a pair of terms to do pairing? *Greedy*.

$$\begin{aligned} v_1 &\leftarrow a \oplus b; \\ v_2 &\leftarrow a \oplus b \oplus c; \\ v_3 &\leftarrow a \oplus b \oplus c \oplus d; \\ v_4 &\leftarrow b \oplus c \oplus d; \end{aligned} \quad \xRightarrow{\textsc{Pair}(a,\, b)} \quad \begin{aligned} t_1 &\leftarrow a \oplus b; \\ \hline v_2 &\leftarrow t_1 \oplus c; \\ v_3 &\leftarrow t_1 \oplus c \oplus d; \\ v_4 &\leftarrow b \oplus c \oplus d; \end{aligned} \quad \#_\oplus = 6$$

Our Heuristic: Grammar Compression Algorithm RePair

Originally, RePair is an algorithm to compress context-free grammars.
We use it identifying SLPs as commutative CFGs.

- Larsson & Moffat. 1999. *Offline dictionary-based compression*
- Paar. 1997. *Optimized arithmetic for Reed-Solomon encoders*

RePair = **Re**peat Pair. The key operation is Pair:

$$\begin{array}{l} v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow a \oplus b \oplus c; \\ v_3 \leftarrow a \oplus b \oplus c \oplus d; \\ v_4 \leftarrow b \oplus c \oplus d; \\ \qquad\qquad \#_\oplus = 8 \end{array} \xrightarrow{\textsc{Pair}(a,\, c)} \begin{array}{l} t_1 \leftarrow a \oplus c; \\ \hline v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow t_1 \oplus b; \\ v_3 \leftarrow t_1 \oplus b \oplus d; \\ v_4 \leftarrow b \oplus c \oplus d; \end{array} \quad \#_\oplus = 7$$

$$\begin{array}{l} t_1 \leftarrow a \oplus b; \\ \hline v_2 \leftarrow t_1 \oplus c; \\ v_3 \leftarrow t_1 \oplus c \oplus d; \\ v_4 \leftarrow b \oplus c \oplus d; \end{array} \xrightarrow{\textsc{Pair}(t_1,c)} \begin{array}{l} t_1 \leftarrow a \oplus b; \\ t_2 \leftarrow t_1 \oplus c; \\ \hline v_3 \leftarrow t_2 \oplus d; \\ v_4 \leftarrow b \oplus c \oplus d; \end{array} \xrightarrow{\textsc{Pair}(b,c)} \begin{array}{l} t_1 \leftarrow a \oplus b; \\ t_2 \leftarrow t_1 \oplus c; \\ t_3 \leftarrow b \oplus c; \\ \hline v_3 \leftarrow t_2 \oplus d; \\ v_4 \leftarrow t_3 \oplus d; \end{array}$$

Our Heuristic: Grammar Compression Algorithm RePair

Originally, RePair is an algorithm to compress context-free grammars.
We use it identifying SLPs as commutative CFGs.

- Larsson & Moffat. 1999. *Offline dictionary-based compression*
- Paar. 1997. *Optimized arithmetic for Reed-Solomon encoders*

RePair = **Re**peat Pair. The key operation is Pair:

$$\begin{aligned} v_1 &\leftarrow a \oplus b; \\ v_2 &\leftarrow a \oplus b \oplus c; \\ v_3 &\leftarrow a \oplus b \oplus c \oplus d; \\ v_4 &\leftarrow b \oplus c \oplus d; \\ &\#_\oplus = 8 \end{aligned} \xRightarrow{\textsc{Pair}(a,\, c)} \begin{aligned} \underline{t_1 \leftarrow a \oplus c;} \\ v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow t_1 \oplus b; \\ v_3 \leftarrow t_1 \oplus b \oplus d; \\ v_4 \leftarrow b \oplus c \oplus d; \end{aligned} \quad \#_\oplus = 7$$

The commutative version of RePair accommodates

$$\text{Commutativity} : x \oplus y = y \oplus x, \;\; \text{Associativity} : (x \oplus y) \oplus z = x \oplus (y \oplus z).$$

In the paper, we extend it to XorRePair by accommodating

$$\text{Cancellativity: } x \oplus x \oplus y = y.$$

Memory Access Optimization: **MultiSLP**

Optimization Metric: $\#_{\mathsf{mem}}(\_) =$ the number of memory access.

Quiz: How many times will this program access memory?

$$\#_{\mathsf{mem}}\left(\quad v \leftarrow A \oplus B \oplus C \oplus D \quad \right) = \ ?$$

Memory Access Optimization: **MultiSLP**

Optimization Metric: $\#_{\mathsf{mem}}(\_) =$ the number of memory access.

Quiz: How many times will this program access memory?

$$\#_{\mathsf{mem}}\left(\quad v \leftarrow A \oplus B \oplus C \oplus D \quad \right) = 9$$

because each $\oplus$ issues two read and one write:

$$t_1 \leftarrow A \oplus B; \quad t_2 \leftarrow t_1 \oplus C; \quad v \leftarrow t_2 \oplus D;$$

Memory Access Optimization: **MultiSLP**

Optimization Metric: $\#_{\mathsf{mem}}(\_)$ = the number of memory access.

Quiz: How many times will this program access memory?

$$\#_{\mathsf{mem}}\left(\quad v \leftarrow A \oplus B \oplus C \oplus D \quad \right) = 9$$

because each $\oplus$ issues two read and one write:

$$t_1 \leftarrow A \oplus B; \quad t_2 \leftarrow t_1 \oplus C; \quad v \leftarrow t_2 \oplus D;$$

t_1 and t_2 are wasteful: they are released immediately after allocated.

To reduce such wastefulness,
we extend SLP to *Multi* SLP, which allows n-arity XORs.

Memory Access Optimization: **MultiSLP**

Optimization Metric: $\#_{\mathsf{mem}}(\_) =$ the number of memory access.

