
# Accelerating XOR-Based Erasure Coding using Program Optimization Techniques

Yuya Uezato
Dwango, Co., Ltd.
Japan
yuuya_uezato@dwango.co.jp

## ABSTRACT

Erasure coding (EC) affords data redundancy for large-scale systems. XOR-based EC is an easy-to-implement method for optimizing EC. This paper addresses a significant performance gap between the state-of-the-art XOR-based EC approach (~4.9 GB/s coding throughput) and Intel's high-performance EC library based on another approach (~6.7 GB/s). We propose a novel approach based on our observation that XOR-based EC virtually generates programs of a Domain Specific Language for XORing byte arrays. We formalize such programs as straight-line programs (SLPs) of compiler construction and optimize SLPs using various program optimization techniques. Our optimization flow is three-fold: 1) reducing the number of XORs using grammar compression algorithms; 2) reducing memory accesses using deforestation, a functional program optimization method; and 3) reducing cache misses using the (red-blue) pebble game of program analysis. We provide an experimental library, which outperforms Intel's library with an ~8.92 GB/s throughput.

## CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Context specific languages**; • **Mathematics of computing** → *Coding theory*; • **Computer systems organization** → *Redundancy*.

**ACM Reference Format:**
Yuya Uezato. 2021. Accelerating XOR-Based Erasure Coding using Program Optimization Techniques. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3458817.3476204

## 1 INTRODUCTION

Ensuring data redundancy is the most critical task for large-scale systems such as distributed storage. Replication—distributing the replicas of data—is the simplest solution. Erasure coding (EC) has attracted significant attention thanks to its space efficiency [102]. For example, the famous distributed system HDFS (Hadoop Distributed File System) [8] offers the codec **RS**(10, 4), Reed-Solomon EC [84] with 10 data blocks and 4 parity blocks. On **RS**(10, 4), we can store 10-times more objects than through replication; however, we cannot recover data if five nodes are down. Another distributed system Ceph [24] offers **RS**$(n, p)$ for any $n$ and $p$. On Linux, we can use RAID-6, a codec similar to **RS**$(n, 2)$ [11, 82]. Using EC instead of replication degrades the system performance since the encoding and decoding of EC are heavy computation and are required for each storing to and loading from a system. It is often stated that EC is suitable only for archiving cold (rarely accessed) data [29, 49, 93].

We clarify the pros and cons of EC by observing how **RS** works. To encode data using matrix multiplication (hereafter we use the acronym MM), **RS** adopts matrices over $\mathbb{F}_{2^8}$, the finite field with $2^8 = 256$ elements. Since each element of $\mathbb{F}_{2^8}$ is coded by one byte (8 bits), we can identify an $N$-bytes data as an $N$-elements array of $\mathbb{F}_{2^8}$. **RS**$(n, p)$ encodes an $N$-bytes data $D$ using an $(n + p) \times n$ *Vandermonde* matrix $\mathcal{V} \in \mathbb{F}_{2^8}^{(n+p)\times n}$, which is crucial for decoding as we will see below, as follows:

$$\begin{matrix} n \\ n+p \end{matrix}\left(\begin{array}{c} n \\ \mathcal{V} \end{array}\right) \cdot_{\mathbb{F}_{2^8}} \begin{pmatrix} \vec{d}_1 \\ \vdots \\ \vec{d}_n \end{pmatrix} = \begin{pmatrix} \vec{b}_1 \\ \vdots \\ \vec{b}_n \\ \vdots \\ \vec{b}_{n+p} \end{pmatrix}$$

where
$\cdot_{\mathbb{F}_{2^8}}$ is the MM over $\mathbb{F}_{2^8}$;
$\vec{d}_i$ is $i$-th $\frac{N}{n}$-bytes block of $D$;
$\vec{b}_j$ is an $\frac{N}{n}$-bytes coded block.

We store an encoded block $\vec{b}_i$ to a node $n_i$ of a system with $n + p$ nodes. For decoding, we gather $n$-blocks $B = (\vec{b}_{i_1} \vec{b}_{i_2} \dots \vec{b}_{i_n})^T$ from alive nodes. The $(n \times n)$-submatrix $\mathcal{M}$ of $\mathcal{V}$ obtained by extracting row-vectors at $\{i_1, i_2, \dots, i_n\}$ satisfies $B = \mathcal{M} \cdot_{\mathbb{F}_{2^8}} D$. Since any square submatrix of Vandermonde matrices is invertible [65, 73, 94], the inverse $\mathcal{M}^{-1}$ of $\mathcal{M}$ recovers $D$ as $\mathcal{M}^{-1} \cdot_{\mathbb{F}_{2^8}} B = D$.

Now, the advantage of **RS** *space efficiency* emerges as the size of encoded blocks. For example, on **RS**(10, 4), nodes of a system require $\frac{N}{10}$-bytes of space for an $N$-bytes data. On the other hand, the disadvantage *slowness* results from MM over finite fields. Multiplying $n \times n$ matrices requires $\sim O(n^{2.37287})$ field operations even when using the latest result [5, 62]. Moreover, finite field multiplication $\times_{\mathbb{F}_k}$ is computationally expensive, and its optimization is an active research area [50, 57, 59, 66, 83].

There are two primary acceleration methods of **RS**.
**(1)** Tightly coupling sophisticated optimization methods for MM and finite field multiplication. Intel provides an EC library, ISA-L (Intelligent Storage Acceleration Library), based on this approach [52]. ISA-L is exceptionally optimized for MM over $\mathbb{F}_{2^8}$ and offers different assembly codes for each platform to maximize the performance of SIMD instructions [7, 10, 56]. Intel reported ISA-L scored about **6.0 GB/s** encoding throughput for **RS**(10,4) in [54]. In our evaluation at §7, ISA-L scores about **6.7 GB/s**.


*SC '21, November 14–19, 2021, St. Louis, MO, USA*

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476204

**(2)** XOR-based EC [13, 74, 104] converts MM over $\mathbb{F}_{2^8}$ to MM over $\mathbb{F}_2$ where $\mathbb{F}_2$ is the finite field of the bits $\{0, 1\}$. This approach depends on the following two properties:

(1) There is an isomorphism $\mathfrak{B} : \mathbb{F}_{2^8} \cong \mathbb{F}_2^{8\times 1}$ between bytes and 8-bits column vectors;
(2) There is a function $\widetilde{\cdot} : \mathbb{F}_{2^8} \to \mathbb{F}_2^{8\times 8}$ from bytes to 8×8 matrices over $\mathbb{F}_2$ such that: $\forall x, y \in \mathbb{F}_{2^8}.\ x \times_{\mathbb{F}_{2^8}} y = \mathfrak{B}^{-1}(\widetilde{x} \cdot_{\mathbb{F}_2} \mathfrak{B}(y))$.

We can calculate the above $\mathcal{V} \cdot_{\mathbb{F}_{2^8}} D$ without the finite field multiplication of $\mathbb{F}_{2^8}$, $\times_{\mathbb{F}_{2^8}}$, extending $\mathfrak{B}$ and $\widetilde{\cdot}$ to matrices as follows:

$$\mathcal{V} \cdot_{\mathbb{F}_{2^8}} D = \mathfrak{B}^{-1}(\widetilde{\mathcal{V}} \cdot_{\mathbb{F}_2} \mathfrak{B}(D)).$$

Since the addition (resp. multiplication) of $\mathbb{F}_2$ is the bit XOR $x \oplus y$ (resp. bit AND), MM over $\mathbb{F}_2$ is just array XORs, as presented below:

$$\begin{pmatrix} 1100000 \\ 0011110 \\ 0011101 \end{pmatrix} \cdot_{\mathbb{F}_2} \begin{pmatrix} \vec{x}_1 \\ \vdots \\ \vec{x}_7 \end{pmatrix} = \begin{pmatrix} \vec{x}_1 \oplus \vec{x}_2 \\ \vec{x}_3 \oplus \vec{x}_4 \oplus \vec{x}_5 \oplus \vec{x}_6 \\ \vec{x}_3 \oplus \vec{x}_4 \oplus \vec{x}_5 \oplus \vec{x}_7 \end{pmatrix} \quad \text{where } \vec{x}_i \text{ is a bitvector.}$$

Namely, MM over $\mathbb{F}_2$ is easy-to-implement. Thanks to its implementability, this method was used in VLSI to realize finite field arithmetic on a small circuit [74]. This method is currently receiving attention since XORing byte arrays is quickly executed via recent SIMD instructions [83]. In exchange for the ease of implementation, the obtained coding matrix $\widetilde{\mathcal{V}} \in \mathbb{F}_2^{8a\times 8b}$ is much larger than an original coding matrix $\mathcal{V} \in \mathbb{F}_{2^8}^{a\times b}$; thus, $\widetilde{\mathcal{V}}$ needs more (but simple) operations of $\mathbb{F}_2$ than those of $\mathbb{F}_{2^8}$ in $\mathcal{V}$.

Recently, Zhou and Tian published an invaluable study [104] that synthesized several acceleration methods for executing array XORs. It is the state-of-the-art study based on XOR-based EC; however, it scored **4.9 GB/s** for **RS**(10, 4) encoding.

Now, we have a question: "Is XOR-based EC essentially slower than the former approach in exchange for the ease of implementation?". The answer is "No". We provide a streamlined method to optimize XOR-based EC by employing various program optimization techniques. We also implement and provide an experimental EC library outperforming ISA-L.

## 2 OUR APPROACH AND CONTRIBUTION

We identify the MM over $\mathbb{F}_2$, $\cdot_{\mathbb{F}_2}$, as *straight-line programs* (SLPs), a classical compiler theory tool [2, 3], as follows:

$$\begin{pmatrix} 1100000 \\ 0011110 \\ 0011101 \end{pmatrix} \cdot_{\mathbb{F}_2} \begin{pmatrix} \vec{a} \\ \vec{b} \\ \vdots \\ \vec{g} \end{pmatrix} \Longmapsto P : \begin{array}{l} v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow c \oplus d \oplus e \oplus f; \\ v_3 \leftarrow c \oplus d \oplus e \oplus g; \\ \mathsf{ret}(v_1, v_2, v_3) \end{array}$$

where $a, b, \ldots, g$ are constants meaning input arrays, and $v_1, v_2, v_3$ are variables meaning arrays allocated at runtime. SLPs are programs with a single binary operator without branchings, loops, and functions as above.

Replacing the MM over $\mathbb{F}_2$ by SLPs is a simple but key idea for importing various optimization methods from programming language theory. This is the crucial difference between our study and that of Zhou and Tian [104], where they treated topics directly on matrices of $\mathbb{F}_2$ and introduced ad-hoc constructions without sophisticated results of program optimization.

***Our Goal.*** This paper aims to provide an efficient EC library by importing various programmer-friendly optimization methods from programming language theory. Technically, we implement our optimizer as a translator that converts an SLP to a more efficient one. When encoding and decoding data, we run optimized SLPs line-by-line in our host language in the interpreter style.

### 2.1 Idea and Contribution in Our Optimizer

We optimize SLPs via compressing, fusing, and scheduling. Let us see the idea of each step by optimizing the above $P$ as follows:

$$P \underset{comp.}{\Longmapsto} \begin{array}{l} \lambda \leftarrow c \oplus d \oplus e; \\ v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow \lambda \oplus f; \\ v_3 \leftarrow \lambda \oplus g; \\ \mathsf{ret}(v_1, v_2, v_3) \end{array} \underset{fuse}{\Longmapsto} \begin{array}{l} \lambda \leftarrow \bigoplus(c, d, e); \\ v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow \lambda \oplus f; \\ v_3 \leftarrow \lambda \oplus g; \\ \mathsf{ret}(v_1, v_2, v_3) \end{array} \underset{sched.}{\Longmapsto} \begin{array}{l} v_1 \leftarrow a \oplus b; \\ \lambda \leftarrow \bigoplus(c, d, e); \\ v_2 \leftarrow \lambda \oplus f; \\ \lambda \leftarrow \lambda \oplus g; \\ \mathsf{ret}(v_1, v_2, \lambda) \end{array}$$

*Compressing.* We use the compression algorithm RePair [67], which is used to compress context-free grammars (CFGs) in grammar compression. We can immediately adapt it by ignoring $\oplus$ of SLPs, and by identifying constants (resp. variables) of SLPs as terminals (resp. nonterminals) of CFGs.

RePair compresses a program (or CFG) by extracting its hidden repetition structures. For $P$, RePair extracts the repeatedly appearing subterm $c \oplus d \oplus e$ and replaces it with a new variable $\lambda$. It reduces the seven XORs to five and thus speeds up ∼30%.

We extend RePair to XorRePair by adding the XOR-cancellation property $(x \oplus x \oplus y = y)$, not considered in grammar compression.

*Fusing.* To reduce memory access, we employ a technique called deforestation in functional program optimization [101]. Deforestation eliminates intermediate data via fusing functions. Although it has a deep background theory, we can easily adapt it thanks to the simplicity of SLP, i.e., a single binary operator, no branchings, and no functions.

In our example, $c \oplus d \oplus e$ invokes six memory accesses because $c \oplus d$ invokes three for loading $c$ and $d$ and storing the result to an *intermediate* array $I_{c\oplus d}$, and then $I_{c\oplus d} \oplus e$ invokes three. Fusing $c \oplus d \oplus e$ to $\bigoplus(c, d, e)$, we eliminate (deforest) the intermediate array $I_{c\oplus d}$. The fused XOR only invokes four memory accesses: loading $c$, $d$, and $e$ and storing the result array.

*Scheduling.* To reduce cache misses, we revisit the well-known (but vague) maxim for cache optimization *increasing the locality of data access.* It appears in our example to reorder $\lambda$ and $v_1$ to adjust the generation site of $\lambda$ to the use sites, $\lambda \oplus f$ and $\lambda \oplus g$. Furthermore, we reuse $\lambda$ without allocating and accessing $v_3$.

