Accelerating XOR-based Erasure Coding using Program Optimization Techniques

Yuya Uezato
Dwango, Co., Ltd.
Tokyo, 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 optimization techniques. Our optimization flow is three-fold: 1) reducing operations 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 ~8.92 GB/s throughput.

1 INTRODUCTION
Assuring 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 [101]. 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 the 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-array of \( \mathbb{F}_{2^8} \). RS(n, p) encodes an N-byte data D using an \( (n + p) \times n \) Vandermonde matrix \( V \in \mathbb{F}_{2^8}^{(n+p)\times n} \), which is key for decoding as we will see below, as follows:

\[
\begin{pmatrix}
V
\end{pmatrix}
\begin{pmatrix}
\tilde{d}_1 \\
\vdots \\
\tilde{d}_n
\end{pmatrix}
= 
\begin{pmatrix}
\tilde{b}_1 \\
\vdots \\
\tilde{b}_{n+p}
\end{pmatrix}
\]

where

- \( \tilde{b}_i \) is the MM over \( \mathbb{F}_{2^8} \);
- \( \tilde{d}_i \) is i-th \( \frac{n}{p} \)-bytes block of D;
- \( \tilde{b}_i \) is \( \frac{n}{p} \)-bytes coded block.

We store an encoded block \( \tilde{b}_i \) to a node \( n_i \) of a system with \( n + p \) nodes. For decoding, we gather \( n \)-blocks \( B = (\tilde{b}_i, \tilde{b}_{i+1}, ..., \tilde{b}_{i+n}) \) from alive nodes. The \( (n \times n) \)-submatrix \( M \) of \( V \) obtained by extracting row-vectors at \( (i_1, i_2, ..., i_n) \) satisfies \( B = M \tilde{x}_i \), \( \tilde{x}_i \) is the inverse \( M^{-1} \) of \( M \). Ceph [24] offers RS[10, 4] in [54]. In our evaluation at §7, RS[10, 4] encoding throughput for RS[10,4] in [54]. In our evaluation at §7, 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 for 6.7 GB/s.

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 SIMD instruction performance [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 for 6.7 GB/s.

(2) XOR-based EC [13, 74, 103] converts MM over \( \mathbb{F}_{2^8} \) to MM over \( \mathbb{F}_{2^8} \) where \( \mathbb{F}_{2^8} \) is the finite field of the bits \( \{0, 1\} \). This approach depends on the following two properties:

(i) There is an isomorphism \( \mathfrak{B} : \mathbb{F}_{2^8} \cong \mathbb{F}_{2^8} \times \mathbb{F}_{2^8} \) from bytes to 8 × 8 matrices over \( \mathbb{F}_{2^8} \) such that: \( \forall x, y \in \mathbb{F}_{2^8} \), \( x \in \mathbb{F}_{2^8}, y = \mathfrak{B}^{-1}(x \cdot \mathfrak{B}(y)) \).

(ii) We can calculate the above \( V \cdot \mathfrak{B}_D \) without the finite field multiplication of \( \mathbb{F}_{2^8} \), extending \( \mathfrak{B} \) to matrices as follows:

\[
V \cdot \mathfrak{B}_D = \mathfrak{B}^{-1}(V \cdot \mathfrak{B}(D))
\]

Since the addition (resp. multiplication) of \( \mathbb{F}_{2^8} \) is the bit XOR (resp. bit AND), MM over \( \mathbb{F}_{2^8} \) is just array XORs, as presented below:

\[
\begin{pmatrix}
11000000 \\
00111110 \\
00111101
\end{pmatrix}
\cdot
\begin{pmatrix}
\tilde{d}_1 \\
\vdots \\
\tilde{d}_7
\end{pmatrix}
= 
\begin{pmatrix}
\tilde{d}_1 + \tilde{d}_2 \\
\vdots \\
\tilde{d}_3 + \tilde{d}_4 + \tilde{d}_5 + \tilde{d}_6
\end{pmatrix}
\]
MM over $\mathbb{F}_2$ is easy-to-implement. Thanks to its implementability, this method was proposed in VLSI to realize finite field arithmetic on a small circuit [74]. This method is currently receiving attention since array XORs $d_i \oplus d_j$ are quickly executed via recent SIMD instructions [83]. In exchange for the ease of implementation, the obtained coding matrix $V \in \mathbb{F}_2^{p \times q}$ is much larger than the original coding matrix $V \in \mathbb{F}_2^{p \times q}$, and $V'$ needs more (but simple) operations of $\mathbb{F}_2^p$ than those of $\mathbb{F}_2^p$ in $V$.