Quiz: How many times will this program access memory?

$$\#_{\mathsf{mem}}\left(\quad v \leftarrow A \oplus B \oplus C \oplus D \quad \right) = 9$$

because each $\oplus$ issues two read and one write:

$$t_1 \leftarrow A \oplus B; \quad t_2 \leftarrow t_1 \oplus C; \quad v \leftarrow t_2 \oplus D;$$

On MultiSLP, we can

$$v \leftarrow \oplus_4(A, B, C, D);$$

Thus, we have $\#_{\mathsf{mem}} = 5$.

```
⊕4(A, B, C, D: [byte]) {
  var v = Array::new(A.len);
  for i in [0..A.len):
    byte r = A[i] ^ B[i]
    r = r ^ C[i];
    v[i] = r ^ D[i];
  return v;
}
```

New Metric and Memory Optimization Problem

From a given P, can we quickly (= in polynomial time)
find an equivalent and most memory efficient Q w.r.t. $\#_{\mathsf{mem}}$?

$$P: \begin{array}{l} v_1 \leftarrow a \oplus b \oplus c \oplus d \oplus e; \\ v_2 \leftarrow a \oplus b \oplus c \oplus d \oplus f; \end{array}$$

$$\#_{\mathsf{mem}}(P) = 24$$

New Metric and Memory Optimization Problem

From a given P, can we quickly ($=$ in polynomial time)
find an equivalent and most memory efficient Q w.r.t. $\#_{\text{mem}}$?

$$P: \begin{array}{r} v_1 \leftarrow a \oplus b \oplus c \oplus d \oplus e; \\ v_2 \leftarrow a \oplus b \oplus c \oplus d \oplus f; \\ \#_{\text{mem}}(P) = 24 \end{array} \implies Q: \begin{array}{r} t \leftarrow \oplus_4(a, b, c, d); \\ v_1 \leftarrow t \oplus e; \\ v_2 \leftarrow t \oplus f; \\ \#_{\text{mem}}(Q) = 11 \end{array}$$

New Metric and Memory Optimization Problem

From a given P, can we quickly ($=$ in polynomial time)
find an equivalent and most memory efficient Q w.r.t. $\#_{\text{mem}}$?

$$P: \begin{array}{r} v_1 \leftarrow a \oplus b \oplus c \oplus d \oplus e; \\ v_2 \leftarrow a \oplus b \oplus c \oplus d \oplus f; \\ \#_{\text{mem}}(P) = 24 \end{array} \implies Q: \begin{array}{r} t \leftarrow \oplus_4(a, b, c, d); \\ v_1 \leftarrow t \oplus e; \\ v_2 \leftarrow t \oplus f; \\ \#_{\text{mem}}(Q) = 11 \end{array}$$

Unfortunately, we showed the following intractability result:

Theorem (Our NEW theoretical result)

Unless **P = NP**, *for a given SLP* P, *in polynomial time,*
we cannot find Q *that* $[\![P]\!] = [\![Q]\!]$ *and minimizes* $\#_{\text{mem}}(Q)$.

Our Heuristic: XOR Fusion

We fuse XORs when the following holds:

$$\begin{aligned} &\alpha \leftarrow \oplus(x_1, \ldots, x_n); \\ &\quad \vdots \\ &\beta \leftarrow \oplus(y_1, \ldots, \alpha, \ldots, y_m) \end{aligned} \xRightarrow{\text{fuse}} \beta \leftarrow \oplus(y_1, \ldots, x_1, \ldots, x_n, \ldots, y_m)$$

☆ α appears once in the program

Our Heuristic: XOR Fusion

We fuse XORs when the following holds:

$$\begin{array}{l} \alpha \leftarrow \oplus(x_1, \ldots, x_n); \\ \quad \vdots \\ \beta \leftarrow \oplus(y_1, \ldots, \alpha, \ldots, y_m) \end{array} \xRightarrow{\text{fuse}} \beta \leftarrow \oplus(y_1, \ldots, x_1, \ldots, x_n, \ldots, y_m)$$

☆ α appears once in the program

Example.

$$\begin{array}{l} v_1 \leftarrow a \oplus b \oplus c \oplus d \oplus e; \\ v_2 \leftarrow a \oplus b \oplus c \oplus d \oplus f; \\ \qquad\qquad \#_{\text{mem}}(24) \end{array} \xRightarrow{\textsc{RePair}} \begin{array}{l} t_1 \leftarrow a \oplus b; \\ t_2 \leftarrow t_1 \oplus c; \\ t_3 \leftarrow t_2 \oplus d; \\ v_1 \leftarrow t_3 \oplus e; \\ v_2 \leftarrow t_3 \oplus f; \\ \qquad \#_{\text{mem}}(15) \end{array} \xRightarrow{\text{fuse}(t_1)} \begin{array}{l} t_2 \leftarrow \oplus_3(a, b, c); \\ t_3 \leftarrow t_2 \oplus d; \\ v_1 \leftarrow t_3 \oplus e; \\ v_2 \leftarrow t_3 \oplus f; \\ \qquad \#_{\text{mem}}(13) \end{array}$$

Our Heuristic: XOR Fusion

We fuse XORs when the following holds:

$$\begin{array}{l} \alpha \leftarrow \oplus(x_1, \ldots, x_n); \\ \quad \vdots \\ \beta \leftarrow \oplus(y_1, \ldots, \alpha, \ldots, y_m) \end{array} \xrightarrow{\text{fuse}} \beta \leftarrow \oplus(y_1, \ldots, x_1, \ldots, x_n, \ldots, y_m)$$

$\star$ α appears once in the program

Example.

$$\begin{array}{l} v_1 \leftarrow a \oplus b \oplus c \oplus d \oplus e; \\ v_2 \leftarrow a \oplus b \oplus c \oplus d \oplus f; \\ \qquad\qquad \#_{\text{mem}}(24) \end{array} \xrightarrow{\text{RePair}} \begin{array}{l} t_1 \leftarrow a \oplus b; \\ t_2 \leftarrow t_1 \oplus c; \\ t_3 \leftarrow t_2 \oplus d; \\ v_1 \leftarrow t_3 \oplus e; \\ v_2 \leftarrow t_3 \oplus f; \\ \qquad \#_{\text{mem}}(15) \end{array} \xrightarrow{\text{fuse}(t_1)} \begin{array}{l} t_2 \leftarrow \oplus_3(a, b, c); \\ t_3 \leftarrow t_2 \oplus d; \\ v_1 \leftarrow t_3 \oplus e; \\ v_2 \leftarrow t_3 \oplus f; \\ \qquad \#_{\text{mem}}(13) \end{array}$$

$$\xrightarrow{\text{fuse}(t_2)} \begin{array}{l} t_3 \leftarrow \oplus_4(a, b, c, d); \\ v_1 \leftarrow t_3 \oplus e; \\ v_2 \leftarrow t_3 \oplus f; \\ \qquad\qquad \#_{\text{mem}}(11) \end{array}$$