The maxim for cache optimization is too vague to automatically optimize SLPs and incorporate it into our optimizer. Thus, in §6, we introduce measures for cache efficiency and concretize our optimization problem as reducing the measures of a given SLP. To formalize our cache optimization problem for SLPs, we employ the (red-blue) pebble game of program analysis [47, 91]. The game is a simple abstract model of computation with fast and slow devices. In our setting, the fast and slow devices are cache and main memory, respectively.

*Performance.* Each step improves coding performance as follows:

*Encoding Throughput Improvement on* **RS**(10, 4) (in §7 & §7.5)

| Base: | Comp. | In §4: | Fuse | In §5: | Sched. | In §6: |
|---|---|---|---|---|---|---|
| 4.03GB/s | → | 4.36GB/s | → | 7.50GB/s | → | 8.92GB/s |

where Base runs unoptimized SLPs that are obtained from matrices, such as the above $P$. Interestingly, although (Xor)RePair reduce about 60% XORs on average (as we will see in §7), the summary says the compressing effect is small. This is because (Xor)RePair generate cache-poor compressed SLPs, and this observation will be substantiated by cache efficiency analysis based on the pebble game. The sole application of (Xor)RePair is not good as the theoretical improvements suggest; however, compressing achieves excellent performance in combination with memory and cache optimization.

## 3 RELATED WORK

Zhou and Tian earnestly studied and evaluated various acceleration techniques for XOR-based EC [48, 72, 82] in [104]. Their study comprises two stages: (i) reducing XORs of bitmatrices (matrices over $\mathbb{F}_2$) [48, 82]; and (ii) reordering XORs for cache optimization [72]. We emphasize that the previous works [48, 72, 82]—and thus, Zhou and Tian—never considered SLPs, deforestation, and pebble games. The lack of considering SLP makes a problem in each stage. First, the XOR reducing heuristics of [48, 82] run on graphs, which are obtained in an ad-hoc manner from bitmatrices. This leads to a lack of considering the XOR-cancellation, unlike our XorRePair, and limited performance. Indeed, Zhou and Tian reported the average reducing ratio (the smaller the better) $\frac{\#\text{XOR of reduced}}{\#\text{XOR of original}}$ ∼65%. Their ratio is larger than ours—42.1% of RePair and 40.8% of XorRePair, as we will see in §7. Next, the cache optimization heuristics of [72], which reorder XORs locally without considering the pebble game, are not quite effective, $\frac{\text{throughput of optimized}}{\text{throughput of original}}$ ∼101% in [104]. In contrast, our heuristics for the scheduling problem are effective, with an improvement ratio of ∼125% in §7.

SLP has been widely studied in the early days of program optimization [1, 3, 21]. Recently, Boyar et al. revisited SLP with the XOR operator [17, 18] to optimize (compress) bitmatrices used in the field of cryptography, such as the AES S-box [85, 95, 97]. Their approach is based on Paar's heuristic [78], which is almost the same as RePair [67]. Previous works for cryptography [18, 60, 85, 97] focus on reducing XORs in such special SLPs even if spending several days on one SLP. Indeed, the proposed heuristics run in exponential time for aggressive optimization. However, for **RS**(10, 4), as we will see in §7, we need to optimize 1002 SLPs for encoding and decoding. On the basis of this difference, we propose the new heuristic XorRePair running in polynomial time. Although the work of Boyar et al. inspired the authors, we emphasize that they did not consider memory and cache optimization because their goal was to compress bitmatrices.

Hong and Kung proposed the red-blue pebble game [47] to model and study the transfer cost between fast and slow devices. This game has been mainly used to analyze a fixed algorithm rather than for program optimization. There is a recent remarkable work by Kwasniewski et al. where they used the pebble game to prove the near-optimality of their fixed MM algorithms [61]. Recently, there has been a trend to use the red-blue pebble game for program optimization [23, 35, 80]. Our work is in this direction; indeed, the pebble game is the basis of our cache optimization algorithm.

## 4 REDUCING NUMBER OF XORS

We formally introduce SLP with the XOR operator. To optimize SLP, we employ a compression algorithm, RePair, and extend it to XorRePair by accommodating a property of XOR. We will measure and compare the performance of RePair and XorRePair in §7.

### 4.1 Straight-Line Program

A straight-line program (SLP) is a program without branchings, loops, and functions [1, 3, 21]. An SLP is a tuple $\langle \mathscr{V}, \mathscr{C}, \vec{s}, \vec{g}, \otimes \rangle$ where $\mathscr{V}$ is a set of variables, $\mathscr{C}$ is a set of constants, $\vec{s}$ is a sequence of instructions (i.e., the body of the program), $\vec{g}$ is a sequence of variables returned by the program, and $\otimes$ is a binary operator. The set of instructions $\langle \text{instr} \rangle$ is defined by the following BNF:

$$\langle \text{instr} \rangle := \mathscr{V} \leftarrow \langle \text{expr} \rangle$$
$$\langle \text{expr} \rangle := \mathscr{V} \mid \mathscr{C} \mid \langle \text{expr} \rangle \otimes \langle \text{expr} \rangle$$

We consider a class of SLPs, *XOR SLP* [17, 18], whose binary operator only satisfies the associativity ($(x \oplus y) \oplus z = x \oplus (y \oplus z)$), commutativity ($x \oplus y = y \oplus x$), and cancellativity ($x \oplus x \oplus y = y$) laws. We write $\mathbb{SLP}_\oplus$ for the class.

$\mathbb{SLP}_\oplus$ is a DSL for XORing byte arrays. For example, the following left $\mathbb{SLP}_\oplus$ abstracts the right array program:

$v_1 \leftarrow a \oplus b;$
$v_2 \leftarrow b \oplus c \oplus d;$
$v_3 \leftarrow v_1 \oplus v_2;$
$\text{ret}(v_2, v_3, v_1)$

$\Leftrightarrow$

```
P(a, b, c, d: [byte]) {
  var v1 = a xor b;
  var v2 = (b xor c) xor d;
  var v3 = v1 xor v2;
  return (v2, v3, v1);
}
```

where for the left SLP $\mathscr{V} = \{v_1, v_2, v_3\}$, $\mathscr{C} = \{a, b, c, d\}$, and $\vec{g} = \langle v_2, v_3, v_1 \rangle$, and for the array program the infix function `xor` performs XOR element-wise for two byte arrays. On the basis of this idea, we have the following correspondence:

constants of $\mathbb{SLP}_\oplus$ ⇔ program input arrays,
variables of $\mathbb{SLP}_\oplus$ ⇔ arrays allocated at runtime.

*Calculus on* $\mathbb{SLP}_\oplus$. We consider a set-based semantics where the value of a variable is a set of constants. We interpret $\oplus$ as the symmetric difference of sets; e.g., $\{a, b\} \oplus \{c, d\} = \{a, b, c, d\}$ and $\{a, b\} \oplus \{a, c\} = \{b, c\}$. This semantics enables us to compute the above example $\mathbb{SLP}_\oplus$ as follows:

| SLP $P$ | $v_1$-value | $v_2$-value | $v_3$-value |
|---|---|---|---|
| $v_1 \leftarrow a \oplus b;$ | $\{a, b\}$ | | |
| $v_2 \leftarrow b \oplus c \oplus d;$ | $\{a, b\}$ | $\{b, c, d\}$ | |
| $v_3 \leftarrow v_1 \oplus v_2;$ | $\{a, b\}$ | $\{b, c, d\}$ | $\{a, c, d\}$ |
| $\text{ret}(v_2, v_3, v_1)$ | | | |

**Notation.** We use $[\![\cdot]\!]$ to denote the returned values of a program; e.g., $[\![P]\!] = \langle \{b, c, d\}, \{a, c, d\}, \{a, b\} \rangle$. We use $\#_\oplus \cdot$ to denote the size of a program, i.e., the number of XORs; e.g., $\#_\oplus P = 4$. We use $\text{NVar}(\cdot)$ to denote the number of variables; e.g., $\text{NVar}(P) = |\{v_1, v_2, v_3\}| = 3$ where $|S|$ is the cardinality of a finite set $S$.

### 4.2 Shortest SLP Problem

We formalize our first optimization problem.

**The shortest $\mathbb{SLP}_\oplus$ problem**
For a given $P \in \mathbb{SLP}_\oplus$, we find $Q \in \mathbb{SLP}_\oplus$ that satisfies $[\![P]\!] = [\![Q]\!]$ and minimizes $\#_\oplus Q$.

We cannot solve this problem in polynomial time unless **P=NP** since the NP-completeness of its decision problem version was shown by Boyar et al [18]. They reduced the NP-complete problem Vertex Cover Problem [39] to the above problem.

*Example: Minimizing via Cancellation.* Let us consider the following three equivalent SLPs:

| $\mathbb{SLP}_\oplus\ P_0$ | $\mathbb{SLP}_\oplus\ P_1$ | $\mathbb{SLP}_\oplus\ P_2$ |
|---|---|---|
| $v_1 \leftarrow a \oplus b;$ | $v_1 \leftarrow a \oplus b;$ | $v_1 \leftarrow a \oplus b;$ |
| $v_2 \leftarrow a \oplus b \oplus c;$ | $v_2 \leftarrow v_1 \oplus c;$ | $v_2 \leftarrow v_1 \oplus c;$ |
| $v_3 \leftarrow a \oplus b \oplus c \oplus d;$ | $v_3 \leftarrow v_2 \oplus d;$ | $v_3 \leftarrow v_2 \oplus d;$ |
| $v_4 \leftarrow b \oplus c \oplus d;$ | $v_4 \leftarrow b \oplus c \oplus d;$ | $v_4 \leftarrow v_3 \oplus a;$ |
| $\mathsf{ret}(v_1, v_2, v_3, v_4)$ | $\mathsf{ret}(v_1, v_2, v_3, v_4)$ | $\mathsf{ret}(v_1, v_2, v_3, v_4)$ |

where $[\![P_0]\!] = [\![P_1]\!] = [\![P_2]\!]$, $\#_\oplus P_0 = 8$, $\#_\oplus P_1 = 5$, and $\#_\oplus P_2 = 4$. We notice that, in $P_2$, the $\oplus$-cancellativity is effectively used to compute $v_4$. We can verify that there is no $Q$ with $\#_\oplus Q < 4$ and $[\![P_0]\!] = [\![Q]\!]$ enumerating $Q \in \mathbb{SLP}_\oplus$, Moreover, there is no $Q$ with $\#_\oplus Q < 5$ and $[\![P_0]\!] = [\![Q]\!]$ unless using the $\oplus$-cancellativity.

This examples emphasizes the $\oplus$-cancellativity is essential to shorten $\mathbb{SLP}_\oplus$. We can also say that $P_2$ is 2x faster than $P_0$.

## 4.3 Compressing SLP by RePair

Instead of searching the shortest SLPs by tackling the intractable optimization problem, we employ the grammar compression algorithm RePair from grammar compression theory as a heuristic.

In the original paper of RePair [67], Larsson and Moffat applied a procedure called *pairing* recursively to compress data (RePair stands for ***re**cursive **pair**ing*). For an SLP $P$ and a pair $(x, y)$ of terms (constants and variables), we replace all the occurrences of the pair in $P$ introducing a fresh variable. Hereafter, we call this step Pair$(x, y)$. Let us apply Pair$(a, b)$ to the previous $\mathbb{SLP}_\oplus\ P_0$.

$$\begin{array}{l} v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow a \oplus b \oplus c; \\ v_3 \leftarrow a \oplus b \oplus c \oplus d; \\ v_4 \leftarrow b \oplus c \oplus d; \\ \mathsf{ret}(v_1, v_2, v_3, v_4) \end{array} \overset{\textsc{Pair}(a,b)}{\Longmapsto} \begin{array}{l} t_1 \leftarrow a \oplus b; \\ \hline v_2 \leftarrow t_1 \oplus c; \\ v_3 \leftarrow t_1 \oplus c \oplus d; \\ v_4 \leftarrow b \oplus c \oplus d; \\ \mathsf{ret}(t_1, v_2, v_3, v_4) \end{array}$$

It replaces all $a \oplus b$ with the new variable $t_1$ and reduces XORs.

To distinguish variables introduced by Pair and the others, we use horizontal lines as above. Variables introduced by Pair, $t_1, t_2, \ldots$, are called *temporals* and the others *originals*; e.g., $t_1$ is temporal and $v_2, v_3, v_4$ are original.

To define our version of RePair, we need a total order $\prec$ on terms and extend it to the lexicographic ordering $\sqsubset$ on pairs. In this paper, as an example, we use the total order defined as follows: we order all constants in the alphabetical order and order temporal variables using their generation order: $t_1 \prec t_2 \prec \ldots \prec t_m$ where $t_i$ is generated before $t_{i+1}$ by Pair. Furthermore, we require $t \prec c$ for a temporal variable $t$ and a constant $c$.

Now, we define RePair using Pair as its subroutine.

**RePair**
***loop***
**(1)** If there is no original variable, we terminate.
**(2)** Otherwise, we choose a pair of terms that most frequently appears in the definitions of original variables (i.e., below the horizontal line). We then apply Pair with the pair. If there are multiple candidates, we select the smallest one for $\sqsubset$.