Recently, Zhou and Tian published an invaluable study [103] 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 $\mathbb{F}_2$ as straight-line programs (SLPs), a classical compiler theory tool [2, 3], as follows:

$$
\begin{bmatrix}
1100000 \\
0011110 \\
0011101
\end{bmatrix}_{\mathbb{F}_2^2} \xrightarrow{\begin{pmatrix} \hat{a} \\ \hat{b} \\ \vdots \\ \hat{g} \end{pmatrix}} P : \begin{pmatrix} v_1 := a \oplus b; \\ v_2 := c @ d @ e @ f; \\ v_3 := e @ d @ e @ g; \\ \text{ret}(v_1, v_2, v_3) \end{pmatrix}
$$

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 theory of programming. This is the crucial difference between our study and that of Zhou and Tian [103], where they treated topics directly on matrices $\mathbb{F}_2$, and introduced ad-hoc constructions, without sophisticated results of program optimization.

Our Goal. The goal of this paper is to provide an efficient EC library importing various programmer-friendly optimization methods from theory of programming. Technically, we implement our optimizer as a translator, which 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:

$$
\begin{align*}
\lambda &:= c @ d @ e; \\
\lambda &:= \bigoplus (c, d, e); \\
v_1 &:= a @ b; \\
v_2 &:= c @ d @ e @ f; \\
v_3 &:= e @ d @ e @ g; \\
\text{ret}(v_1, v_2, v_3) &:= \text{ret}(v_1, v_2, v_3)
\end{align*}
$$

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 $c@d@e$ and replaces it with a new variable $\lambda$. It reduces the seven XORs to five and speeds up $\frac{7}{5} \approx 30\%$.

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 [100]. 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 (one operator and no functions).

In our example, $c@d@e$ invokes six memory accesses because $c@d$ invokes three—loading $c$, $d$ and storing the result to an intermediate array $I_{c,d}$. The remaining $I_{d,e}$ also invokes three. By fusing the two XORs to $\bigoplus (c, d, e)$, we eliminate (deforest) the intermediate array $I_{d,e}$. 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) maximum 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, $\lambda @ f$ and $\lambda @ g$. 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 optimize 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: 4.03GB/s | Comp.: 4.36GB/s | Fuse: 7.50GB/s | Sched.: 8.92GB/s |
|-------------------------------------------------------------|-----------------|-----------------|----------------|-----------------|

where Base runs unoptimized SLPs that are obtained from matrices, such as the above $P$. Interestingly, although (Xor)RePair reduces about 60% XORs on average (as we will see in §7), the summary says the compressing effect is small. This is because (Xor)RePair generates cache-poor compressed SLPs, and this observation will be substantiated by the cache analysis using our pebble game. The sole application of (Xor)RePair is not as 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 [103]. 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) of XOR of reduced = 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 reorders XORs locally without considering pebble games, is not quite effective; the throughput of optimized throughput of original ≈ 101% in [103]. 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 the 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 heuristics 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 used to analyze a fixed algorithm and program, and study the transfer cost between fast and slow devices. This 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 the 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 heuristics 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.

4 Reducing XOR Operations

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 branches, loops, and procedures [1, 3, 21]. An SLP is a tuple \( (\gamma, \mathcal{E}, \mathcal{C}, g, \circ) \) where \( \gamma \) is a set of variables, \( \mathcal{E} \) is a set of constants, \( s \) is a sequence of instructions (i.e., the body of the program), \( g \) is a sequence of variables returned by the program, and \( \circ \) is a binary operator. The set of instructions \( \text{instr} \) is defined by the following BNF:

\[
\begin{align*}
\text{instr} & ::= \gamma \rightarrow \text{expr} \\
\text{expr} & ::= \gamma \mid \mathcal{E} \mid \text{expr} \circ \text{expr}
\end{align*}
\]

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 \( \text{SLP}_\oplus \) for the class.