Our Heuristic: XOR Fusion

We fuse XORs when the following holds:

$$\begin{array}{l} \alpha \leftarrow \oplus(x_1, \ldots, x_n); \\ \quad\vdots \\ \beta \leftarrow \oplus(y_1, \ldots, \alpha, \ldots, y_m) \end{array} \xrightarrow{\text{fuse}} \beta \leftarrow \oplus(y_1, \ldots, x_1, \ldots, x_n, \ldots, y_m)$$

$\star$ α appears once in the program

Example.

$$\begin{array}{l} v_1 \leftarrow a \oplus b \oplus c \oplus d \oplus e; \\ v_2 \leftarrow a \oplus b \oplus c \oplus d \oplus f; \\ \hfill \#_{\text{mem}}(24) \end{array} \xrightarrow{\text{RePair}} \begin{array}{l} t_1 \leftarrow a \oplus b; \\ t_2 \leftarrow t_1 \oplus c; \\ t_3 \leftarrow t_2 \oplus d; \\ v_1 \leftarrow t_3 \oplus e; \\ v_2 \leftarrow t_3 \oplus f; \\ \hfill \#_{\text{mem}}(15) \end{array} \xrightarrow{\text{fuse}(t_1)} \begin{array}{l} t_2 \leftarrow \oplus_3(a, b, c); \\ t_3 \leftarrow t_2 \oplus d; \\ v_1 \leftarrow t_3 \oplus e; \\ v_2 \leftarrow t_3 \oplus f; \\ \hfill \#_{\text{mem}}(13) \end{array}$$

$$\xrightarrow{\text{fuse}(t_2)} \begin{array}{l} t_3 \leftarrow \oplus_4(a, b, c, d); \\ v_1 \leftarrow t_3 \oplus e; \\ v_2 \leftarrow t_3 \oplus f; \\ \hfill \#_{\text{mem}}(11) \end{array} \xrightarrow{\text{NOT fuse}(t_3) \text{ by } \star} \begin{array}{l} v_1 \leftarrow \oplus_5(a, b, c, d, e); \\ v_2 \leftarrow \oplus_5(a, b, c, d, f); \\ \hfill \#_{\text{mem}}(12) \end{array}$$

Cache Optimization: SLP + LRU Cache

Metric $\#_{\mathsf{I/O}}(K, \_)$: the total number of I/O transfers between memory andf cache of K-capacity.

Cache Optimization: SLP + LRU Cache

Metric $\#_{\mathsf{I/O}}(K, \_)$: the total number of I/O transfers between memory andf cache of K-capacity.

We have three kinds of operations for cache:

- $\mathcal{H}(x)$: Cache Hit for an element x. $\#_{\mathsf{I/O}} = 0$.
- $\mathcal{R}(x)$: Cache miss. Evict LRU to mem. and read x from mem. $\#_{\mathsf{I/O}} = 2$.
- $\mathcal{W}(x)$: Cache miss. Evict LRU to mem. and write x to cache. $\#_{\mathsf{I/O}} = 1$.

Cache Optimization: SLP + LRU Cache

Metric $\#_{\mathsf{I/O}}(K, \_)$: the total number of I/O transfers between memory andf cache of K-capacity.

We have three kinds of operations for cache:

- $\mathcal{H}(x)$: Cache Hit for an element x. $\#_{\mathsf{I/O}} = 0$.
- $\mathcal{R}(x)$: Cache miss. Evict LRU to mem. and read x from mem. $\#_{\mathsf{I/O}} = 2$.
- $\mathcal{W}(x)$: Cache miss. Evict LRU to mem. and write x to cache. $\#_{\mathsf{I/O}} = 1$.

Example: Calculate $\#_{\mathsf{I/O}}(4, P)$ for the following example SLP P:

$v_1 \leftarrow A \oplus B;$ $\quad *_1 \; *_2 \; *_3 \; *_4$

$v_2 \leftarrow \oplus(E, D, A);$

$v_3 \leftarrow v_1 \oplus E;$

$v_4 \leftarrow v_1 \oplus C;$

$\text{return}(v_2, v_3, v_4);$

Cache Optimization: SLP + LRU Cache

Metric $\#_{\mathsf{I/O}}(K, \_)$: the total number of I/O transfers between memory andf cache of K-capacity.

We have three kinds of operations for cache:

- $\mathcal{H}(x)$: Cache Hit for an element x. $\#_{\mathsf{I/O}} = 0$.
- $\mathcal{R}(x)$: Cache miss. Evict LRU to mem. and read x from mem. $\#_{\mathsf{I/O}} = 2$.
- $\mathcal{W}(x)$: Cache miss. Evict LRU to mem. and write x to cache. $\#_{\mathsf{I/O}} = 1$.

Example: Calculate $\#_{\mathsf{I/O}}(4, P)$ for the following example SLP P:

$v_1 \leftarrow A \oplus B;$ $\qquad *_1 *_2 *_3 *_4 \xrightarrow[2]{\mathcal{R}(A)} *_2 *_3 *_4 A$

$v_2 \leftarrow \oplus(E, D, A);$

$v_3 \leftarrow v_1 \oplus E;$

$v_4 \leftarrow v_1 \oplus C;$

$\text{return}(v_2, v_3, v_4);$

Cache Optimization: SLP + LRU Cache

Metric $\#_{\mathsf{I/O}}(K, \_)$: the total number of I/O transfers between memory andf cache of K-capacity.

We have three kinds of operations for cache:

- $\mathcal{H}(x)$: Cache Hit for an element x. $\#_{\mathsf{I/O}} = 0$.
- $\mathcal{R}(x)$: Cache miss. Evict LRU to mem. and read x from mem. $\#_{\mathsf{I/O}} = 2$.
- $\mathcal{W}(x)$: Cache miss. Evict LRU to mem. and write x to cache. $\#_{\mathsf{I/O}} = 1$.

Example: Calculate $\#_{\mathsf{I/O}}(4, P)$ for the following example SLP P:

$v_1 \leftarrow A \oplus B;$ $\quad *_1 *_2 *_3 *_4 \xrightarrow[2]{\mathcal{R}(A)} *_2 *_3 *_4 A \xrightarrow[2]{\mathcal{R}(B)} *_3 *_4 AB$

$v_2 \leftarrow \oplus(E, D, A);$

$v_3 \leftarrow v_1 \oplus E;$

$v_4 \leftarrow v_1 \oplus C;$

$\mathsf{return}(v_2, v_3, v_4);$

Cache Optimization: SLP + LRU Cache

Metric $\#_{\mathsf{I/O}}(K, \_)$: the total number of I/O transfers between memory andf cache of K-capacity.