For example, let us apply RePair to the above $P_0$ omitting ret:

$$\begin{array}{l} v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow a \oplus b \oplus c; \\ v_3 \leftarrow a \oplus b \oplus c \oplus d; \\ v_4 \leftarrow b \oplus c \oplus d; \end{array} \overset{(a,b)}{\Longmapsto} \begin{array}{l} t_1 \leftarrow a \oplus b; \\ \hline v_2 \leftarrow t_1 \oplus c; \\ v_3 \leftarrow t_1 \oplus c \oplus d; \\ v_4 \leftarrow b \oplus c \oplus d; \end{array} \overset{(t_1,c)}{\Longmapsto} \begin{array}{l} t_1 \leftarrow a \oplus b; \\ t_2 \leftarrow t_1 \oplus c; \\ \hline v_3 \leftarrow t_2 \oplus d; \\ v_4 \leftarrow b \oplus c \oplus d; \end{array}$$

$$\overset{(t_2,d)}{\Longmapsto} \begin{array}{l} t_1 \leftarrow a \oplus b; \\ t_2 \leftarrow t_1 \oplus c; \\ t_3 \leftarrow t_2 \oplus d; \\ \hline v_4 \leftarrow b \oplus c \oplus d; \end{array} \overset{(b,c)}{\Longmapsto} \begin{array}{l} t_1 \leftarrow a \oplus b; \\ t_2 \leftarrow t_1 \oplus c; \\ t_3 \leftarrow t_2 \oplus d; \\ \hline t_4 \leftarrow b \oplus c; \\ v_2 \leftarrow t_4 \oplus d; \end{array} \overset{(t_4,d)}{\Longmapsto} \begin{array}{l} t_1 \leftarrow a \oplus b; \\ t_2 \leftarrow t_1 \oplus c; \\ t_3 \leftarrow t_2 \oplus d; \\ t_4 \leftarrow b \oplus c; \\ \hline t_5 \leftarrow t_4 \oplus d; \end{array}$$

At the first step, the pairs $(a, b)$ and $(b, c)$ appear three times, and we choose $(a, b)$ because $(a, b) \sqsubset (b, c)$. The rest of the parts are processed in the same way. We note that RePair reduces eight XORs to five, and the obtained SLP equals the previous $P_1$.

## 4.4 New Heuristic: XorRePair

We extend RePair by accommodating the XOR-cancellativity, which is not considered at all in RePair.

First, we introduce an auxiliary procedure, Rebuild$(v)$, which rewrites the definition of a given original variable $v$ using the values of temporal variables. We also use the auxiliary notation $(\!|\, w \,|\!)$ to denote the value of a variable $w$.

**Rebuild**($v$: original variable)
**Initialize**: Let $rem := (\!|\, v \,|\!)$ and $\mathcal{S} := \emptyset$. $rem$ denotes a set of constants to be eliminated by XORing existing temporal variables.
***loop***
**(1)** If we cannot shorten $rem$ (i.e., there is no temporal variable $t$ such that $|rem \oplus (\!|\, t \,|\!)| < |rem|$), we return $rem \cup \mathcal{S}$ as the new definition of $v$.
**(2)** Otherwise, we choose a temporal variable $t$ that minimizes $|rem \oplus (\!|\, t \,|\!)|$ and update $rem := rem \oplus (\!|\, t \,|\!)$ and $\mathcal{S} := \mathcal{S} \cup \{t\}$. If there are multiple candidates $t$, we choose the smallest one for $\prec$.

For example, applying Rebuild$(v_4)$ to the following left SLP, we obtain a new equivalent definition $v_4 \leftarrow a \oplus t_3$:

$$\begin{array}{l} t_1 \leftarrow a \oplus b; \\ t_2 \leftarrow t_1 \oplus c; \\ t_3 \leftarrow t_2 \oplus d; \\ \hline v_4 \leftarrow b \oplus c \oplus d; \end{array} \quad \Downarrow \quad \{a, t_3\}$$

(1) Set $rem = \{b, c, d\} = (\!|\, v_4 \,|\!)$.
(2) Choose $t_3$ because
$t_1$: $|rem \oplus (\!|\, t_1 \,|\!)| = |\{a, c, d\}| = 3$;
$t_2$: $|rem \oplus (\!|\, t_2 \,|\!)| = |\{a, d\}| = 2$;
$t_3$: $|rem \oplus (\!|\, t_3 \,|\!)| = |\{a\}| = 1$;
(3) Set $rem = \{a\}$ and $\mathcal{S} = \{t_3\}$.
(4) Return $\{a\} \cup \{t_3\}$ because
$|rem \oplus (\!|\, t_i \,|\!)| \geq |\{a\}| \ \ \forall i \in \{1, 2, 3\}$.

Augmenting RePair with Rebuilt, we obtain XorRePair:

**XorRePair = RePair + Rebuild**

***loop***

> **(1) and (2)** are the same as RePair.
> **(3)** For each original variable $v$, if Rebuild($v$) is strictly smaller than the current definition of $v$, we update $v$.

Let us apply XorRePair to the example $P_0$:

$$\cdots \Longmapsto \begin{array}{l} t_1 \leftarrow a \oplus b; \\ t_2 \leftarrow t_1 \oplus c; \\ t_3 \leftarrow t_2 \oplus d; \\ \hline v_4 \leftarrow b \oplus c \oplus d; \end{array} \overset{\text{Rebuild}}{\Longmapsto} \begin{array}{l} t_1 \leftarrow a \oplus b; \\ t_2 \leftarrow t_1 \oplus c; \\ t_3 \leftarrow t_2 \oplus d; \\ \hline v_4 \leftarrow a \oplus t_3 \end{array} \overset{(a,t_3)}{\Longmapsto} \begin{array}{l} t_1 \leftarrow a \oplus b; \\ t_2 \leftarrow t_1 \oplus c; \\ t_3 \leftarrow t_2 \oplus d; \\ \hline t_4 \leftarrow a \oplus t_3; \end{array}$$

First, we reach the above left form. Next, we update $v_4$ as $v_4 \leftarrow a \oplus t_3$ since $\text{Rebuilt}(v_4) = \{a, t_3\}$ shortens the definition of $v_4$. Finally, $\text{Pair}(a, t_3)$ generates the shortest $\mathbb{SLP}_\oplus$ with 4 XORs as we have seen in §4.2. Clearly, XorRePair runs in polynomial time.

*Related approaches.* We note that pairing is one of *factoring* in the context of common subexpression elimination (CSE) of compiler construction [19]. Combining algebraic properties for simplifying expressions with CSE, like XorRePair, has been naturally considered in compiler construction [2, 75]; however, primal methods to choose terms to be factored are elaborated. We adopt RePair to simply implement our compressor. The effectiveness of RePair is already known in grammar compression [27] and in the context of compressing matrices over $\mathbb{F}_2$ for cryptography [60].

# 5 REDUCING MEMORY ACCESS

We reduce memory accesses of SLPs by employing *deforestation*, an optimization method of functional program [32, 41, 101].

Deforestation has a rich and deep theory, and optimizing general (functional) programs via deforestation requires carefully analyzing programs, and transformation for optimization becomes a complex procedure [22, 32, 41, 96, 100, 101]. On the other hand, thanks to the simplicity of SLPs, we can transform and optimize SLPs easily. To see the core idea of deforestation, let us consider the following SLP and the corresponding program:

$$\begin{array}{l} v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow v_1 \oplus c; \\ v_3 \leftarrow v_2 \oplus d; \\ \text{ret}(v_3) \end{array} \Leftrightarrow$$

```
program(a, b, c, d) {
  var out = ((a xor b) xor c) xor d;
  return(out);
}
```

As the reader might notice, `program` makes two intermediate byte arrays, which correspond to $v_1$ and $v_2$. Since these intermediate arrays are immediately released, we would like to eliminate them. To this end, we rewrite `program` by replacing `((a xor b) xor c) xor d` with the following fused one:

```
Xor4(a, b, c, d) {
  var out = Array::new(a.len());
  for i in 0..out.len():
    out[i] = ((a[i]^b[i])^c[i])^d[i]; // ^ = byte XOR
  return(out);
}
```

$\texttt{Xor}_4$ does not only generate intermediate arrays, but also reduces memory accesses. For $N$-bytes arrays $A$ and $B$, $A$ `xor` $B$ invokes $3N$ memory accesses (loading $A$ and $B$, and writing the XORed result); thus, `program` invokes $9N$ memory accesses for $N$-bytes arrays. On the other hand, $\texttt{Xor}_4$ invokes $5N$ memory accesses.

Now, we augment SLPs with variadic XOR operators to formalize fused XORs, like $\texttt{Xor}_4$. We note that no SLP faithfully reflects $\texttt{Xor}_4$ since the formalization of SLPs only admits binary operators.

## 5.1 MultiSLP and Memory Accessing Problem

We extend $\mathbb{SLP}_\oplus$ to $\mathbb{SLP}_{\vec{\oplus}}$ by accommodating variadic XORs $\bigoplus(\vec{t})$. $\mathbb{SLP}_{\vec{\oplus}}$ can represent the above $\texttt{Xor}_4$ as follows:

$$1)\ v \leftarrow \bigoplus(a, b, c, d); \qquad 2)\ \text{ret}(v);$$

We also impose on $\mathbb{SLP}_{\vec{\oplus}}$ that there is no nested XORs; indeed, we can remove them as $v \leftarrow (x_1 \oplus x_2) \oplus x_3 \Longmapsto v \leftarrow \bigoplus(x_1, x_2, x_3)$.

To formalize our memory access optimization problem, we define the number of memory accesses $\#_M(P)$ for $P \in \mathbb{SLP}_{\vec{\oplus}}$ as follows:

$$\#_M(P) = \sum \{n + 1 : v \leftarrow \bigoplus(t_1, t_2, \ldots, t_n) \in P\}.$$

**The minimum memory access problem**

For an $P \in \mathbb{SLP}_\oplus$, we find $Q \in \mathbb{SLP}_{\vec{\oplus}}$ that satisfies $[\![P]\!] = [\![Q]\!]$ and minimizes $\#_M(Q)$.

Theorem 1. *The minimum memory access problem cannot be solved in polynomial time unless* ***P=NP****.*

This problem is also intractable as the same as the shortest $\mathbb{SLP}_\oplus$. We prove the intractability in the long version of this paper [98] by reducing the Vertex Cover Problem (VCP) to the above one. The VCP was used by Boyar et al. [18] to prove the intractability of the shortest $\mathbb{SLP}_\oplus$ problem; however, we cannot apply their construction to our problem. This is because their key construction—normalization, presented below, from SLPs to binary SLPs where each instruction forms $v \leftarrow x \oplus y$— does not work well for $\mathbb{SLP}_{\vec{\oplus}}$ and $\#_M$:

$$v \leftarrow a \oplus b \oplus c; \ \Longmapsto \ \begin{array}{l} v' \leftarrow a \oplus b; \\ v \leftarrow v' \oplus c; \end{array}$$

Although the normalization keeps $\#_\oplus$, it increases $\#_M$ ($4 \to 6$). This normalization brought a significantly useful syntactic property on $\mathbb{SLP}_\oplus$ in [18]. Since we cannot count on such the property, we give a more detailed and elaborated construction in the long version.

## 5.2 XOR Fusion

We propose a heuristic, XOR fusion, which reduces memory accesses of a given $\mathbb{SLP}_\oplus$ by transforming it to $\mathbb{SLP}_{\vec{\oplus}}$.

**XOR fusion**

Repeatedly find and unfold a variable $v$ that is used *just once* in the program and does not appear in ret:

$$\begin{array}{l} v \leftarrow \bigoplus(t_1, t_2, \ldots, t_n); \\ v' \leftarrow \bigoplus(\ldots, v, \ldots); \end{array} \ \Longmapsto \ v' \leftarrow \bigoplus(\ldots, t_1, t_2, \ldots, t_n, \ldots);$$

The following is an example of the XOR fusion:

$$\begin{array}{l} v_1 \leftarrow a \oplus b; \\ v_2 \leftarrow v_1 \oplus c; \\ v_3 \leftarrow v_2 \oplus d; \end{array} \overset{v_1}{\Longmapsto} \begin{array}{l} v_2 \leftarrow \bigoplus(a, b, c); \\ v_3 \leftarrow v_2 \oplus d; \end{array} \overset{v_2}{\Longmapsto} v_3 \leftarrow \bigoplus(a, b, c, d);$$

The fusion reduces memory accesses, and the following holds.

Theorem 2. *Let $P$ be an $\mathbb{SLP}_{\vec{\oplus}}$, and $Q$ be an $\mathbb{SLP}_{\vec{\oplus}}$ obtained by applying the XOR fusion to $P$. Then, $\#_M(Q) < \#_M(P)$ holds.*

*Why do not unfold variables used more than once?* Let us consider the following three SLPs where $A$ is a source SLP, $B$ is obtained one by compressing $A$, and $C$ is obtained by fusing $A$.

$$
\begin{array}{lll}
A: & v_2 \leftarrow a \oplus b \oplus c \oplus d \oplus e \oplus f; & \text{compress} \searrow \\
 & v_3 \leftarrow a \oplus b \oplus c \oplus d \oplus e \oplus g; & \\
 & \downarrow \text{fuse} & B: \ v_1 \leftarrow \bigoplus(a,b,c,d,e); \\
 & & \quad\ v_2 \leftarrow v_1 \oplus f; \\
C: & v_2 \leftarrow \bigoplus(a,b,c,d,e,f); & \quad\ v_3 \leftarrow v_1 \oplus g; \\
 & v_3 \leftarrow \bigoplus(a,b,c,d,e,g); & \swarrow \text{disallow to fuse } v_1
\end{array}
$$

where $\#_M(A) = 30$ since one XOR issues three memory accesses, $\#_M(B) = 12$, and $\#_M(C) = 14$. Therefore, Theorem 2 does not hold. In other words, the fusion without the restriction uncompresses a given SLP too much.

On the other hand, this situation suggests that uncompressed but fused SLP may run quickly in the real situation. Since compressing introduces extra variables as above, it may bring terrible effects on cache. In the next section §6, we will consider cache optimization. Furthermore, we will compare the coding throughputs of directly fused SLPs and fully optimized (compressed, fused, and cache optimized) ones in §7.

# 6 REDUCING CACHE MISSES

We proposed the XOR fusion to reduce memory accesses in the previous section. We now go one step further and reduce cache misses. To this end, we first review a classical cache optimization technique, blocking, and then formalize our cache optimization problem on the basis of the (red-blue) pebble game [47, 91]. We see that our optimization problem cannot be solved in polynomial-time (unless **P** = **NP**) and provide polynomial-time heuristics.

## 6.1 Blocking Technique for Cache Reusing

Since the size of a CPU cache is small with compared to that of main memory, we can only put a few arrays if a given data to be encoded is large. For example, on a CPU with the typical L1 cache size 32KB, if a user encodes 1MB data on $\mathbf{RS}(10,4)$, the cache can only hold two arrays at once since the input data is divided into $8 \cdot 10$ arrays of $\frac{1\text{MB}}{80}$ ~12KB. Obviously, putting entire arrays to cache degrades the cache performance.