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

\[
P(a, b, c, d: \text{[byte]}) \{
\begin{align*}
v_1 & \leftarrow a \oplus b; \\
v_2 & \leftarrow b \oplus c \oplus d; \\
v_3 & \leftarrow v_1 \oplus v_2;
\end{align*}
\]

\[
\text{return}(v_2, v_3, v_1);
\]

where for the left SLP \( \gamma' = \{v_1, v_2, v_3\}, \mathcal{E} = \{a, b, c, d\}, \) and \( g = (v_2, v_3, v_1), \) and for the array program the infix function xor performs XOR element-wise for input byte arrays. On the basis of this idea, we have the following correspondence:

- constants of \( \text{SLP}_\oplus \) ⇔ program input arrays,
- variables of \( \text{SLP}_\oplus \) ⇔ arrays allocated at runtime.

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

\[
\begin{align*}
\text{SLP}_P & \mid v_1 \text{-value} \mid v_2 \text{-value} \mid v_3 \text{-value} \\
v_1 & \leftarrow a \oplus b; \\
v_2 & \leftarrow b \oplus c \oplus d; \\
v_3 & \leftarrow v_1 \oplus v_2; \\
\text{ret}(v_2, v_3, v_1)
\end{align*}
\]

Notation. We use \( [\cdot] \) to denote the returned values of a program; e.g., \( [P] = \{(b, c, d), \{a, c, d\}, \{a, b\}\} \). We use \#\( \cdot \) to denote the size of a program, i.e., the number of XORs; e.g., \#\( P \) = 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 \( \text{SLP}_\oplus \) problem

For a given \( P \in \text{SLP}_\oplus \), we find \( Q \in \text{SLP}_\oplus \) that satisfies \( [P] = [Q] \) and minimizes \#\( Q \).

We cannot solve this problem in polynomial time unless \( \text{P}=\text{NP} \) 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 Calculation. Let us consider the following three equivalent SLPs:

\[
\begin{align*}
\text{SLP}_{P_0} & \mid v_1 \leftarrow a \oplus b; \\
\text{SLP}_{P_1} & \mid v_2 \leftarrow a \oplus b; \\
\text{SLP}_{P_2} & \mid v_3 \leftarrow a \oplus b;
\end{align*}
\]

where \( [P_0] = [P_1] = [P_2], \#\( P_0 \) = 8, \#\( P_1 \) = 5, \) and \( \#\( P_2 \) = 4 \). We notice that, in \( P_2 \), the \( \oplus \) operator is effectively used to compute \( v_4 \). Indeed, we can show that there is no \( Q \) with \( \#\( Q \) < 4 \)
and $[P_0] = [Q]$ enumerating $Q \in \text{SLP}_b$. Moreover, there is no $Q$ with $\#Q < 5$ and $[P_0] = [Q]$ unless using the $\ominus$-cancellativity.

This example emphasizes the $\ominus$-cancellativity is essential to shorten $\text{SLP}_b$. 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 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 recursive pairing). 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 SLP $P_0$.

$$
\begin{align*}
v_1 & \leftarrow a \oplus b; \\
v_2 & \leftarrow a \oplus b \oplus c; \\
v_3 & \leftarrow a \oplus b \oplus c \oplus d; \\
v_4 & \leftarrow b \oplus c \oplus d; \\
\text{ret}(v_1, v_2, v_3, v_4) & \leftarrow \text{ret}(v_1, v_2, v_3, v_4).
\end{align*}
$$

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 $\preceq$ 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 $t < c$ for a temporal variable $t$ and a constant $c$.

Now, we define RePAIR using Pair as its subroutine.

**RePAIR**

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 $\preceq$.

**Example.** Let us apply RePAIR to the above $P_0$ omitting ret:

$$
\begin{align*}
v_1 & \leftarrow a \oplus b; \\
v_2 & \leftarrow a \oplus b \oplus c; \\
v_3 & \leftarrow a \oplus b \oplus c \oplus d; \\
v_4 & \leftarrow b \oplus c \oplus d; \\
\text{ret}(v_1, v_2, v_3, v_4) & \leftarrow \text{ret}(v_1, v_2, v_3, v_4).
\end{align*}
$$

At the first step, the pairs $(a, b)$ and $(b, c)$ appear three times, and we choose $(a, b)$ because $(a, b) \prec (b, c)$. 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 $\langle w \rangle$ to denote the value of a variable $w$.