We have three kinds of operations for cache:

- $\mathcal{H}(x)$: Cache Hit for an element x. $\#_{\mathsf{I/O}} = 0$.
- $\mathcal{R}(x)$: Cache miss. Evict LRU to mem. and read x from mem. $\#_{\mathsf{I/O}} = 2$.
- $\mathcal{W}(x)$: Cache miss. Evict LRU to mem. and write x to cache. $\#_{\mathsf{I/O}} = 1$.

Example: Calculate $\#_{\mathsf{I/O}}(4, P)$ for the following example SLP P:

$v_1 \leftarrow A \oplus B;$ $\quad *_1 *_2 *_3 *_4 \xrightarrow[2]{\mathcal{R}(A)} *_2 *_3 *_4 A \xrightarrow[2]{\mathcal{R}(B)} *_3 *_4 AB \xrightarrow[1]{\mathcal{W}(v_1)}$

$v_2 \leftarrow \oplus(E, D, A);$ $\quad *_4\ ABv_1$

$v_3 \leftarrow v_1 \oplus E;$

$v_4 \leftarrow v_1 \oplus C;$

$\mathsf{return}(v_2, v_3, v_4);$

Cache Optimization: SLP + LRU Cache

Metric $\#_{\text{I/O}}(K, \_)$: the total number of I/O transfers between memory andf cache of K-capacity.

We have three kinds of operations for cache:

- $\mathcal{H}(x)$: Cache Hit for an element x. $\#_{\text{I/O}} = 0$.
- $\mathcal{R}(x)$: Cache miss. Evict LRU to mem. and read x from mem. $\#_{\text{I/O}} = 2$.
- $\mathcal{W}(x)$: Cache miss. Evict LRU to mem. and write x to cache. $\#_{\text{I/O}} = 1$.

Example: Calculate $\#_{\text{I/O}}(4, P)$ for the following example SLP P:

$v_1 \leftarrow A \oplus B;$	$*_1 *_2 *_3 *_4 \xrightarrow[2]{\mathcal{R}(A)} *_2 *_3 *_4 A \xrightarrow[2]{\mathcal{R}(B)} *_3 *_4 AB \xrightarrow[1]{\mathcal{W}(v_1)}$
$v_2 \leftarrow \oplus(E, D, A);$	$*_4\, ABv_1 \xrightarrow[2]{\mathcal{R}(E)} ABv_1E \xrightarrow[2]{\mathcal{R}(D)} Bv_1ED \xrightarrow[2]{\mathcal{R}(A)} v_1EDA \xrightarrow[1]{\mathcal{W}(v_2)}$
$v_3 \leftarrow v_1 \oplus E;$	$EDAv_2 \xrightarrow[2]{\mathcal{R}(v_1)} DAv_2v_1 \xrightarrow[2]{\mathcal{R}(E)} Av_2v_1E \xrightarrow[1]{\mathcal{W}(v_3)}$
$v_4 \leftarrow v_1 \oplus C;$	$v_2v_1Ev_3 \xrightarrow[0]{\mathcal{H}(v_1)} v_2Ev_3v_1 \xrightarrow[2]{\mathcal{R}(C)} Ev_3v_1C \xrightarrow[1]{\mathcal{W}(v_4)}$
return(v_2, v_3, v_4);	$v_3v_1Cv_4 \Longrightarrow \#_{\text{I/O}}(4, P) = 20.$

First approach: Register Assignment

Idea: Reducing the number of variables can relax the pressure of cache, and thus may reduce $\#_{\mathrm{I/O}}$.

We do Recycling variables by *Register assignment*.

First approach: Register Assignment

Idea: Reducing the number of variables can relax the pressure of cache, and thus may reduce $\#_{\mathsf{I/O}}$.

We do Recycling variables by *Register assignment*.

$$
\begin{array}{ll}
 & \#_{\mathsf{I/O}} \\
v_1 \leftarrow A \oplus B; & [5] \\
v_2 \leftarrow \oplus(E, D, A); & [7] \\
v_3 \leftarrow v_1 \oplus E; & [5] \\
v_4 \leftarrow v_1 \oplus C; & [3] \\
\text{return}(v_2, v_3, v_4); &
\end{array}
\xRightarrow{\text{Register assignment}}
\begin{array}{ll}
 & \#_{\mathsf{I/O}} \\
v_1 \leftarrow A \oplus B; & [5] \\
v_2 \leftarrow \oplus(E, D, A); & [7] \\
v_3 \leftarrow v_1 \oplus E; & [5] \\
v_1 \leftarrow v_1 \oplus C; & [2] \\
\text{return}(v_2, v_3, v_1); &
\end{array}
$$

First approach: Register Assignment

Idea: Reducing the number of variables can relax the pressure of cache, and thus may reduce $\#_{\text{I/O}}$.

We do Recycling variables by *Register assignment*.

$$
\begin{array}{lc}
 & \#_{\text{I/O}} \\
v_1 \leftarrow A \oplus B; & [5] \\
v_2 \leftarrow \oplus(E, D, A); & [7] \\
v_3 \leftarrow v_1 \oplus E; & [5] \\
v_4 \leftarrow v_1 \oplus C; & [3] \\
\text{return}(v_2, v_3, v_4); &
\end{array}
\xRightarrow{\text{Register assignment}}
\begin{array}{lc}
 & \#_{\text{I/O}} \\
v_1 \leftarrow A \oplus B; & [5] \\
v_2 \leftarrow \oplus(E, D, A); & [7] \\
v_3 \leftarrow v_1 \oplus E; & [5] \\
v_1 \leftarrow v_1 \oplus C; & [2] \\
\text{return}(v_2, v_3, v_1); &
\end{array}
$$

$$
\xrightarrow[0]{\mathcal{R}(v_1)} v_2 E v_3 v_1 \xrightarrow[2]{\mathcal{R}(C)} E v_3 v_1 C \xrightarrow[1]{\mathcal{W}(v_4)} v_3 v_1 C v_4
$$

$$
\Downarrow
$$

$$
\xrightarrow[0]{\mathcal{R}(v_1)} v_2 E v_3 v_1 \xrightarrow[2]{\mathcal{R}(C)} E v_3 v_1 C \xrightarrow[0]{\mathcal{W}(v_1)} E v_3 C v_1
$$

It works, but the effect is so limited.

Next Approach: Reordering Statements and Arguments

No side effects on SLPs; thus, we can reorder statements and arguments.