To hold many arrays in cache at once, we use the established cache optimization technique *blocking*, which splits large arrays into small blocks introducing a loop. Let us perform blocking for the following example, where we split each array into arrays of $\mathcal{B}$ bytes.

Original Program

```
main(...) {

  v1 = xor(A, B);
  v2 = xor(C, D);
  v3 = xor(v1, E, F);
  v4 = xor(v3, G, A);
  v5 = xor(v1, v3, v4);

  return (v2, v4, v5);
}
```

Blocked One (Blocksize is $\mathcal{B}$)

```
main(...) {
  for i in 0..(A.len() / B) {
    v1[i] = xor(A[i], B[i]);
    v2[i] = xor(C[i], D[i]);
    v3[i] = xor(v1[i], E[i], F[i]);
    v4[i] = xor(v3[i], G[i], A[i]);
    v5[i] = xor(v1[i], v3[i], v4[i]);
  }
  return (v2, v4, v5);
}
```

where $X^{[i]}$ is the $i$-th $\mathcal{B}$-bytes block of an array $X$; i.e., $X^{[0]} = X[0..\mathcal{B}]$, $X^{[1]} = X[\mathcal{B}..2\mathcal{B}]$, and so on.

***Measures of Cache Efficiency.*** We consider two measures of cache efficiency. (1) We consider the minimum cache capacity CCap to avoid *cache reloading* while computing a given program. It is called cache reloading to load a certain block that is once spilled from cache from memory to cache again. If we can transform a given program $P$ to an equivalent one $Q$ with $\mathrm{CCap}(Q) < \mathrm{CCap}(P)$, then we can say $Q$ is more cache efficient. (2) We consider IOcost the total number of I/O transfers between cache and memory. To formalize these measures, we augment SLPs with abstract cache memory.

## 6.2 SLP Augmented with Abstract LRU Cache

To formalize our measures, hereafter we use SLPs to represent the inside of the loop introduced by the blocking technique by forgetting indices. For example, the above program is rewritten as the following SLP $P_{eg}$:

$$
\begin{array}{lll}
P_{eg}: & 1)\ v_1 \leftarrow A \oplus B; & 2)\ v_2 \leftarrow C \oplus D; \\
 & 3)\ v_3 \leftarrow \bigoplus(v_1, E, F); & 4)\ v_4 \leftarrow \bigoplus(v_3, G, A); \\
 & 5)\ v_5 \leftarrow \bigoplus(v_1, v_3, v_4); & 6)\ \mathsf{ret}(v_2, v_4, v_5);
\end{array}
$$

We introduce notations to operate cache through SLPs.

***Computation with Cache.*** We simply consider a cache $C$ as an ordered sequence of blocks, $C = \beta_1\beta_2\ldots\beta_n$, where each block $\beta_i$ is just a variable or constant. The rightmost (resp. leftmost) block represents the most (resp. least) recently used element.

Let us consider to execute an XOR $v \leftarrow \bigoplus(t_1, t_2, \ldots, t_k)$. We access the arguments and then $v$; concretely,

(1) For the arguments, in the order $i = 1, 2, \ldots, k$, we load $t_i$ to $C$ if $t_i \notin C$ or update the position of $t_i$ if $t_i \in C$;
(2) Then, we allocate $v$ in $C$ if $v \notin C$ or update the position of $v$ if $v \in C$.

If the cache is full and there is no room for loading or allocating, we evict the LRU (least recently used) element in the cache. Each eviction corresponds to spilling or writing back a cached block to memory, and this eviction way corresponds to the typical cache replacement policy *LRU replacement policy* [46].

*Example.* Let us run our example SLP $P_{eg}$ with a 10-capacity cache. For the first XOR, we load $A$ and then $B$ and finally allocate $v_1$. These operations change $C$ as follows:

$$\mathit{empty} \overset{A}{\Rightarrow} A \overset{B}{\Rightarrow} A\ B \overset{v_1}{\multimap} A\ B\ v_1$$

where we use $\overset{\bullet}{\Rightarrow}$ to denote a load from memory and $\overset{\bullet}{\multimap}$ to an allocation in the cache.

For the second XOR, we load $C$ and $D$ and allocate $v_2$ as follows:

$$A\ B\ v_1 \overset{C}{\Rightarrow} \overset{D}{\Rightarrow} \overset{v_2}{\multimap} A\ B\ v_1\ C\ D\ v_2.$$

For the third XOR, we update the position of $v_1$ in the cache, load $E$ and $F$, and allocate $v_3$:

$$A\ B\ v_1\ C\ D\ v_2 \overset{v_1}{\longrightarrow} \overset{E}{\Rightarrow} \overset{F}{\Rightarrow} \overset{v_3}{\multimap} A\ B\ C\ D\ v_2\ v_1\ E\ F\ v_3$$

where we use $\overset{\bullet}{\longrightarrow}$ to denote a position update in the cache.

For the fourth XOR, fetching the arguments ($v_3$, $G$, and $A$) changes $C$ as follows:

$$A\ B\ C\ D\ v_2\ v_1\ E\ F\ v_3 \xrightarrow{v_3} \overset{G\ A}{\Rightarrow} \longrightarrow B\ C\ D\ v_2\ v_1\ E\ F\ v_3\ G\ A.$$

Since the fetch makes the cache $C$ full, we evict the LRU element $B$ and then allocate $v_4$ as follows:

$$B\ C\ D\ v_2\ v_1\ E\ F\ v_3\ G\ A \underset{B}{\overset{v_4}{\Rightarrow\!\!\circ}} C\ D\ v_2\ v_1\ E\ F\ v_3\ G\ A\ v_4$$

where we write $\underset{\bullet}{\Rightarrow}$ for eviction from the cache to memory.

Finally, we change $C$ as follows in the fifth XOR:

$$C\ D\ v_2\ v_1\ E\ F\ v_3\ G\ A\ v_4 \xrightarrow{v_1} \xrightarrow{v_3} \xrightarrow{v_4} \underset{C}{\overset{v_5}{\Rightarrow\!\!\circ}} D\ v_2\ E\ F\ G\ A\ v_1\ v_3\ v_4\ v_5.$$

Now, we introduce two notations for the cache efficiency of SLPs.

**CCap**($P$ : SLP)
CCap($P$) denotes the minimum cache capacity where we can run $P$ without cache reloading.

We here check $\mathsf{CCap}(P_{eg}) = 10$. If we use the cache of capacity 9, we need to replace $A$ with $G$ in the fourth XOR $v_4 \leftarrow \bigoplus(v_3, G, A)$, and this replacement leads to reloading $A$ as follows:

$$A\ B\ C\ D\ v_2\ v_1\ E\ F\ v_3 \underset{A}{\overset{G}{\Rightarrow}} \underset{B}{\overset{A}{\Rightarrow}} C\ D\ v_2\ v_1\ E\ F\ v_3\ G\ A$$

where we write $\underset{x}{\overset{y}{\Rightarrow}}$ for the replacement that evicts $x$ and loads $y$.

We also consider the number of I/O transfers required by SLPs.

**IOcost**($P$ : SLP, $c$ : cache capacity)
IOcost($P, c$) denotes the number of I/O transfers issued while running $P$ with a cache of $c$-capacity. There are two kinds of I/O transfers; transfers from cache to memory, $\underset{\bullet}{\Rightarrow}$, and transfers from memory to cache, $\underset{\bullet}{\Rightarrow}$.

It is clear that $\mathsf{IOcost}(P_{eg}, 10) = 7(\text{of } \underset{\bullet}{\Rightarrow}) + 2(\text{of } \underset{\bullet}{\Rightarrow}) = 9$. This measure is useful when cache capacity is determined by hardware. For example, when considering a CPU where the cache size is 32KB, and the cache block size is 64B (these are one of the standard parameters on recent CPUs), the cache can hold 512 blocks maximally at once; therefore, we should optimize $\mathsf{IOcost}(P, 512)$.

Hereafter, as an example, we consider a cache that can holds eight blocks maximally. We can easily check $\mathsf{IOcost}(P_{eg}, 8) = 13$.

## 6.3 Optimizing SLP via Register Allocation

To reduce CCap and IOcost, we try register allocation by identifying cache (resp. memory) of our setting as registers (resp. memory) of the usual register allocation setting. Register allocation basically consists from three phases [9, 20, 25, 26, 40]: (1) the register assignment phase where we rename variables of a given program so that it has smaller variables; (2) the register spilling phase where we insert instructions to move the contents of registers to/from memory if variables are many than actual registers; (3) the register coalescing phase where we merge variables that has the same meaning in the syntactic or semantic way.

Using the standard graph-coloring register assignment algorithm, we can obtain the following SLP from $P_{eg}$:

$$P_{\text{reg}}: \begin{array}{ll} 1)\ v_1 \leftarrow A \oplus B; & 2)\ v_2 \leftarrow C \oplus D; \\ 3)\ v_3 \leftarrow \bigoplus(v_1, E, F); & 4)\ v_4 \leftarrow \bigoplus(v_3, G, A); \\ 5')\ v_1 \leftarrow \bigoplus(v_1, v_3, v_4); & 6)\ \mathsf{ret}(v_2, v_4, v_1); \end{array}$$

Unlike 5) of $P_{eg}$, in 5′), we store the result $\bigoplus(v_1, v_3, v_4)$ to $v_1$ instead of $v_5$ of $P_{eg}$ since $v_1$ is no more needed after 5′).

Although register assignment reduces variables, $\mathsf{NVar}(P_{\text{reg}}) = 4$, and I/O transfers, $\mathsf{IOcost}(P_{\text{reg}}, 8) = 12$, it does not reduce CCap since $\mathsf{CCap}(P_{eg}) = \mathsf{CCap}(P_{\text{reg}}) = 10$. We note that register spilling is useless since the LRU replacement disallows to select cached elements to be evicted. Register coalescing also does not make any sense at least in the above example. These tell that the cache optimization for SLPs by register allocation is quite limited.

Below we employ another approach, where we rearrange statements and arguments in SLPs. It should be noted that program rearrangement or (re)scheduling is beyond register allocation.

Although register allocation on SLPs is not so powerful as above, it enjoys the following properties:

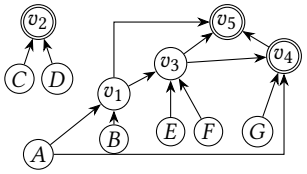
- The register assignment problem of SLPs can be solved in polynomial time. This comes from the relatively new result that the register assignment of programs in the SSA (static single assignment) form is tractable [14, 44, 81]. A program is in the SSA form if each variable is assigned exactly once [6, 33, 87]. Since there is no branching in SLPs, we can easily convert SLPs to SSA SLPs; thus, the register assignment problem for SLPs is tractable. Of course, the problem for general programs is intractable [26].
- The register coalescing problem for SSA SLPs is also tractable; indeed, the variable coalescing operation does not increase the required number of registers. The problem for general programs is intractable [15, 43].
- Register spilling is useful in the case where we can select elements to be evicted from the abstract cache. Then, on SSA SLPs, if each variable is used at most once, the minimum cost register spilling problem for SSA SLPs can be solved in polynomial time [16]. Without the constraint, the problem becomes intractable [37].

## 6.4 Optimizing SLP via Pebble Game

We employ the classical tool of program analysis *pebble game* to make a given SLP cache friendly. On this setting, we do not only rename variables as well as register assignment does but also reorder the entire program. We first introduce computation graphs, which are arenas of the pebble game, and then review the pebble game.

***Computation Graph.*** We use directed acyclic graphs (DAGs) to represent the value dependencies of SLPs:

$\mathcal{G}_{eg}$ :




This DAG corresponds to our example $P_{eg}$ in the following sense. Each leaf node (node with no children) represents the constant of the same name. Each inner node (node with children) represents the value obtained by XORing all children; thus, the inner node $v_1$ means $A \oplus B$, $v_3$ means $A \oplus B \oplus E \oplus F$, and so on. *Computation graphs (CGs)* are DAGs with double-circled nodes, *goal nodes*, which mean values returned by programs. It should be noted that CGs differ

from *interference graphs*, which are used in register allocation [9, 20, 25, 26, 40].

***Pebble Game.*** Let $\mathcal{G}$ be a CG. As the initialization step, for each leaf node $\ell$, we put the same name pebble on $\ell$. To represent this, we write $\mathcal{G}(\ell) = \ell$. We win a game if every goal node has a pebble. To achieve this, we pebble inner nodes using the following rules:

- At each turn, the player proposes an instruction of the form
$$n : p \leftarrow \bigoplus(p_1, p_2, \ldots, p_k)$$
where $n$ is a node of $\mathcal{G}$, $p$ and $p_i$ are pebbles, and the set $\{n_i : \mathcal{G}(n_i) = p_i\}$ equals to $n$'s children.
  - If $p$ is a new pebble, we put $p$ on $n$ so that $\mathcal{G}(n) = p$.
  - Otherwise, $p$ is in a node $m$, we *move* $p$ from $m$ to $n$.
- To avoid computing a single node multiple times, we are disallowed to put or move a pebble to a node once pebbled. This rule disallows to move pebbles from goal nodes.

Our pebble game on CGs is equivalent to the standard pebble game of Sethi [90, 91]. Especially, it is equivalent to the red-blue pebble game of Hong and Kung [47] that our game with the measure IOcost and the abstract cache $C$ where we can select elements to be evicted from $C$ instead of the LRU replacement rule.