**Rebuild $(v)$**

**Initialization:** Let $\text{rem} := \langle v \rangle$ and $S := \emptyset$. rem denotes a set of constants to be eliminated by XORing existing temporal variables.

**Loop:**

1. If we cannot shorten $\text{rem}$ (i.e., there is no temporal variable $t$ such that $\text{rem} \oplus \langle t \rangle < |\text{rem}|$), we return $\text{rem} \cup S$ as the new definition of $v$.
2. Otherwise, we choose a temporal variable $t$ that minimizes $|\text{rem} \oplus \langle t \rangle|$ and update $\text{rem} := \text{rem} \oplus \langle t \rangle$ and $S := S \cup \{t\}$. If there are multiple candidates, we choose the smallest one for $\prec$.

For example, applying Rebuild $(a)$ to the following left SLP, we obtain a new equivalent definition $a_4 := a \oplus t_3$.

$$
\begin{align*}
t_1 & \leftarrow a \oplus b; \\
t_2 & \leftarrow t_1 \oplus c; \\
t_3 & \leftarrow t_2 \oplus d; \\
\text{rem}(a) & \leftarrow t_3 \oplus b \oplus c \oplus d; \\
\text{ret}(a, t_3) & \leftarrow \text{ret}(a, t_3).
\end{align*}
$$

Augmenting RePAIR with Rebuild, 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 $P_0$.

Let us apply XorRePAIR to the example $P_0$.

$$
\begin{align*}
t_1 & \leftarrow a \oplus b; \\
t_2 & \leftarrow t_1 \oplus c; \\
t_3 & \leftarrow t_2 \oplus d; \\
\text{rem}(a) & \leftarrow t_3 \oplus b \oplus c \oplus d; \\
\text{ret}(a, t_3) & \leftarrow \text{ret}(a, t_3).
\end{align*}
$$

First, we reach the above left form. Next, we update $a_4$ as $a_4 := a \oplus t_3$ since Rebuild $(a) = \{a, t_3\}$ shortens the definition of $a_4$. Finally, we perform Pair $(a, t_3)$ and obtain the shortest SLP with 4 XORs as we have seen in §4.2. Clearly, XorRePAIR runs in polynomial time.

**Related approaches.** We note that pairing is sometimes called 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
We prove the intractability in Appendix §A by reducing the Vertex Cover Problem (VCP) to the above one. Although using the VCP is the same as the construction given by Boyar et al. [18] to prove the intractability of the shortest $\text{SLP}_\oplus$ problem, we need to deeply analyze $\text{SLP}_\oplus$ and $\text{SLP}_\oplus$ because a normalization from SLPs to SLPs of the binary form as follows, which is the key in the proof of [18], does not work well for $\text{SLP}_\oplus$ and $\#_M$:

$$v \leftarrow a \oplus b \oplus c; \quad \Rightarrow \quad v' \leftarrow a \oplus b;$$

$$v \leftarrow v' \oplus c;$$

The above normalization to the binary form does not change $\#_M$; however, it increases $\#_M(4 \to 6)$. This normalization brought a significantly useful syntactic property on $\text{SLP}_\oplus$ in [18]. Since we cannot count on such the property, in Appendix §A, we give a more detailed and elaborated construction.

### 5.2 XOR Fusion

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

**Theorem 2.** Let $P$ be an $\text{SLP}_\oplus$, and $Q$ be an $\text{SLP}_\oplus$ 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 $A$.

$$A : \begin{align*}
    v_2 & \leftarrow a \oplus b \oplus c \oplus d \oplus e \oplus f; \\
    v_3 & \leftarrow a \oplus b \oplus c \oplus d \oplus e \oplus g; \\
    \text{compress} & \quad \downarrow B: \quad v_1 \leftarrow (a, b, c, d, e); \\
    v_3 & \leftarrow v_1 \oplus f; \\
    C : \quad v_2 & \leftarrow (a, b, c, d, e, f); \\
    v_3 & \leftarrow (a, b, c, d, e, g);
\end{align*}$$

disallow

where $\#_M(A) = 30$ since one XOR issues three accesses, $\#_M(B) = 12$, and $\#_M(C) = 14$. Therefore, if we would allow to unfold $v_1$ in $B$, then fusing increases memory accesses. 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.
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 compared to that of main memory, we can only put a few arrays if the given data to be encoded is large. For example, if the size of L1 cache is 32KB, which is a typical L1 cache size, and a user encodes 1MB data on RS(10, 4), the input data is divided into 8-10 arrays of \( \frac{1\text{MB}}{32\text{KB}} \sim 12\text{KB} \); therefore, the cache can only hold two arrays at once, and thus the cache performance becomes poor.