	#I/O			#I/O
$v_1 \leftarrow A \oplus B;$	[5]		$v_2 \leftarrow \oplus(A, D, E);$	[5]
$v_2 \leftarrow \oplus(E, D, A);$	[7]	$\xRightarrow{\text{Reordering}}$	$v_1 \leftarrow A \oplus B;$	[3]
$v_3 \leftarrow v_1 \oplus E;$	[5]		$v_3 \leftarrow v_1 \oplus E;$	[3]
$v_4 \leftarrow v_1 \oplus C;$	[3]		$v_4 \leftarrow v_1 \oplus C;$	[3]
return$(v_2, v_3, v_4);$	20		return$(v_2, v_3, v_4);$	14

Next Approach: Reordering Statements and Arguments

No side effects on SLPs; thus, we can reorder statements and arguments.

$$
\begin{array}{ll}
 & \#\text{I/O} \\
v_1 \leftarrow A \oplus B; & [5] \\
v_2 \leftarrow \oplus(E, D, A); & [7] \\
v_3 \leftarrow v_1 \oplus E; & [5] \\
v_4 \leftarrow v_1 \oplus C; & [3] \\
\text{return}(v_2, v_3, v_4); & 20
\end{array}
\xRightarrow{\text{Reordering}}
\begin{array}{ll}
 & \#\text{I/O} \\
v_2 \leftarrow \oplus(A, D, E); & [5] \\
v_1 \leftarrow A \oplus B; & [3] \\
v_3 \leftarrow v_1 \oplus E; & [3] \\
v_4 \leftarrow v_1 \oplus C; & [3] \\
\text{return}(v_2, v_3, v_4); & 14
\end{array}
$$

Using *Pebble Game*, we can integrate $\left\{ \begin{array}{l} \text{Recycling Variables and} \\ \text{Reordering} \end{array} \right.$

⋆ R. Sethi, 1975, *Complete register allocation problems*.

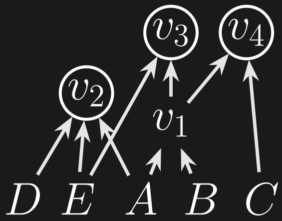
Next Approach: Reordering Statements and Arguments

No side effects on SLPs; thus, we can reorder statements and arguments.

$$
\begin{array}{lc}
 & \#\text{I/O} \\
v_1 \leftarrow A \oplus B; & [5] \\
v_2 \leftarrow \oplus(E, D, A); & [7] \\
v_3 \leftarrow v_1 \oplus E; & [5] \\
v_4 \leftarrow v_1 \oplus C; & [3] \\
\text{return}(v_2, v_3, v_4); & 20
\end{array}
\quad \xRightarrow{\text{Reordering}} \quad
\begin{array}{lc}
 & \#\text{I/O} \\
v_2 \leftarrow \oplus(A, D, E); & [5] \\
v_1 \leftarrow A \oplus B; & [3] \\
v_3 \leftarrow v_1 \oplus E; & [3] \\
v_4 \leftarrow v_1 \oplus C; & [3] \\
\text{return}(v_2, v_3, v_4); & 14
\end{array}
$$

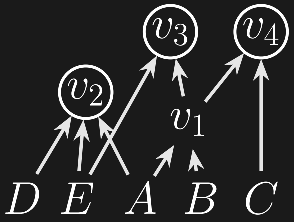
Using *Pebble Game*, we can integrate $\left\{ \begin{array}{l} \text{Recycling Variables and} \\ \text{Reordering} \end{array} \right.$

- ★ R. Sethi, 1975, *Complete register allocation problems*.
- ▶ We play the pebble game on DAGs or abstract syntax graphs.
- ▶ We aim to put pebbles in return nodes.



Pebble Game & Intractability of Optimization Problem

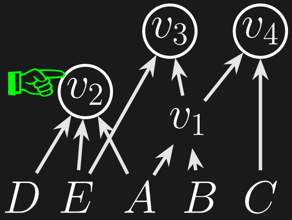
Playing Pebble Game = Deciding Evaluation Order + Variable Recycling



Example: Evaluating strategy based on Depth-first-search

Pebble Game & Intractability of Optimization Problem

Playing Pebble Game = Deciding Evaluation Order + Variable Recycling



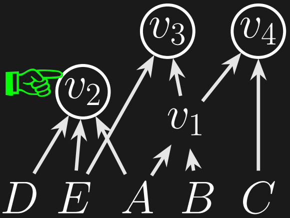
v_2 :

Example: Evaluating strategy based on Depth-first-search

1. Choose v_2 from unvisited roots: alphabetical small $v_2 \prec v_3 \prec v_4$.

Pebble Game & Intractability of Optimization Problem

Playing Pebble Game = Deciding Evaluation Order + Variable Recycling



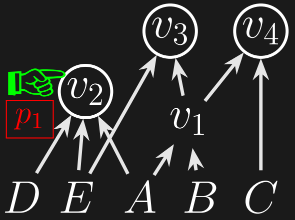
$v_2 : \quad \leftarrow \oplus(A, D, E);$

Example: Evaluating strategy based on Depth-first-search

1. Choose v_2 from unvisited roots: alphabetical small $v_2 \prec v_3 \prec v_4$.
2. Evaluate the children of v_2 in alphabetical order.

Pebble Game & Intractability of Optimization Problem

Playing Pebble Game = Deciding Evaluation Order + Variable Recycling



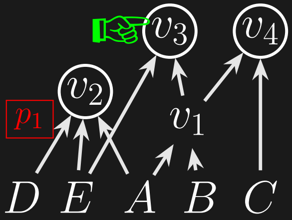
$$v_2: \quad p_1 \leftarrow \oplus(A, D, E);$$

Example: Evaluating strategy based on Depth-first-search

1. Choose v_2 from unvisited roots: alphabetical small $v_2 \prec v_3 \prec v_4$.
2. Evaluate the children of v_2 in alphabetical order.
3. Put a pebble p_1 on v_2 to denote v_2 is visited.

Pebble Game & Intractability of Optimization Problem

Playing Pebble Game = Deciding Evaluation Order + Variable Recycling



v_2 : $p_1 \leftarrow \oplus(A, D, E);$

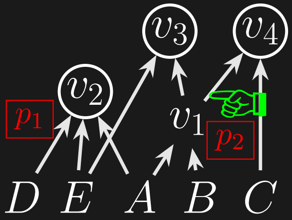
v_3 : $\leftarrow E \oplus$

Example: Evaluating strategy based on Depth-first-search

1. Choose v_2 from unvisited roots: alphabetical small $v_2 \prec v_3 \prec v_4$.
2. Evaluate the children of v_2 in alphabetical order.
3. Put a pebble p_1 on v_2 to denote v_2 is visited.
4. Choose v_3 from 2 unvisited roots, and first visit E.