*Example.* Let us consider the following winning strategy (with a return statement) of the above CG $\mathcal{G}_{eg}$:

$$Q: \begin{array}{ll} 1)\ v_1 : p_1 \leftarrow B \oplus A; & 2)\ v_3 : p_2 \leftarrow \bigoplus(E, F, p_1); \\ 3)\ v_4 : p_3 \leftarrow \bigoplus(A, G, p_2); & 4)\ v_5 : p_1 \leftarrow \bigoplus(p_1, p_2, p_3); \\ 5)\ v_2 : p_2 \leftarrow C \oplus D; & 6)\ \mathsf{ret}(p_2, p_3, p_1); \end{array}$$

This is better than $P_{\mathrm{reg}}$ at all the parameters since $\mathsf{NVar}(Q) = 3$, $\mathsf{CCap}(Q) = 5$, and $\mathsf{IOcost}(Q, 8) = 9$. For example, we can easily confirm $\mathsf{CCap}(Q) = 5$ as follows:

$$\mathit{empty} \overset{B}{\Rightarrow}\overset{A}{\Rightarrow}\overset{p_1}{\Rightarrow}\ B\ A\ p_1 \overset{E}{\Rightarrow}\overset{F}{\Rightarrow}\overset{p_1}{\longrightarrow}\overset{p_2}{\multimap_B}\ A\ E\ F\ p_1\ p_2 \overset{A}{\longrightarrow}\overset{G}{\Rightarrow}\overset{p_2}{\longrightarrow}\overset{p_3}{\multimap_E}$$
$$p_1\ A\ G\ p_2\ p_3 \overset{p_1}{\longrightarrow}\overset{p_2}{\longrightarrow}\overset{p_3}{\longrightarrow}\overset{p_1}{\longrightarrow}\ A\ G\ p_2\ p_3\ p_1 \overset{C}{\Rightarrow}\overset{D}{\Rightarrow}\overset{p_2}{\longrightarrow}\ p_3\ p_1\ C\ D\ p_2$$

where $\overset{y}{\multimap_x}$ means the replacement that evicts $x$ and allocates $y$.

The pebble game immediately implies the following property.

Proposition 1. *Let $\mathcal{W}$ be a winning strategy of the CG of an SLP $P$. Forgetting the node information from $\mathcal{W}$ and adding the adequate return statement, we can obtain an SLP $Q_{\mathcal{W}}$ such that $[\![P]\!] = [\![Q_{\mathcal{W}}]\!]$.*

**Notation**: Let $P$ and $Q$ be $\mathbb{SLP}^{\vec{}}_{\oplus}$. If $Q$ is obtained from a winning strategy of the CG of $P$ in the above manner, we write $P \vdash Q$.

## 6.5 Intractability of Optimization Problems

Employing the pebble game is not only valuable for concretizing our cache optimization problem but also valuable to correctly refer to established results of compiler construction and program analysis.

Theorem 3. *Let $P$ be an $\mathbb{SLP}^{\vec{}}_{\oplus}$. All the following optimization problems cannot be solved in polynomial-time unless **P=NP**:*

(1) *Finding $Q \in \mathbb{SLP}^{\vec{}}_{\oplus}$ that satisfies $P \vdash Q$ and minimizes $\mathsf{NVar}(Q)$.*
(2) *Finding $Q \in \mathbb{SLP}^{\vec{}}_{\oplus}$ that satisfies $P \vdash Q$ and minimizes $\mathsf{CCap}(Q)$.*
(3) *For a given cache capacity $c$, finding $Q \in \mathbb{SLP}^{\vec{}}_{\oplus}$ that satisfies $P \vdash Q$ and minimizes $\mathsf{IOcost}(Q, c)$.*

In order to show the intractability of Problems (1) and (2), we can use the NP-completeness of the standard pebble game shown by Sethi [90, 91]. Sethi reduced the classical NP-complete problem 3SAT [30] to the decision problem of the standard pebble game. Problem (1) is the optimizing version of the decision problem of the standard pebble game; hence, it is intractable. Although Problem (2) seems a problem involved in cache, it is a problem of the standard pebble game rather than the pebble game with cache. Indeed, if we could select pebbles to be evicted instead of the LRU rule, Problems (1) and (2) are essentially equivalent. Even if we follow the LRU rule, the construction given by Sethi in [91] also works well; therefore, Problem (2) is intractable.

On the other hand, Problem (3) should be analyzed using the pebble game augmented with cache, i.e., the red-blue pebble game of Hong and Kung [47]. We can choose nodes to be evicted from the cache on the ordinal formalization of the red-blue pebble game. The intractability of Problem (3) on the red-blue pebble game was already shown in [35, 80]. In [80], Papp and Wattenhofer used the NP-complete problem Hamiltonian path problem [39, 58] and succeeded in providing a simple NP-completeness proof. Fortunately, we can directly apply the construction of Papp and Wattenhofer to Problem (3) in our setting with the LRU eviction rule.

It is worth noting that Problem (1) and its variant can be efficiently solved when playing the pebble game on trees rather than DAGs [36, 69, 71, 76, 89, 92].

## 6.6 Two Scheduling Heuristics

We consider two simple heuristics for solving the pebble game since our interested problems are intractable. More technically, it is known that those problems are hard to approximate [34, 80].

*DFS-based algorithm.* Our first heuristic visits the nodes of a given CG in the postorder traversal.

Let us see how our heuristic works for our CG $\mathcal{G}_{eg}$. We need to decide which root node is visited first; here, we choose $v_2$ on the basis of the total ordering $\prec$ defined in §4.3 since $v_2 \prec v_5$. We then visit the children $C$ and $D$ in this order since $C \prec D$. Using $\prec$ as the tie-breaker, we make the following postorder traversing:

$$C \to D \to v_2 \to A \to B \to v_1 \to E \to F \to v_3 \to G \to v_4 \to v_5.$$

On the basis of this order, we generate a winning strategy as follows:

$$Q_{\mathrm{DFS}}: \begin{array}{ll} 1)\ v_2 : p_1 \leftarrow C \oplus D; & 2)\ v_1 : p_2 \leftarrow A \oplus B; \\ 3)\ v_3 : p_3 \leftarrow \bigoplus(p_2, E, F); & 4)\ v_4 : p_4 \leftarrow \bigoplus(p_3, A, G); \\ 5)\ v_5 : p_3 \leftarrow \bigoplus(p_2, p_3, p_4); & 6)\ \mathsf{ret}(p_1, p_4, p_3); \end{array}$$

Our pebble assigning policy is simple: If we have a pebble on $\mathcal{G}_{eg}$ that can move, we reuse it; otherwise, we put a fresh pebble. It can be verified that $\mathsf{NVar}(Q_{\mathrm{DFS}}) = 4$, $\mathsf{CCap}(Q_{\mathrm{DFS}}) = 7$, and $\mathsf{IOcost}(Q_{\mathrm{DFS}}, 8) = 10$.

*Bottom-up greedy algorithm.* Our next heuristic is a greedy one. Unlike the above DFS-based algorithm, this heuristic requires a parameter $c$ corresponding to cache capacity.

(i) Choose a computable node $n$, whose children have pebbles, that maximises the ratio $\frac{|H|}{|C|}$ where $C$ are the children of $n$ and $H \subseteq C$ are the children whose pebble in the cache.
(ii) Access $H$ and then access $C$.

(iii) If there is a movable cached pebble, we move it to $n$. Otherwise, we use a movable pebble or allocate a fresh pebble.

Here we again use $\prec$ as the tie-breaker.

We revisit the CG $\mathcal{G}_{eg}$ as follows. On the initial state, $v_1$ and $v_2$ are ready with $\frac{|\emptyset|}{|\{A,B\}|} = \frac{0}{2}$ for $v_1$ and $\frac{|\emptyset|}{|\{C,D\}|} = \frac{0}{3}$ for $v_2$. We choose $v_1$ since $v_1 \prec v_2$, and the generated statement changes $C$ as the following right:

$$v_1 : p_1 \leftarrow A \oplus B; \qquad empty \overset{A}{\Rightarrow}\overset{B}{\Rightarrow}\overset{p_1}{\multimap} A\ B\ p_1.$$

Next, we choose $v_3$ since $v_3 : \frac{|\{v_1\}|}{|\{v_1,E,F\}|} = \frac{1}{3}$ and $v_2 : \frac{|\emptyset|}{|\{C,D\}|} = \frac{0}{2}$, and compute $v_3$ with a fresh pebble $p_2$. Repeating this procedure, we obtain the following sequences and an SLP $Q_{\text{greedy}}$:

$$\begin{aligned}
&v_3 : p_2 \leftarrow \bigoplus(p_1, E, F); && \cdots \overset{p_1}{\rightarrow}\overset{E}{\Rightarrow}\overset{F}{\Rightarrow}\overset{p_2}{\multimap} ABp_1EFp_2,\\
&v_4 : p_3 \leftarrow \bigoplus(p_2, A, G); && \cdots \overset{p_2}{\rightarrow}\overset{A}{\rightarrow}\overset{G}{\Rightarrow}\overset{p_3}{\multimap} Bp_1EFp_2AGp_3,\\
&v_5 : p_1 \leftarrow \bigoplus(p_1, p_2, p_3); && \cdots \overset{p_1}{\rightarrow}\overset{p_2}{\rightarrow}\overset{p_3}{\rightarrow}\overset{p_1}{\rightarrow} BEFAGp_2p_3p_1,\\
&v_2 : p_2 \leftarrow C \oplus D; && \cdots \overset{C}{\underset{B}{\Rightarrow}}\overset{D}{\underset{E}{\Rightarrow}}\overset{p_2}{\rightarrow} EFAGp_3p_1CDp_2,\\
&\text{ret}(p_2, p_3, p_1);
\end{aligned}$$

It can be verified that $\mathsf{NVar}(Q_{\text{greedy}}) = 3$, $\mathsf{CCap}(Q_{\text{greedy}}) = 7$, and $\mathsf{IOcost}(Q_{\text{greedy}}, 8) = 9$. The scores of NVar and IOcost are optimal.

# 7 EVALUATION AND DISCUSSION

We evaluate our optimizing methods. In §7.1, we explain our dataset. In §7.2, we see throughputs of an unoptimized SLP on different block sizes. In §7.3, we show the average performance of (XOR)Repair of §4, the XOR fusion of §5.2, and scheduling heuristics of §6.6. In §7.4, we tell how the block size of the blocking technique affects coding performance. In §7.5, we show coding throughputs of programs fully optimized by our methods. In §7.6, we compare our throughputs with Intel's ISA-L [52] and the state-of-the-art study [104].

All experiments are conducted on the following environments:

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

The cache specification of these CPUs are the same; the L1 cache size is 32KB/core, the L1 cache associativity is 8, and the cache line size is 64 bytes. Our EC library and codes to reproduce the results in this section can be found in [99]. Our library is written by Rust and compiled by rustc-1.50.0.

***Important Remark.*** We select the above environments for the following reason. Since it has not been opened that the source codes implemented and used in the study of Zhou and Tian [104], we cannot directly compare our methods and theirs by running programs. Thus, we borrow values from [104] and compare them with our results measured on the above environments, which close to theirs Intel i7-4790(4.0 GHz, 4 cores, 32KB cache, 64-bytes cache block, 8-way cache assoc.) and AMD Ryzen 1700X(3.8 GHz, 8 cores, 32KB cache, 64-bytes cache block, 8-way cache assoc.).

## 7.1 Dataset

As an evaluation dataset, we use matrices of the codec **RS**$(10, 4)$, which is used in Hadoop HDFS [8] as stated in §1. We have 1002 coding matrices—one encoding matrix and $\binom{14}{4} = 1001$ decoding matrices obtained by removing 4 rows from the encoding matrix. We need the finite field $\mathbb{F}_{2^8}$ to make a Vandermonde matrix for coding, as we have seen in §1. We implemented it in our experimental library on the basis of the standard construction.

Technically speaking (to readers who are familiar with coding theory), we implement $\mathbb{F}_{2^8}$ using the primitive polynomial $x^8 + x^4 + x^3 + x^2 + 1$ used in ISA-L [52]. For **RS**$(10, 4)$, to use the same encoding matrix of ISA-L, we adopt the reduced form (aka, standard form [63, 70, 73]) $\mathcal{V}$ of a $(14 \times 10)$ Vandermonde matrix given by the standard construction as follows:

$$\begin{pmatrix} 1 & \alpha & \cdots & \alpha^9 \\ 1 & \alpha^2 & \cdots & (\alpha^2)^9 \\ \vdots & \vdots & & \vdots \\ 1 & \alpha^{14} & \cdots & (\alpha^{14})^9 \end{pmatrix} = \begin{pmatrix} V_{10\times10} \\ M_{4\times10} \end{pmatrix} \overset{\cdot V^{-1}_{10\times10}}{\mapsto} \mathcal{V} = \begin{pmatrix} \text{Ident}_{10\times10} \\ M_{4\times10}V^{-1}_{10\times10} \end{pmatrix}$$

where $\alpha$ is a primitive element of our $\mathbb{F}_{2^8}$ [84], and the reduced version $\mathcal{V}$ is the actual encoding matrix of ISA-L.

We write $P_{\text{enc}}$ for the SLP that corresponds to the bitmatrix form $\widetilde{\mathcal{V}}$ of our encoding matrix $\mathcal{V}$, as seen in §1. We write $\mathcal{P}_{\text{RS}}$ for the sets of all the SLPs corresponding to the coding matrices.

## 7.2 Performance of Unoptimized $P_{\text{enc}}$ on Various Block Sizes

As we have seen in §4.1, the execution of SLP is executing array XORs. Since we apply the blocking technique of §6.1 to exploit cache, we prepare two procedures for XORing blocked arrays of size $\mathcal{B}$. The first one `xor1` performs the byte XORing element-wise. The second one `xor32` performs the 32byte XORing—`mm256_xor`, SIMD AVX2 instruction—element-wise. Such SIMD instructions are used in ISA-L and the previous study [104]. Furthermore, AVX2 is the standard instruction set for recent CPUs.

```
xor1(x, y) {
 var out = Array::new(ℬ);
 for(i=0; i<ℬ; i+=1):
   out[i] = x[i] xor y[i];
 return out;
}
```

```
xor32(x, y) {
 var out = Array::new(ℬ);
 for(i=0; i<ℬ; i+=32):
   (out[i] as m256)
      = mm256_xor(&x[i], &y[i]);
 return out;
}
```

where `m256` is the type of 32 bytes in AVX2. In the same way, we implement $n(> 2)$ arity versions for running fused SLPs.