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 arrays into arrays of \( B \) bytes.

**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 \( v_2 \). These operations change \( C \) as follows:

\[
\text{empty } \xrightarrow{\triangleright} A \xrightarrow{B} A B \xrightarrow{v_2} A B v_1
\]

where we use \( \triangleright \) to denote a loading from memory and \( \rightsquigarrow \) to an allocation in the cache.

For the second XOR, we load \( C \) and \( D \) and allocate \( v_2 \) as follows:

\[
A B v_1 \xrightarrow{C} D \xrightarrow{v_2} A B v_1 C D v_2.
\]

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

\[
A B v_1 C D v_2 v_1 E F \xrightarrow{v_2} A B C D v_2 v_1 E F v_3.
\]

where we use \( \rightsquigarrow \) to denote a position update in the cache.

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

\[
A B C D v_2 v_1 E F v_3 v_2 \xrightarrow{G} A B C D v_2 v_1 E F v_3 G A.
\]

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

\[
B C D v_2 v_1 E F v_3 G A \xrightarrow{v_4} C D v_2 v_1 E F v_3 G A v_4
\]

where we write \( \Rightarrow \) for evictions from the cache to memory.

Finally, we change \( C \) as follows in the fifth XOR:

\[
C D v_2 v_1 E F v_3 G A v_4 \xrightarrow{v_3} v_1 \xrightarrow{v_2} D v_2 E F G A v_1 v_3 v_4 v_5.
\]

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

\[
\text{CCap}(P ; \text{SLP})
\]

\[
\text{CCap}(P) \text{ denotes the minimum cache capacity where we can run } P \text{ without cache reloading.}
\]

We can confirm \( \text{CCap}(P_{eg}) = 10 \). Indeed, if we use the cache of capacity \( 9 \), we need to replace \( A \) for \( G \) in the fourth XOR \( v_4 \leftarrow \odot (v_3, G, A) \), and this replacement leads to reloading \( A \) as follows:

\[
A B C D v_2 v_1 E F v_3 G A \xrightarrow{v_4} A B C D v_2 v_1 E F v_3 G A
\]

where we write \( \Rightarrow \) for the replacement that evicts \( x \) and loads \( y \).

We also consider the number of I/O transfers required by SLPs.
It is clear that \( \text{IOcost}(P_{eg}, 10) = 7(\text{of } \oplus) + 2(\text{of } \oplus) = 9 \). This measure is useful when cache capacity is determined by hardware. For example, it is one of the standard parameter on recent CPUs that cache size is 32KB and cache block size is 64B. The cache of such CPUs can hold 512 blocks maximally, and thus we optimize \( \text{IOcost}(P, 512) \).

Hereafter, as an example, we consider that the cache can hold eight blocks maximally. We can easily check \( \text{IOcost}(P_{eg}, 8) = 13 \) running \( P_{eg} \) with the cache of capacity 8.

### 6.3 Optimizing SLP via Register Allocation

To reduce \( \text{CCap} \) and \( \text{IOcost} \), 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 of 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 \( P_{eg} \):

\[
\begin{align*}
1) & \quad v_1 \leftarrow A \oplus B; \\
2) & \quad v_2 \leftarrow C \oplus D; \\
3) & \quad v_3 \leftarrow (v_1, E, F); \\
4) & \quad v_4 \leftarrow (v_3, G, A); \\
5') & \quad v_1 \leftarrow (v_1, v_3, v_4); \\
6) & \quad \text{ret}(v_2, v_4, v_1);
\end{align*}
\]

Unlike 5) of \( P_{eg}, 5' \), we store the result \((v_1, v_3, v_4)\) to \( v_1 \) instead of \( v_3 \) of \( P_{eg} \) since \( v_1 \) is no more needed after 5')