Pebble Game & Intractability of Optimization Problem

Playing Pebble Game = Deciding Evaluation Order + Variable Recycling



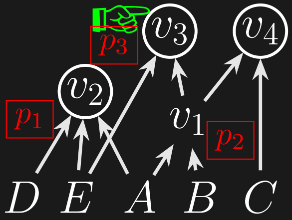
$$
\begin{aligned}
v_2 :&\quad p_1 \leftarrow \oplus(A, D, E);\\
v_1 :&\quad p_2 \leftarrow A \oplus B;\\
v_3 :&\quad \leftarrow E \oplus
\end{aligned}
$$

Example: Evaluating strategy based on Depth-first-search

1. Choose v_2 from unvisited roots: alphabetical small $v_2 \prec v_3 \prec v_4$.
2. Evaluate the children of v_2 in alphabetical order.
3. Put a pebble p_1 on v_2 to denote v_2 is visited.
4. Choose v_3 from 2 unvisited roots, and first visit E.
5. Visit the unvisited child v_1 of v_3, evaluate, and pebble p_2

Pebble Game & Intractability of Optimization Problem

Playing Pebble Game = Deciding Evaluation Order + Variable Recycling



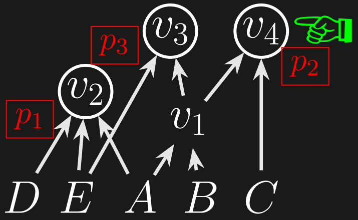
$$\begin{aligned} v_2 &: \quad p_1 \leftarrow \oplus(A, D, E); \\ v_1 &: \quad p_2 \leftarrow A \oplus B; \\ v_3 &: \quad p_3 \leftarrow E \oplus p_2; \end{aligned}$$

Example: Evaluating strategy based on Depth-first-search

1. Choose v_2 from unvisited roots: alphabetical small $v_2 \prec v_3 \prec v_4$.
2. Evaluate the children of v_2 in alphabetical order.
3. Put a pebble p_1 on v_2 to denote v_2 is visited.
4. Choose v_3 from 2 unvisited roots, and first visit E.
5. Visit the unvisited child v_1 of v_3, evaluate, and pebble p_2
6. Back to v_3 and pebble p_3

Pebble Game & Intractability of Optimization Problem

Playing Pebble Game = Deciding Evaluation Order + Variable Recycling



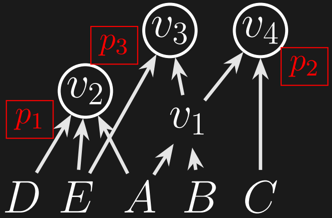
$$
\begin{aligned}
v_2 &: \quad p_1 \leftarrow \oplus(A, D, E);\\
v_1 &: \quad p_2 \leftarrow A \oplus B;\\
v_3 &: \quad p_3 \leftarrow E \oplus p_2;\\
v_4 &: \quad p_2 \leftarrow C \oplus p_2;
\end{aligned}
$$

Example: Evaluating strategy based on Depth-first-search

1. Choose v_2 from unvisited roots: alphabetical small $v_2 \prec v_3 \prec v_4$.
2. Evaluate the children of v_2 in alphabetical order.
3. Put a pebble p_1 on v_2 to denote v_2 is visited.
4. Choose v_3 from 2 unvisited roots, and first visit E.
5. Visit the unvisited child v_1 of v_3, evaluate, and pebble p_2
6. Back to v_3 and pebble p_3
7. Finally, we compute v_4 with *moving/recycling* pebble p_2.

Pebble Game & Intractability of Optimization Problem

Playing Pebble Game = Deciding Evaluation Order + Variable Recycling



		$\#_{I/O}$
v_2 :	$p_1 \leftarrow \oplus(A, D, E);$	[7]
v_1 :	$p_2 \leftarrow A \oplus B;$	[3]
v_3 :	$p_3 \leftarrow E \oplus p_2;$	[3]
v_4 :	$p_2 \leftarrow C \oplus p_2;$	[2]
	return$(p_1, p_3, p_2);$	15

Example: Evaluating strategy based on Depth-first-search

Can we find the best reordering and pebbling in polynomial time?

Theorem (Sethi 1975, Papp & Wattenhofer 2020)

Unless **P = NP**, *for a given* P, *in polynomial time,* *we cannot find a* Q *that* $[\![P]\!] = [\![Q]\!]$ *and minimizes* $\#_{I/O}(Q)$.

We use DFS-based strategy as above in our evaluation.

Evaluation

Data Set & Evaluation Environment

We consider RS(10, 4) as an example data set.

- We have 1-encoding SLP P_{enc}.
- We have $\binom{14}{4} = 1001$ decoding SLPs.

We used two environments in my paper:

name	CPU	Clock	Core	RAM
intel	i7-7567U	4.0GHz	2	DDR3-2133 16GB
amd	Ryzen 2600	3.9GHz	6	DDR4-2666 48GB

In a distributed computation,
our test environments correspond to single nodes.

L1 cache specification:

Size	Associativity	Line Size
32KB/core	8-way	64 bytes

Improvements by heuristics for the encoding SLP on Intel PC

Throughput is Avg. of 1000-runs for 10MB randomly generated data

Metric	Base P_{enc}	RePair	RePair + Fuse	RePair + Fuse + Pebbling
$\#_{\oplus}$	755			
$\#_{\text{mem}}$	2265			

Improvements by heuristics for the encoding SLP on Intel PC

Throughput is Avg. of 1000-runs for 10MB randomly generated data

	Metric	Base P_{enc}	RePair	RePair + Fuse	RePair + Fuse + Pebbling
	$\#_{\oplus}$	755			
	$\#_{\mathsf{mem}}$	2265			
$\mathcal{B} = 512$:	$\#_{\mathsf{I/O}}(K = 64)$	570			
$\mathcal{B} = 1\mathsf{K}$:	$\#_{\mathsf{I/O}}(K = 32)$	1262			
$\mathcal{B} = 2\mathsf{K}$:	$\#_{\mathsf{I/O}}(K = 16)$	1598			

Improvements by heuristics for the encoding SLP on Intel PC

Throughput is Avg. of 1000-runs for 10MB randomly generated data

$\mathcal{B}$-Byte Blocking for Cache Efficiency

$$
\begin{array}{l}
v_1 = \mathtt{xor}(A, B);\\
v_2 = \mathtt{xor}(v_1, C, D);\\
\mathtt{return}(v_1, v_2);
\end{array}
\quad \Longrightarrow \quad
\begin{array}{l}
\mathtt{for}\ i \leftarrow 0 \mathrel{..} (A\mathtt{.len} / \mathcal{B})\ \{\\
\quad v_1^{[i]} = \mathtt{xor}(A^{[i]}, B^{[i]});\\
\quad v_2^{[i]} = \mathtt{xor}(v_1^{[i]}, C^{[i]}, D^{[i]});\\
\ \}\\
\mathtt{return}(v_1, v_2);
\end{array}
$$

where $A^{[i]}$ is the i-th $\mathcal{B}$-bytes block.