To measure the performance of our unoptimized SLP $P_{\text{enc}}$, we execute it for randomly generated arrays of 10MB. The following table is the average throughput (GB/sec) of 1000-times executions.

| Throughput (GB/sec) | xor1 | xor32 | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Blocksize | | 64 | 128 | 256 | 512 | 1K | 2K | 4K |
| **intel** | 0.16 | 0.62 | 1.12 | 2.05 | 3.02 | 4.03 | 4.78 | 4.72 |
| **amd** | 0.17 | 0.67 | 1.17 | 1.72 | 2.16 | 2.78 | 3.17 | 3.29 |

The power of SIMD is remarkable, as already reported in [83, 104]. This result also suggests that the bottleneck is shifted from CPUs to memory I/O by the SIMD instruction.

Despite our argument about cache efficiency in §6, the performances of small blocks—64, 128, 256, and 512—are worse than those of large blocks, 1K, 2K, and 4K. It is well-known in the context of cache optimization that a too-small block is not good in real computing [28, 64, 86, 103]. We will evaluate and discuss how the change of block sizes affects coding performance below in §7.4.

## 7.3 Average Reduction Ratios of Our Methods

*Reducing Operators.* We evaluate our SLP compression heuristics, RePair and XorRePair. The following table displays the average performance of (Xor)RePair for the 1002 SLPs of **RS**(10, 4):

| Avg% | $\frac{\text{Repair}(P)}{P}$ | $\frac{\text{XorRepair}(P)}{P}$ | Corresp. Value from [104] |
|---|---|---|---|
| XOR Num. $\#_{\oplus}(\cdot)$ | 42.1% | 40.8% | ∼65.0% |

where the first and second ratios are the average ratios of reducing XORs by our heuristics defined as follows;

$$Avg\left(\left\{\frac{\#_{\oplus}C(P)}{\#_{\oplus}P} : P \in \mathcal{P}_{\text{RS}}\right\}\right) = \begin{cases} 42.1\% & \text{if } C = \text{Repair}, \\ 40.8\% & \text{if } C = \text{XorRepair}. \end{cases}$$

We note that the smaller the ratio, the better the compressing performance. The value 65.0% is the best ratio among the XOR reduction heuristics for bitmatrices evaluated in [104]. Although RePair is simple and developed initially in grammar compression, we can see it works very well.

This table also says XorRePair exploits the cancellative property of XOR; but, the difference is minor. It is not surprising; indeed, exponential-time compression heuristics and an algorithm, which corresponds to RePair, were already compared in [60] for the application to cryptography, and there was also a slight difference. These results mean that RePair efficiently compresses programs, even though it does not use the cancellativity of XOR. We consider this comes from the robustness of RePair, which also appears in grammar compression when comparing it with other compression algorithms, such as LZ77 and LZ78 [27].

*Reducing Memory Access.* We see how XorRepair and the XOR fusion of §5.1 reduce memory access $\#_M(\cdot)$:

| Avg% | $\frac{\text{Co}(P)}{P}$ | $\frac{\text{Fu}(P)}{P}$ | $\frac{\text{Fu}(\text{Co}(P))}{\text{Co}(P)}$ | $\frac{\text{Fu}(\text{Co}(P))}{P}$ |
|---|---|---|---|---|
| $\#_M(\cdot)$ | 40.8% | 35.1% | 59.2% | 24.1% |

where Co means XorRepair, and Fu means the XOR fusion.

The second ratio says that the XOR fusion averagely reduces ∼65% memory accesses for *uncompressed* SLPs. We can see the other columns in the same way. Therefore, we can tell that XorRepair and the XOR fusion work well independently; furthermore, combining them averagely reduces ∼76% memory accesses on average.

*Reducing Variables and Required Cache Size.* We consider how the XOR fusion and our DFS-scheduling heuristic averagely affect the two measures of the cache efficiency NVar and CCap.

| Avg% | $\frac{\text{Co}(P)}{P}$ | $\frac{\text{Fu}(P)}{P}$ | $\frac{\text{Fu}(\text{Co}(P))}{\text{Co}(P)}$ | $\frac{\text{Dfs}(\text{Fu}(\text{Co}(P)))}{\text{Co}(P)}$ |
|---|---|---|---|---|
| NVar | 1552% | 100% | 38.9% | 24.5% |
| CCap | 498% | 98.7% | 51.2% | 40.0% |

where Dfs means our DFS-based scheduling heuristic. We skip using our greedy-scheduling heuristic and the measure IOcost(_, _) since they depend on the cache sizes determined by our block size $64, 128, \ldots, 4K$, and the table including those values for of all the cache sizes becomes too large.

The first ratio clarifies that XorRepair significantly degrade cache efficiency. Comparing the third and fourth ratios, we can say the scheduling heuristic certainly improves cache efficiency. Multiplying the first and fourth ratios derives $\frac{\text{CCap}(\text{Dfs}(\text{Fu}(\text{Co}(P))))}{\text{CCap}(P)} \sim$ 199%; therefore, we can say that the scheduling heuristic can suppress the side effect of XorRepair to some extent.

We consider why XorRepair significantly degrades NVar and CCap. It results from the intrinsic behaviors of (Xor)RePair; namely, they add many temporal variables without considering cache and register efficiency. The same inefficiency problem was pointed in the early research of program optimization as the weak point of CSE [4]. Furthermore, comparing the second and third ratios, we can also tell that (Xor)RePair generate many variables used just once in programs, which can be deforested.

## 7.4 Selecting Adequate Blocksize

As we have seen in §6, the block size $\mathcal{B}$ of the blocking technique is an essential optimization parameter. Although small blocks are supposed to enable the cache to hold all blocks at once, the performance table in §7.2 defies our prediction. Here we see additional experiments and think about why the performance on small blocks is not good. To check whether or not the tendency about cache block sizes is peculiar in unoptimized programs, we first see the performance of the uncompressed but fused version $P_{\text{enc}}^{+F}$ of $P_{\text{enc}}$.

***Case1: Uncompressed but Fused SLP.*** The following table is the coding throughputs (GB/sec) of $P_{\text{enc}}^{+F}$:

| Block size (byte) | 64 | 128 | 256 | 512 | 1K | 2K | 4K |
|---|---|---|---|---|---|---|---|
| **intel** | 0.87 | 1.73 | 2.85 | 4.08 | 5.29 | 5.78 | 4.36 |
| **amd** | 1.32 | 2.18 | 3.15 | 3.54 | 3.97 | 4.16 | 3.82 |

where $\text{NVar}(P_{\text{enc}}^{+F}) = 32$ and $\text{CCap}(P_{\text{enc}}^{+F}) = 88$.

We see there are the same patterns at **intel** and **amd**; i.e., $2K > 1K > 4K > 512 > 256 > 128 > 64$. This result again defies our prediction since we need $\mathcal{B} \leq 256$ to avoid cache reloading.

***Possible reasons for poor performance of small blocks.*** The performance problem on small blocks may cause from two sources.

*Cache conflicts in cache sets.* The first source is cache conflicts in *cache sets*, and it prevents cache from holding $32K/\mathcal{B}$ blocks. Generally, the 32K bytes cache with 8 cache associativity has $\frac{32K}{8} = 4K$ cache sets where each cache set can hold 8 cache blocks. Accessing a cache block $b$ whose start address is $\mathcal{A}(b)$, a CPU with the cache tries to assign $b$ to the $(\mathcal{A}(b) \bmod 4K)$-th cache set. If the cache set is full (i.e., it already has 8 cache blocks), the CPU evicts the LRU cache block in the set to memory. Therefore, accessing two blocks $b_1$ and $b_2$ such that $\mathcal{A}(b_1) \equiv_{4K} \mathcal{A}(b_2)$ may cause an eviction in a cache set. For more detailed explanations about cache conflicts, the reader is referred to standard literature such as [46].

Because of cache conflicts, a cache holds at most 8 blocks regardless of the size $\mathcal{B}$ in the worst case. Therefore, in that case, we can use cache most effectively when $\mathcal{B} = 4K$, and least when $\mathcal{B} = 64$.

Several approaches have been proposed to avoid such the worst situation [64, 79]; however, it is a hard problem to align (the start addresses of) involved arrays optimally.

Namely, the smaller the block size, the more difficult using cache efficiently as expected.

*Latency Penalty.* The second source of the poor performance may be memory access latency. It is clear that, if $\mathcal{B}$ becomes smaller, then the number of required iteration becomes larger. Therefore, on small blocks, there are many unavoidable block loading caused by changing iterations.

We focus the memory access latency on modern CPUs. For example, we consider Intel's Haswell microarchitecture [45] released in 2013, and it and its successor are widely used today. Haswell needs about 150 CPU cycles as latency to communicate with memory [46, 55]. Even if the CPU pipeline maximally works, we need $150 + \frac{\mathcal{B}}{n \cdot 8}$ cycles to load or store a block on an $n$-channel memory. Thus, we need 158-cycles to load a 64-bytes block at once on a single channel memory. If we load a 64-bytes block in eight separate 8-bytes loads, then we need $(151 \times 8)$-cycles; thus, we should load a block from the memory as possible as large. Haswell can load two 32-bytes data from the cache, XOR the two data using AVX2 or AVX512, and store the result 32-bytes to the L1 cache in *a single cycle* [42]. Thus, we can perform XORing for two cached blocks of $\mathcal{B}$ bytes in $\frac{\mathcal{B}}{32}$ cycles. In a single channel memory, for $\mathcal{B} = 64$ (resp. $\mathcal{B} = 4K$), we can perform xor32 79-times (resp. ∼5 times) while loading one $\mathcal{B}$-bytes block from the memory. We notice the ratio $\frac{79}{5}$ ∼15.8 is smaller than $\frac{4K}{64} = 64$. Namely, the smaller the block size, then more block loads are required, and the total latency penalty of a small block is relatively larger than that of a large block.

***Case2: Full Optimization.*** As we have seen above, small blocks may degrade the performance of blocked programs. Here we see the coding performance of fully optimized—compressed, fused, and scheduled—version of $P_{\text{enc}}$, $P_{\text{enc}}^{\text{Full}}$, to check whether or not large blocks better for blocked programs than smaller ones.

| Block size | 64 | 128 | 256 | 512 | 1K | 2K | 4K |
|---|---|---|---|---|---|---|---|
| **intel** (greedy) | 2.29 | 4.00 | 6.02 | 7.61 | 8.68 | 8.37 | 7.24 |
| **intel** (dfs) | 2.32 | 3.97 | 6.09 | 7.37 | **8.92** | 8.55 | 7.64 |
| **amd** (greedy) | 1.91 | 3.30 | 4.36 | 5.07 | 6.08 | 7.32 | 7.15 |
| **amd** (dfs) | 1.84 | 3.25 | 4.60 | 5.04 | 6.36 | **7.58** | 7.31 |

where $\mathsf{NVar}(P_{\text{enc}}^{\text{Full}}) \sim 90$ and $\mathsf{CCap}(P_{\text{enc}}^{\text{Full}}) \sim 170$ for all the entries. We should note that our greedy scheduling generates different programs for each $\mathcal{B}$. However, for all $\mathcal{B}$, $\mathsf{NVar}(\cdot)$ is about 90, and $\mathsf{CCap}(\cdot)$ is about 170. The same is true for the DFS scheduling.

On the basis of the performance, hereafter we set $\mathcal{B} = 1K$ on **intel** and $\mathcal{B} = 2K$ on **amd** and use the DFS scheduling for comparison with ISA-L and the previous work.

We consider a reason why the scores of $1K$ and $2K$ are better than that of $4K$ in **intel**. Even if conflicts in cache sets happen, the cache with $1K$ and $2K$ blocks may hold more blocks than with $4K$; therefore, the CPU can efficiently use the cache in the case $1K$ and $2K$. On the other hand, in **amd**, the score of $1K$ is lower than $2K$ and $4K$. It possibly comes from a feature of the microarchitecture, *Zen+*, of **amd**'s CPU. Zen+, unlike **intel**'s CPU, performs each 256 bitwidth instruction of AVX2, splitting it into two 128 bitwidth instructions [38]. To put it simply, the performance for AVX2 of **amd** is half that of **intel**. Therefore, in the $1K$ case of **amd**, we think that the total latency penalty is more significant than the benefit of cache efficiency.

## 7.5 Throughput Analysis

Beyond the average analysis, we optimize the encoding SLP $P_{\text{enc}}$.

| | $P_{\text{enc}}$ | $\mathrm{Co}(P_{\text{enc}})$ | $\mathrm{Fu}(\mathrm{Co}(P_{\text{enc}}))$ | $\mathrm{Drs}(\mathrm{Fu}(\mathrm{Co}(P_{\text{enc}})))$ |
|---|---|---|---|---|
| $\#_{\oplus}(\cdot)$ | 755 | 385 | 146 | ← |
| $\#_{M}(\cdot)$ | 2265 | 1155 | 677 | ← |
| NVar | 32 | 385 | 146 | 88 |
| CCap | 92 | 447 | 224 | 167 |
| IOcost($1K$) | 1262 | 1465 | 1086 | 779 |
| **intel**($1K$) | 4.03 | 4.36 | 7.50 | **8.92** |
| IOcost($2K$) | 1598 | 1599 | 1144 | 845 |
| **amd**($2K$) | 3.17 | 4.46 | 6.62 | **7.58** |

where we note that our scheduling heuristics do not affect the number of XORs and memory accesses, and we represent it by ←.

Comparing the first and second columns, we see that $\#_M$ and IOcost are more dominant than CCap on the performance. On the other hand, comparing the third and fourth columns, we see that CCap and IOcost certainly reflect cache efficiency.