Although register assignment reduces variables, \( \text{NVar}(P_{reg}) = 4 \), and I/O transfers, \( \text{IOcost}(P_{reg}, 8) = 12 \), it does not reduce \( \text{CCap} \) since \( \text{CCap}(P_{eg}) = 3 \) \text{CCap}(P_{eg}) = 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:

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 \( G \) be a CG. As the initialization step, for each leaf node \( t \), we put the same name pebble on \( t \). To represent this, we write \( G(t) = t \). 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

  \[
  n : p \leftarrow (p_1, p_2, \ldots, p_k)
  \]

  where \( n \) is a node of \( G \), \( p \) and \( p_i \) are pebbles, and \( \{ n_1 : G(n_1) = p_i \} \) equals to \( n \)'s children.

  - If \( p \) is a new pebble, we put \( p \) on \( n \) so that \( G(n) = p \).
  - 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.

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 \( \text{IOcost} \) and the abstract cache \( C \) where we can select elements to be evicted from \( C \) instead of the LRU rule.
Example. Let us consider the following winning strategy (with a return statement) of the above CG $G_{eq}$:

\[
\begin{align*}
1) & \quad q_1 : p_1 \leftarrow B \oplus A; \\
2) & \quad q_3 : p_2 \leftarrow \bigoplus (E, F, p_1); \\
3) & \quad q_4 : p_3 \leftarrow (A, G, p_2); \\
4) & \quad q_5 : p_1 \leftarrow \bigoplus (p_1, p_2, p_3); \\
5) & \quad v_2 : p_3 \leftarrow C \oplus D; \\
6) & \quad \text{return}(p_3, p_2, p_1);
\end{align*}
\]

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

\[
\begin{align*}
\text{empty} & \quad \frac{}{B \oplus A}; \\
\frac{}{A A p_1} & \quad \frac{B F}{P_1} \quad \frac{P_2}{P_3} \quad A E F p_1 p_2 \quad \frac{A G}{F} \quad \frac{P_3}{P_4}.
\end{align*}
\]

\[
p_1 A G p_2 p_3 \quad \frac{p_1}{p_1} \quad \frac{p_1}{p_1} \quad A G p_2 p_1 p_1 \quad \frac{C D}{A} \quad \frac{p_1}{p_1} \quad p_2 p_1 C D p_3
\]

where $\frac{y}{z}$ means the replacement that evicts $x$ and allocates $y$.

The pebble game immediately implies the following property.

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

**Notation:** Let $P$ and $Q$ be SLPs. If $Q$ is obtained from a winning strategy of the CG of $P$ in the above manner, we write $P \succeq Q$.

### 6.5 Intractability of Optimization Problems

Introducing the pebble game is not only useful for cache optimization but also useful to correctly refer established results of compiler construction and program analysis.

**Theorem 3.** Let $P$ be an SLP. All the following optimization problems cannot be solved in polynomial-time unless $P=NP$:

1. Finding $Q \in \text{SLP}_{\leq}$ that satisfies $P + Q$ and minimizes $NVar(Q)$.
2. Finding $Q \in \text{SLP}_{\leq}$ that satisfies $P + Q$ and minimizes $CCap(Q)$.
3. For a given cache capacity $c$, finding $Q \in \text{SLP}_{\leq}$ that satisfies $P + Q$ and minimizes $IOcost(Q, c)$.

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; namely, we use 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 classic 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 $G_{eq}$. We need to decide which root node is visited first; here, we choose $v_2$ on the basis of the total ordering $<$ defined in §4.3 since $v_2 < v_5$. We then visit the children $C$ and $D$ in this order since $C < D$. Using $<$ as the tie-breaker, we make the following postorder traversal:

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

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

\[
\begin{align*}
1) & \quad v_2 : p_1 \leftarrow C \oplus D; \\
2) & \quad v_1 : p_2 \leftarrow A \oplus B; \\
3) & \quad \text{CCap}(Q_{DFS}) = 3, \quad \text{CCap}(Q_{dfs}) = 7, \quad \text{IOcost}(Q_{dfs}, 8) = 10.
\end{align*}
\]