		Base
$\mathcal{B} =$		
$\mathcal{B} =$		
$\mathcal{B} = 2\text{K}$:	$\#_{\text{I/O}}(K = 16)$	1598

Improvements by heuristics for the encoding SLP on Intel PC

Throughput is Avg. of 1000-runs for 10MB randomly generated data

	Metric	Base P_{enc}	RePair	RePair + Fuse	RePair + Fuse + Pebbling
	$\#_{\oplus}$	755			
	$\#_{\text{mem}}$	2265			
$\mathcal{B} = 512$:	$\#_{\text{I/O}}(K = 64)$	570			
	Throughput (GB/s)	3.10			
$\mathcal{B} = 1\text{K}$:	$\#_{\text{I/O}}(K = 32)$	1262			
	Throughput (GB/s)	4.03			
$\mathcal{B} = 2\text{K}$:	$\#_{\text{I/O}}(K = 16)$	1598			
	Throughput (GB/s)	4.45			

Improvements by heuristics for the encoding SLP on Intel PC

Throughput is Avg. of 1000-runs for 10MB randomly generated data

	Metric	Base P_{enc}	RePair	RePair + Fuse	RePair + Fuse + Pebbling
	$\#_{\oplus}$	755			
	$\#_{\text{mem}}$	2265			
$\mathcal{B} = 512$:	$\#_{\text{I/O}}(K = 64)$	570			
	Throughput (GB/s)	3.10			
$\mathcal{B} = 1\text{K}$:	$\#_{\text{I/O}}(K = 32)$	1262			
	Throughput (GB/s)	4.03			
$\mathcal{B} = 2\text{K}$:	$\#_{\text{I/O}}(K = 16)$	1598			
	Throughput (GB/s)	4.45			

Why smaller blocks are slower than the large one?

Pros: Smaller blocks,

- More cache-able blocks $\frac{32K}{\mathcal{B}}$.

Cons: Smaller blocks,

- Due to cache conflicts, using cache identically is more difficult.
- Latency penalty becomes totally large.

Improvements by heuristics for the encoding SLP on Intel PC

Throughput is Avg. of 1000-runs for 10MB randomly generated data

	Metric	Base P_{enc}	RePair	RePair + Fuse	RePair + Fuse + Pebbling
	$\#_{\oplus}$	755	385		
	$\#_{\text{mem}}$	2265	1155		
$\mathcal{B} = 512$:	$\#_{\text{I/O}}(K = 64)$	570			
	Throughput (GB/s)	3.10			
$\mathcal{B} = 1\text{K}$:	$\#_{\text{I/O}}(K = 32)$	1262			
	Throughput (GB/s)	4.03			
$\mathcal{B} = 2\text{K}$:	$\#_{\text{I/O}}(K = 16)$	1598			
	Throughput (GB/s)	4.45			

Improvements by heuristics for the encoding SLP on Intel PC

Throughput is Avg. of 1000-runs for 10MB randomly generated data

	Metric	Base P_{enc}	RePair	RePair + Fuse	RePair + Fuse + Pebbling
	$\#_{\oplus}$	755	385		
	$\#_{\text{mem}}$	2265	1155		
$\mathcal{B} = 512$:	$\#_{\text{I/O}}(K = 64)$	570	1231		
	Throughput (GB/s)	3.10			
$\mathcal{B} = 1\text{K}$:	$\#_{\text{I/O}}(K = 32)$	1262	1465		
	Throughput (GB/s)	4.03			
$\mathcal{B} = 2\text{K}$:	$\#_{\text{I/O}}(K = 16)$	1598	1599		
	Throughput (GB/s)	4.45			

Improvements by heuristics for the encoding SLP on Intel PC

Throughput is Avg. of 1000-runs for 10MB randomly generated data

	Metric	Base P_{enc}	RePair	RePair + Fuse	RePair + Fuse + Pebbling
	$\#_{\oplus}$	755	385		
	$\#_{\text{mem}}$	2265	1155		
$\mathcal{B} = 512$:	$\#_{\text{I/O}}(K = 64)$	570	1231		
	Throughput (GB/s)	3.10	4.18		
$\mathcal{B} = 1\text{K}$:	$\#_{\text{I/O}}(K = 32)$	1262	1465		
	Throughput (GB/s)	4.03	4.36		
$\mathcal{B} = 2\text{K}$:	$\#_{\text{I/O}}(K = 16)$	1598	1599		
	Throughput (GB/s)	4.45	4.86		

Improvements by heuristics for the encoding SLP on Intel PC

Throughput is Avg. of 1000-runs for 10MB randomly generated data

	Metric	Base P_{enc}	RePair	RePair + Fuse	RePair + Fuse + Pebbling
	$\#_{\oplus}$	755	385	N/A	
	$\#_{\text{mem}}$	2265	1155	677	
$\mathcal{B} = 512$:	$\#_{\text{I/O}}(K = 64)$	570	1231		
	Throughput (GB/s)	3.10	4.18		
$\mathcal{B} = 1\text{K}$:	$\#_{\text{I/O}}(K = 32)$	1262	1465		
	Throughput (GB/s)	4.03	4.36		
$\mathcal{B} = 2\text{K}$:	$\#_{\text{I/O}}(K = 16)$	1598	1599		
	Throughput (GB/s)	4.45	4.86		

Improvements by heuristics for the encoding SLP on Intel PC

Throughput is Avg. of 1000-runs for 10MB randomly generated data

	Metric	Base P_{enc}	RePair	RePair + Fuse	RePair + Fuse + Pebbling
	$\#_{\oplus}$	755	385	N/A	
	$\#_{\text{mem}}$	2265	1155	677	
$\mathcal{B} = 512$:	$\#_{\text{I/O}}(K = 64)$	570	1231	936	
	Throughput (GB/s)	3.10	4.18		
$\mathcal{B} = 1\text{K}$:	$\#_{\text{I/O}}(K = 32)$	1262	1465	1086	
	Throughput (GB/s)	4.03	4.36		
$\mathcal{B} = 2\text{K}$:	$\#_{\text{I/O}}(K = 16)$	1598	1599	1144	
	Throughput (GB/s)	4.45	4.86		