We also measure the performance of unoptimized and optimized versions of decoding SLPs. Here, we consider the decoding SLP $P_{\text{dec}}$ obtained by removing $\{2, 4, 5, 6\}$ rows from the encoding matrix because this SLP has the most XORs—1368 as we see in the following table—among decoding SLPs. The following table summarizes the related numbers and decoding performance of $P_{\text{dec}}$:

| | $P_{\text{dec}}$ | $\mathrm{Co}(P_{\text{dec}})$ | $\mathrm{Fu}(\mathrm{Co}(P_{\text{dec}}))$ | $\mathrm{Drs}(\mathrm{Fu}(\mathrm{Co}(P_{\text{dec}})))$ |
|---|---|---|---|---|
| $\#_{\oplus}$ | 1368 | 511 | 206 | ← |
| $\#_{M}$ | 4104 | 1533 | 923 | ← |
| NVar | 32 | 511 | 206 | 125 |
| CCap | 89 | 585 | 283 | 205 |
| **intel**($1K$) | 2.35 | 3.32 | 5.51 | 6.67 |
| IOcost($1K$) | 2824 | 1991 | 1530 | 1077 |
| **amd**($2K$) | 2.28 | 3.58 | 5.27 | 6.01 |
| IOcost($2K$) | 2824 | 2175 | 1590 | 1184 |

Since $P_{\text{dec}}$ has more instructions than $P_{\text{enc}}$, we can see that the throughputs of $P_{\text{dec}}$ is smaller than those of $P_{\text{enc}}$. On the other hand, the overall structure is similar to that of $P_{\text{enc}}$.

## 7.6 Throughput Comparison

We compare the performance of our fully optimized versions of $P_{\text{enc}}$ and $P_{\text{dec}}$ with ISA-L v2.30.0 [53] and values in [104]. As the same as [104], we consider three kinds of codec; 4-parities $\mathbf{RS}(d, 4)$, 3-parities $\mathbf{RS}(d, 3)$, and 2-parities $\mathbf{RS}(d, 2)$. We summarize main measures for each codec in Table 1. These values correspond to the rightmost value of the above tables in §7.5.

We compare the coding throughputs of $\mathbf{RS}(d, 4)$ on **intel** where we use $\mathcal{B} = 1K$ as our blocksize:

| **intel 1K** | Ours | | ISA-L v2.30 | | Values of [104] | |
|---|---|---|---|---|---|---|
| (GB/sec) | Enc | Dec | Enc | Dec | Enc | Dec |
| $\mathbf{RS}(8, 4)$ | 8.86 | 6.78 | 7.18 | 7.04 | 4.94 | 4.50 |
| $\mathbf{RS}(9, 4)$ | 8.83 | 6.71 | 6.91 | 6.58 | Not Available in [104] | |
| $\mathbf{RS}(10, 4)$ | 8.92 | 6.67 | 6.79 | 4.88 | 4.94 | 4.71 |

**Table 1: Values of $\#_{\oplus}$, $\#_M$, NVar$(\cdot)$, CCap$(\cdot)$, and IOcost$(_, 1K)$ of optimized coding SLPs for various codec.**

| | $\#_{\oplus}$ | | $\#_M$ | | NVar | | CCap | | IOcost$(1K)$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec | Enc | Dec |
| **RS**$(8,4)$ | 121 | 170 | 543 | 747 | 79 | 102 | 143 | 166 | 585 | 811 |
| **RS**$(9,4)$ | 132 | 182 | 611 | 829 | 83 | 117 | 155 | 189 | 671 | 968 |
| **RS**$(10,4)$ | 146 | 206 | 677 | 923 | 88 | 125 | 167 | 205 | 779 | 1077 |
| **RS**$(8,3)$ | 75 | 129 | 364 | 561 | 45 | 77 | 109 | 141 | 382 | 628 |
| **RS**$(9,3)$ | 87 | 144 | 417 | 641 | 58 | 91 | 128 | 163 | 468 | 724 |
| **RS**$(10,3)$ | 96 | 145 | 471 | 661 | 69 | 85 | 148 | 165 | 562 | 769 |
| **RS**$(8,2)$ | 26 | 65 | 180 | 286 | 17 | 38 | 80 | 102 | 191 | 313 |
| **RS**$(9,2)$ | 29 | 73 | 202 | 322 | 19 | 42 | 90 | 113 | 245 | 342 |
| **RS**$(10,2)$ | 30 | 77 | 222 | 352 | 19 | 50 | 98 | 130 | 285 | 418 |

The table claims that our EC library exceeds ISA-L in encoding and parallels in decoding.

Let us consider why there is no difference in the performance between **RS**(10, 4) and **RS**(8, 4) in our implementation, although **RS**(8, 4) is better than **RS**(10, 4) in terms of the measures in Table 1. To encode or decode a given data of $N$-bytes, we run SLPs for $8 \times 8$ input arrays of $\frac{N}{8\times 8}$-bytes on **RS**(8, 4). Similarly, we run SLPs for $8 \times 10$ input arrays of $\frac{N}{8\times 10}$-bytes on **RS**(10, 4). The difference in Table 1 is due to the fact that the number of input arrays of **RS**(10, 4) is larger than that of **RS**(8, 4). On the other hand, the total number of iterations $\frac{N}{8\times 8\times \mathcal{B}}$ on **RS**(8, 4) is larger than that $\frac{N}{8\times 10\times \mathcal{B}}$ on **RS**(10, 4). As a result, there is no difference in the performance between them.

The encoding and decoding performance of ISA-L are close; however, there is indeed a difference between them in our implementation. Our encoding and decoding matrices, which are sources of $P_{\mathrm{enc}}$ and $P_{\mathrm{dec}}$, equal those of ISA-L in the binary representation. Namely, we use the same matrix that ISA-L uses. We believe that this situation could be caused by the following fact. In ISA-L or EC libraries based on algorithms of MM over finite fields, for coding matrices $M_1$ and $M_2$ and a data matrix $D$, there is not much difference between the two computational costs of $M_1 \cdot D$ and $M_2 \cdot D$ because finite field multiplication is usually implemented using a multiplication table $T$; i.e., $a \cdot b$ is computed by accessing $T[a][b]$. On the other hand, in our libraries or EC libraries based on the method XOR-based EC, even if the sizes of two matrices $M_1$ and $M_2$ are equal, the number of required XORs could be very different. For example, for an element $e_1, e_2 \in \mathbb{F}_{2^8}$, the number of 1 in the bitmatrix $\widetilde{e_1}$ may be significantly larger than those of $\widetilde{e_2}$. Namely, we consider the performance gap between encoding and decoding in our implementation is intrinsic in XOR-based EC.

We also have similar structures in the throughputs table of **RS**$(d, 4)$ on **amd** where we use $\mathcal{B} = 2K$ as the blocksize:

| **amd 2K** | Ours | | ISA-L v2.30 | | Values of [104] | |
|---|---|---|---|---|---|---|
| (GB/sec) | Enc | Dec | Enc | Dec | Enc | Dec |
| **RS**$(8,4)$ | 7.09 | 5.53 | 4.60 | 4.61 | 4.69 | 4.06 |
| **RS**$(9,4)$ | 7.22 | 5.86 | 4.70 | 4.70 | Not Available in [104] | |
| **RS**$(10,4)$ | 7.58 | 6.01 | 4.76 | 4.75 | 4.67 | 3.91 |

*Comparison in Low Parities.* We compare **RS**$(d, 3)$ and **RS**$(d, 2)$:

| **intel 1K** | Ours | | ISA-L v 2.30 | | Values of [104] | |
|---|---|---|---|---|---|---|
| (GB/sec) | Enc | Dec | Enc | Dec | Enc | 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 | $8.40_E$ | $8.22_E$ |

| **amd 2K** | Ours | | ISA-L v 2.30 | | Values of [104] | |
|---|---|---|---|---|---|---|
| (GB/sec) | Enc | Dec | Enc | Dec | Enc | Dec |
| **RS**$(8,3)$ | 9.35 | 7.43 | 5.01 | 4.93 | $6.38_S$ | $5.18_Q$ |
| **RS**$(9,3)$ | 9.41 | 7.44 | 5.07 | 5.02 | $6.53_S$ | $6.53_S$ |
| **RS**$(10,3)$ | 9.51 | 7.46 | 5.04 | 5.02 | $6.49_S$ | $5.31_Q$ |
| **RS**$(8,2)$ | 13.60 | 12.07 | 7.11 | 7.09 | $8.96_R$ | $10.11_E$ |
| **RS**$(9,2)$ | 13.83 | 12.06 | 7.17 | 7.19 | $9.12_R$ | $9.31_R$ |
| **RS**$(10,2)$ | 14.13 | 12.19 | 7.24 | 7.15 | $9.31_R$ | $10.60_R$ |

The column "Values of [104]" consists of the best throughputs among results of the corresponding parameters in [104] where the authors compared their proposal method with some codecs specialized for low parities—STAR [51] and QFS [77] for three parities, and EvenOdd [12] and RDP [31] for two parities. Indeed, the values $\cdot_S$, $\cdot_Q$, $\cdot_E$, and $\cdot_R$ are scored by STAR, QFS, EvenOdd, and RDP, respectively (the other values are scored by their proposal approach). We can say our library works well without specializing for low parities.

## 8 CONCLUSION AND FUTURE WORK

We have proposed a streamlined approach to implement an efficient XOR-based EC library. We combined the four notions: straight-line programs (SLPs) from program optimization, the grammar compression algorithm RePair, the functional program optimization technique deforestation, and the pebble game of program analysis. We extended RePair to our XorRePair to accommodate the cancellative property of XOR. We used the pebble game to formalize our cache optimization problem on SLPs with the abstract LRU cache. Orthogonally composing these methods, we have implemented an experimental library that outperforms Intel's high-performance library, ISA-L [52].

Analyzing the result of experiments, we have noticed the importance of cache optimization. In this paper, we only tried to abstract the L1 cache but not the L2 and L3 caches. We are considering adopting the multilevel pebble game introduced by Savage in [88] to accommodate multilevel caches. We are also interested in automatically inserting software prefetches [68]. It may hide the cache transfer penalty from memory to cache if a CPU concentrates on performing array XORs against cached data.

## ACKNOWLEDGMENTS

We gratefully thank anonymous reviewers for their invaluable and thorough comments, which improved the presentation of this paper and also helped us improve the performance of our experimental library. Many thanks to our colleague Masahiro Fukasawa for fruitful discussions of cache optimization. Thanks also to Iori Yoneji for his full support in providing evaluation environments.

# REFERENCES


[1] A. V. Aho, S. C. Johnson, and J. D. Ullman. 1977. Code Generation for Expressions with Common Subexpressions. *J. ACM* 24, 1 (1977), 146–160.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

[3] A. V. Aho and J. D. Ullman. 1972. Optimization of Straight Line Programs. *SIAM J. Comput.* 1, 1 (1972), 1–19.

[4] F. E. Allen and J. Cocke. 1972. A Catalogue of Optimizing Transformations. In *Design and Optimization of Compilers*, R. Rustin (Ed.). Prentice-Hall, 1–30.

[5] J. Alman and V. V. Williams. 2021. A Refined Laser Method and Faster Matrix Multiplication. In *SODA '21*. 522–539.

[6] B. Alpern, M. N. Wegman, and F. K. Zadeck. 1988. Detecting Equality of Variables in Programs. In *POPL '88*. ACM, 1–11.

[7] AMD. 2020. *AMD64 Architecture Programmer's Manual.* https://www.amd.com/system/files/TechDocs/26568.pdf

[8] Apache Hadoop. 2020. *HDFS Erasure Coding.* https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html

[9] A. W. Appel and L. George. 2001. Optimal Spilling for CISC Machines with Few Registers. In *PLDI '01*. ACM, 243–253.

[10] ARM. 2020. *SIMD Neon.* https://developer.arm.com/documentation/ddi0487/latest

[11] Storage Networking Industry Association. 2009. *Common RAID Disk Data Format.* https://www.snia.org/tech_activities/standards/curr_standards/ddf

[12] M. Blaum, J. Brady, J. Bruck, and Jai Menon. 1995. EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Trans. Comput.* 44, 2 (1995), 192–202.

[13] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. 1995. An XOR-Based Erasure-Resilient Coding Scheme. *ICSI Technical Report No. TR-95-048* (1995).

[14] F. Bouchez, A. Darte, C. Guillon, and F. Rastello. 2007. Register Allocation: What Does the NP-Completeness Proof of Chaitin et al. Really Prove? Or Revisiting Register Allocation: Why and How. In *LCPC '07*. Springer, 283–298.

[15] F. Bouchez, A. Darte, and F. Rastello. 2007. On the Complexity of Register Coalescing. In *CGO '07*. 102–114.

[16] F. Bouchez, A. Darte, and F. Rastello. 2007. On the Complexity of Spill Everywhere under SSA Form. In *LCTES '07*. ACM, 103–112.

[17] J. Boyar, P. Matthews, and R. Peralta. 2008. On the Shortest Linear Straight-Line Program for Computing Linear Forms. In *MFCS '08*. Springer, 168–179.

[18] J. Boyar, P. Matthews, and R. Peralta. 2013. Logic Minimization Techniques with Applications to Cryptology. *Journal of Cryptology* 26 (2013), 280–312. Issue 2.

[19] M. A. Breuer. 1969. Generation of Optimal Code for Expressions via Factorization. *Commun. ACM* 12, 6 (June 1969), 333–340.

[20] P. Briggs, K. D. Cooper, and L. Torczon. 1994. Improvements to Graph Coloring Register Allocation. *ACM Trans. Program. Lang. Syst.* 16, 3 (May 1994), 428–455.

[21] J. Bruno and R. Sethi. 1976. Code Generation for a One-Register Machine. *J. ACM* 23, 3 (1976), 502–510.

[22] R. M. Burstall and J. Darlington. 1977. A Transformation System for Developing Recursive Programs. *J. ACM* 24, 1 (1977), 44–67.

[23] T. Carpenter, F. Rastello, P. Sadayappan, and A. Sidiropoulos. 2016. Brief Announcement: Approximating the I/O Complexity of One-Shot Red-Blue Pebbling. In *SPAA '16*. ACM, 161–163.

[24] Ceph. 2016. *Ceph Erasure Code.* https://docs.ceph.com/en/latest/rados/operations/erasure-code/

[25] G. J. Chaitin. 1982. Register Allocation & Spilling via Graph Coloring. (1982), 98–105.

[26] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. 1981. Register allocation via coloring. *Computer Languages* 6, 1 (1981), 47–57.

[27] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. 2005. The smallest grammar problem. *IEEE Trans. on Information Theory* 51, 7 (2005), 2554–2576.

[28] S. Coleman and K. S. McKinley. 1995. Tile Size Selection Using Cache Organization and Data Layout. In *PLDI '95*. ACM, 279–290.

[29] J. Cook, R. Primmer, and A. de Kwant. 2013. Comparing cost and performance of replication and erasure coding. *CoRR* abs/1308.1887 (2013). http://arxiv.org/abs/1308.1887

[30] S. A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *STOC '71*. ACM, 151–158.

[31] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. 2004. Row-Diagonal Parity for Double Disk Failure Correction. In *FAST '04*. USENIX Association.

[32] D. Coutts, R. Leshchinskiy, and D. Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. In *ICFP '07*. ACM, 315–326.

[33] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *Trans. on Programming Languages and Systems* 13, 4 (Oct 1991), 451–490.

[34] E. D. Demaine and Q. C. Liu. 2017. Inapproximability of the Standard Pebble Game and Hard to Pebble Graphs. In *WADS '07*. Springer, 313–324.

[35] E. D. Demaine and Q. C. Liu. 2018. Red-Blue Pebble Game: Complexity of Computing the Trade-Off between Cache Size and Memory Transfers. In *SPAA '18*. ACM, 195–204.

[36] A. P. Ershov. 1958. On Programming of Arithmetic Operations. *Commun. ACM* 1, 8 (Aug. 1958), 3–6.

[37] M. Farach-Colton and V. Liberatore. 2000. On Local Register Allocation. *Journal of Algorithms* 37, 1 (2000), 37–65.

[38] A. Fog. 2021. *The microarchitecture of Intel, AMD, and VIA CPUs.* https://www.agner.org/optimize/microarchitecture.pdf

[39] M. R. Garey and D. S. Johnson. 1979. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co.

[40] L. George and A. W. Appel. 1996. Iterated Register Coalescing. *ACM Trans. Program. Lang. Syst.* 18, 3 (May 1996), 300–324.

[41] A. Gill, J. Launchbury, and S. L. P. Jones. 1993. A Short Cut to Deforestation. In *FPCA '93*. ACM, 223–232.

[42] A. González, F. Latorre, and G. Magklis. 2010. *Processor Microarchitecture: An Implementation Perspective.* Morgan & Claypool Publishers.

[43] D. Grund and S. Hack. 2007. A Fast Cutting-Plane Algorithm for Optimal Coalescing. In *CC '07*. Springer, 111–125.

[44] S. Hack, D. Grund, and G. Goos. 2006. Register Allocation for Programs in SSA-Form. In *CC '06*. Springer, 247–262.

[45] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. 2014. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro* 34, 2 (2014), 6–20.

[46] J. L. Hennessy and D. A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc.

[47] J-W. Hong and H. T. Kung. 1981. I/O Complexity: The Red-Blue Pebble Game. In *STOC '81*. ACM, 326–333.

[48] C. Huang, J. Li, and M. Chen. 2007. On Optimizing XOR-Based Codes for Fault-Tolerant Storage Applications. In *ITW '07*. 218–223.

[49] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. 2012. Erasure Coding in Windows Azure Storage. In *USENIX ATC'12*. USENIX.

[50] C. Huang and L. Xu. 2003. *Fast software implementation of finite field operations.* Technical Report. Washington University.

[51] C. Huang and L. Xu. 2008. STAR : An Efficient Coding Scheme for Correcting Triple Storage Node Failures. *IEEE Trans. Comput.* 57, 7 (2008), 889–901.

[52] Intel. [n.d.]. *Intelligent Storage Acceleration Library.* https://github.com/intel/isa-l/

[53] Intel. [n.d.]. *Intelligent Storage Acceleration Library (version 2.30.0).* https://github.com/intel/isa-l/releases/tag/v2.30.0

[54] Intel. 2017. *ISA-L performance report.* https://01.org/intel%C2%AE-storage-acceleration-library-open-source-version/documentation/documentation

[55] Intel. 2020. *Intel 64 and IA-32 architectures optimization reference manual.* https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

[56] Intel. 2021. *Intel Architecture Instruction Set Extensions Programming Reference.* https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html

[57] S. Kalcher and V. Lindenstruth. 2011. Accelerating Galois Field Arithmetic for Reed-Solomon Erasure Codes in Storage Applications. In *CLUSTER '11*. 290–298.

[58] Richard M. Karp. 1972. *Reducibility among Combinatorial Problems.* Springer, 85–103.

[59] C. K. Koc and T. Acar. 1998. Montgomery Multiplication in GF(2k). *Designs, Codes and Cryptography* 14, 1 (1998), 57–69.

[60] T. Kranz, G. Leander, K. Stoffelen, and F. Wiemer. 2017. Shorter Linear Straight-Line Programs for MDS Matrices. *IACR Trans. on Symmetric Cryptology* 2017, 4 (2017), 188–211.

[61] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler. 2019. Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication. In *SC' 19*. ACM, Article 24.

[62] F. L. Gall. 2014. Powers of Tensors and Fast Matrix Multiplication. In *ISSAC '14*. ACM, 296–303.

[63] J. Lacan and J. Fimes. 2004. Systematic MDS erasure codes based on Vandermonde matrices. *IEEE Communications Letters* 8, 9 (2004), 570–572.

[64] M. D. Lam, E. E. Rothberg, and M. E. Wolf. 1991. The Cache Performance and Optimizations of Blocked Algorithms. In *ASPLOS '91*. ACM, 63–74.

[65] S. Lang. 1986. *Introduction to Linear Algebra.* Springer.

[66] R. Larrieu. 2019. *Fast finite field arithmetic.* Ph.D. Dissertation. University of Paris-Saclay, France.

[67] N. J. Larsson and A. Moffat. 1999. Offline dictionary-based compression. In *DCC'99*. 296–305.

[68] J. Lee, H. Kim, and R. Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Trans. Archit. Code Optim.* 9, 1, Article 2 (March 2012).

[69] T. Lengauer and R. E. Tarjan. 1980. The space complexity of pebble games on trees. *Inform. Process. Lett.* 10, 4 (1980), 184–188.

[70] S. Ling and C. Xing. 2004. *Coding Theory: A First Course.* Cambridge University Press.

[71] J. W. H. Liu. 1986. On the Storage Requirement in the Out-of-Core Multifrontal Method for Sparse Factorization. *ACM Trans. Math. Softw.* 12, 3 (Sept. 1986), 249–264.

[72] J. Luo, M. Shrestha, L. Xu, and J. S. Plank. 2014. Efficient Encoding Schedules for XOR-Based Erasure Codes. *IEEE Trans. Comput.* 63, 09 (2014), 2259–2272.

[73] F. MacWilliams and N. Sloane. 1977. *The Theory of Error-Correcting Codes.* Elsevier.

[74] E. D. Mastrovito. 1989. VLSI designs for multiplication over finite fields GF(2m). In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes.* Springer, 297–309.

[75] S. Muchnick. 1998. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc.

[76] I. Nakata. 1967. On Compiling Algorithms for Arithmetic Expressions. *Commun. ACM* 10, 8 (Aug. 1967), 492–494.

[77] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. 2013. The Quantcast File System. *VLDB Endow.* 6, 11 (Aug. 2013), 1092–1101.

[78] C. Paar. 1997. Optimized arithmetic for Reed-Solomon encoders. In *IEEE Intern. Symp. on Information Theory.* 250–250.

[79] P.R. Panda, H. Nakamura, N.D. Dutt, and A. Nicolau. 1999. Augmenting loop tiling with data alignment for improved cache performance. *IEEE Trans. Comput.* 48, 2 (1999), 142–149.

[80] P. A. Papp and R. Wattenhofer. 2020. On the Hardness of Red-Blue Pebble Games. In *SPAA '20.* ACM, 419–429.

[81] F. M. Q. Pereira and J. Palsberg. 2005. Register Allocation Via Coloring of Chordal Graphs. In *APLAS '05.* Springer, 315–329.

[82] J. Plank. 2008. The RAID-6 liberation codes. In *FAST '08.* 97–110.

[83] J. Plank, K. Greenan, and E. L. Miller. 2013. Screaming Fast Galois Field Arithmetic Using Intel SIMD Extensions. In *FAST'13.*

[84] I. S. Reed and G. Solomon. 1960. Polynomial Codes Over Certain Finite Fields. *J. Soc. Indust. Appl. Math.* 8, 2 (1960), 300–304.

[85] A. Reyhani-Masoleh, M. Taha, and D. Ashmawy. 2018. Smashing the Implementation Records of AES S-box. *IACR Trans. on Cryptographic Hardware and Embedded Systems* 2018, 2 (2018), 298–336.

[86] G. Rivera and C.-W. Tseng. 1999. A Comparison of Compiler Tiling Algorithms. In *CC '99.* Springer, 168–182.

[87] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *POPL '88.* ACM, 12–27.

[88] J. E. Savage. 1995. Extending the Hong-Kung Model to Memory Hierarchies. In *COCOON '95.* Springer, 270–281.

[89] V. Schneider. 1971. On the number of registers needed to evaluate arithmetic expressions. *BIT Numerical Mathematics* 11, 1 (01 Mar 1971), 84–93.

[90] R. Sethi. 1973. Complete Register Allocation Problems. In *STOC '73.* ACM, 182–195.

[91] R. Sethi. 1975. Complete Register Allocation Problems. *SIAM J. Comput.* 4, 3 (1975), 226–248.

[92] R. Sethi and J. D. Ullman. 1970. The Generation of Optimal Code for Arithmetic Expressions. *J. ACM* 17, 4 (Oct. 1970), 715–728.

[93] A. Shenoy. 2015. The Pros and Cons of Developing Erasure Coding and Replication Instead of Traditional RAID in Next-Generation Storage Platforms. (2015). https://www.snia.org/educational-library/pros-and-cons-developing-erasure-coding-and-replication-instead-traditional-raid SDC '15.

[94] G.E. Shilov. 1977. *Linear Algebra.* Dover Publications, Inc.

[95] K. Stoffelen. 2016. Optimizing S-Box Implementations for Several Criteria Using SAT Solvers. In *FSE '16.* Springer, 140–160.

[96] A. Takano and E. Meijer. 1995. Shortcut Deforestation in Calculational Form. In *FPCA '95.* ACM, 306–313.

[97] Q. Q. Tan and T. Peyrin. 2019. Improved Heuristics for Short Linear Programs. *IACR Trans. on Cryptographic Hardware and Embedded Systems* 2020, 1 (2019), 203–230.

[98] Y. Uezato. 2021. Accelerating XOR-based Erasure Coding using Program Optimization Techniques. arXiv:2108.02692 [cs.PL]

[99] Y. Uezato. 2021. *Author's Github Repository.* https://github.com/yuezato/xorslp_ec

[100] P. Wadler. 1989. Theorems for Free!. In *FPCA '89.* ACM, 347–359.

[101] P. Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (1990), 231–248.

[102] H. Weatherspoon and J. Kubiatowicz. 2002. Erasure Coding Vs. Replication: A Quantitative Comparison. In *IPTPS '01.* Springer, 328–338.

[103] K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. 2003. A Comparison of Empirical and Model-Driven Optimization. In *PLDI '03.* ACM, 63–76.

[104] T. Zhou and C. Tian. 2020. Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques. *ACM Trans. Storage* 16, 1, Article 7 (2020).

# Appendix: Artifact Description/Artifact Evaluation

## SUMMARY OF THE EXPERIMENTS REPORTED

We evaluated our library by comparing it with ISA-L (version: 2.30.0, https://github.com/intel/isa-l/releases/tag/v2.30.0) and the result of the state-of-the-art study (https://dl.acm.org/doi/10.1145/3375554) that is based on a similar method to ours.

For measuring performance, we ran our encoding and decoding programs for 10MB randomly generated data and measured the average coding throughputs. We wrote the average throughput of them into our paper. We measured the performance of ISA-L in our environment in the same manner.

We note that the authors of the study—doi/10.1145/3375554—have not opened their implementation; therefore, we could not compare our performance and theirs by running programs on our environment. To address this situation, we prepared two environments, as described below, that are close to the environments reported in their paper.

Our first environment is a MacBook 2017 Pro—Intel Core i7(7567U) CPU (Base: 3.5GHz, Turbo: 4.0GHz), 16GB LPDDR3 memory, and 10.15.7 OS X. In this environment, we compiled our library by using rustc-1.50.0. We compiled ISA-L by using Apple clang version 12.0.0 (clang-1200.0.32.27).

Our second environment is a PC with AMD Ryzen 5 2600 (Base: 3.4GHz, Turbo: 3.9GHz), 48GB DDR4, and Arch Linux (Kernel: 5.10.4). In this environment, we compiled our library by using rustc-1.50.0. We compiled ISA-L by using GCC 10.2.0.

*Author-Created or Modified Artifacts:*

```
Persistent ID: DOI:
↪ https://doi.org/10.5281/zenodo.5167006    Github:
↪ https://github.com/sc2021anonym/slp-ec
Artifact name: XORSLP\_EC
```

## BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

*Relevant hardware details:* PC1(MacBook 2017 Pro, Intel Core i7(7567U) (Base: 3.5GHz, Turbo: 4.0GHz), 16GB LPDDR3), PC2(AMD Ryzen 5 2600 (Base: 3.4GHz, Turbo: 3.9GHz), 48GB DDR4, and Arch Linux (Kernel: 5.10.4))

*Operating systems and versions:* 10.15.7 OS X, Arch Linux 1.4 running Linux kernel 5.10.4

*Compilers and versions:* rust v1.50.0, clang v12.0.0, gcc 10.2.0

*Libraries and versions:* ISA-L v2.30.0

*Key algorithms:* For our library: grammar compression, deforestation, pebble game; (For ISA-L) matrix multiplication, finite field arithmetic.

*Input datasets and versions:* randomly generated 10MB arrays were used for encoding and decoding. Our library and ISA-L do not affect by the contents of arrays.

*URL to output from scripts that gathers execution environment information.*

```
https://github.com/sc2021anonym/slp-ec/blob/main/our⌋
↪ \_environments.txt
```