Improvements by heuristics for the encoding SLP on Intel PC

Throughput is Avg. of 1000-runs for 10MB randomly generated data

	Metric	Base P_{enc}	RePair	RePair + Fuse	RePair + Fuse + Pebbling
	$\#_{\oplus}$	755	385	N/A	
	$\#_{\text{mem}}$	2265	1155	677	
$\mathcal{B} = 512$:	$\#_{\text{I/O}}(K = 64)$	570	1231	936	
	Throughput (GB/s)	3.10	4.18	6.98	
$\mathcal{B} = 1\text{K}$:	$\#_{\text{I/O}}(K = 32)$	1262	1465	1086	
	Throughput (GB/s)	4.03	4.36	7.50	
$\mathcal{B} = 2\text{K}$:	$\#_{\text{I/O}}(K = 16)$	1598	1599	1144	
	Throughput (GB/s)	4.45	4.86	7.12	

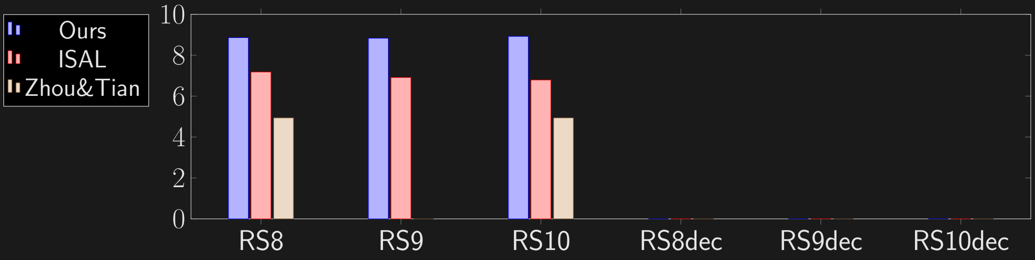
Improvements by heuristics for the encoding SLP on Intel PC

Throughput is Avg. of 1000-runs for 10MB randomly generated data

	Metric	Base P_{enc}	RePair	RePair + Fuse	RePair + Fuse + Pebbling
	$\#_{\oplus}$	755	385	N/A	
	$\#_{\mathsf{mem}}$	2265	1155	677	
$\mathcal{B} = 512$:	$\#_{\mathsf{I/O}}(K = 64)$	570	1231	936	636
	Throughput (GB/s)	3.10	4.18	6.98	7.24
$\mathcal{B} = 1\mathrm{K}$:	$\#_{\mathsf{I/O}}(K = 32)$	1262	1465	1086	779
	Throughput (GB/s)	4.03	4.36	7.50	8.92
$\mathcal{B} = 2\mathrm{K}$:	$\#_{\mathsf{I/O}}(K = 16)$	1598	1599	1144	845
	Throughput (GB/s)	4.45	4.86	7.12	8.55

Throughput Comparison (Intel + 1K-Blocking)

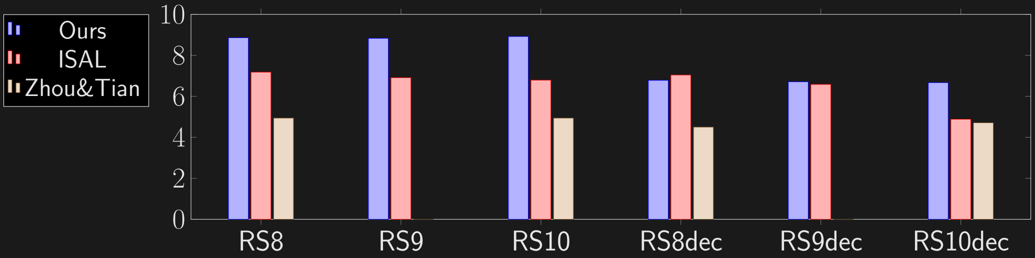
Enc	$\#_{\mathsf{mem}}$	$\#_{\mathsf{I/O}}$	Ours	ISA-L v2.30	Zhou & Tian
RS$(8, 4)$	543	585	8.86 GB/s	7.18 GB/s	4.94 GB/s
RS$(9, 4)$	611	671	8.83	6.91	N/A in their paper
RS$(10, 4)$	677	779	8.92	6.79	4.94



Throughput Comparison (Intel + 1K-Blocking)

Enc	$\#_{\text{mem}}$	$\#_{\text{I/O}}$	Ours	ISA-L v2.30	Zhou & Tian
RS$(8,4)$	543	585	8.86 GB/s	7.18 GB/s	4.94 GB/s
RS$(9,4)$	611	671	8.83	6.91	N/A in their paper
RS$(10,4)$	677	779	8.92	6.79	4.94

Dec	$\#_{\text{mem}}$	$\#_{\text{I/O}}$	Ours	ISA-L v2.30	Zhou & Tian
RS$(8,4)$	747	811	6.78 GB/s	7.04 GB/s	4.50 GB/s
RS$(9,4)$	829	968	6.71	6.58	N/A
RS$(10,4)$	923	1077	6.67	4.88	4.71



Conclusion (+ Other Throughput Scores)

intel 1K (GB/sec)	Ours Enc	Ours Dec	ISA-L v 2.30 Enc	ISA-L v 2.30 Dec	Zhou & Tian Enc	Zhou & Tian Dec
RS$(8,3)$	12.32	8.82	9.09	9.25	6.08	5.57
RS$(9,3)$	11.97	8.27	7.31	7.92	6.17	5.66
RS$(10,3)$	11.78	8.89	6.78	7.93	6.15_S	5.90
RS$(8,2)$	18.79	14.59	12.99	13.34	8.13_E	8.07_E
RS$(9,2)$	18.93	14.27	11.85	12.03	8.34_E	8.04
RS$(10,2)$	18.98	14.66	12.12	12.61	8.40_E	8.22_E

Conclusion

- We identified bitmatrix multiplication as straight line programs (SLP).
- We optimized XOR-based EC by optimizing SLPs using various program optimization techniques.
- Each of our techniques is not difficult; however, it suffices to match Intel's high performance library ISAL.
- As future work on cache optimization, I plan to accommodate multi-layer cache L1, L2, and L3 cache.