**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 $<$ as the tie-breaker.

We revisit the CG $G_{eq}$ as follows. On the initial state, $v_1$ and $v_2$ are ready with $\frac{|\{\{A, B, F\}\}|}{|\{\{A, B, F\}\}|} = \frac{2}{2}$ for $v_1$ and $\frac{|\{\{C, D\}\}|}{|\{\{C, D\}\}|} = \frac{2}{2}$ for $v_2$. We choose $v_1$ since $v_1 < v_2$, and the generated statement changes $C$ as the following right:

\[
v_1 : p_1 \leftarrow A \oplus B; \quad \text{empty} \quad \frac{\text{empty} B \oplus A_p_1}{} \quad \frac{A B p_1}{A B p_1};
\]

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

\[
\begin{align*}
& v_3 : p_2 \leftarrow \bigoplus (p_1, E, F); \\
& v_4 : p_3 \leftarrow \bigoplus (p_2, A, G); \\
& v_5 : p_4 \leftarrow \bigoplus (p_3, A, G); \\
& v_2 : p_3 \leftarrow C \oplus D; \\
& v_1 : p_1 \leftarrow A \oplus B;
\end{align*}
\]

It can be verified that $NVar(Q_{greedy}) = 3$, $CCap(Q_{greedy}) = 7$, and $IOcost(Q_{greedy}, 8) = 9$. 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 show our throughputs with Intel’s ISA-L [52] and the state-of-the-art study [103].

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 [98]. 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 [103], we cannot directly compare our methods and theirs by running programs. Thus, we borrow values from [103] 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 assoc.) and AMD Ryzen 1700X(3.8 GHz, 8 cores, 32KB cache, 64-bytes cache block, 8-way 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 \((10^4) = 1001\) decoding matrices obtained by removing 4 rows from the encoding matrix. We need the finite field \(\mathbb{F}_p\) 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 \(\mathbb{F}_p\) 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 \([63, 70, 73]\)) \(V\) of a \((14 \times 10)\) Vandermonde matrix given by the standard construction as follows:

\[
\begin{pmatrix}
1 & \alpha & \cdots & \alpha^9 \\
1 & \alpha^2 & \cdots & (\alpha^2)^9 \\
\vdots & \vdots & \ddots & \vdots \\
1 & \alpha^{14} & \cdots & (\alpha^4)^9
\end{pmatrix} = \begin{pmatrix} V_{10 \times 10} \\ M_{4 \times 10} \end{pmatrix} \text{ reduces } \begin{pmatrix} V_{10 \times 10} \\ M_{4 \times 10} \end{pmatrix} = \begin{pmatrix} \text{Ident}_{10 \times 10} \\ M_{4 \times 10}V_{10 \times 10} \end{pmatrix}
\]

where \(\alpha\) is a primitive element of our \(\mathbb{F}_p\) [84], and the reduced version \(V\) is the actual encoding matrix of ISA-L.

We write \(P_{\text{enc}}\) for the SLP that corresponds to the bitmatrix form \(V\) of our encoding matrix \(V\), as seen in §1. We write \(P_{\text{RS}}\) 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 \(B\). The first one xor1 performs the byte XORing element-wise. The second one xor32 performs the 32-byte XORing—mm256_xor, SIMD AVX2 instruction—element-wise. Such SIMD instructions are used in ISA-L and the previous study [103]. Furthermore, AVX2 is the standard instruction set for recent CPUs.

The power of SIMD is remarkable, as already reported in [83, 103]. This result also suggests that the bottleneck is shifted from the CPU to memory I/O by the SIMD instruction.

Despite our argument about cache efficiency in §6, the performances of small blocks—64, 128, 256, 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, 102]. 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):

| Throughput (GB/sec) | xor1 | xor32 |
|---------------------|------|-------|
| Blocksize           |      |       |
| intel               | 0.16 | 0.62  |
| amd                 | 0.17 | 0.67  |

The power of SIMD is remarkable, as already reported in [83, 103]. This result also suggests that the bottleneck is shifted from the CPU to memory I/O by the SIMD instruction.

Despite our argument about cache efficiency in §6, the performances of small blocks—64, 128, 256, 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, 102]. 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%   | RePair(P) | XorRePair(P) | Corresp. Value from [103] |
|--------|-----------|--------------|---------------------------|
| XOR Num \(\#_\text{rs}(\cdot)\) | 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:

\[
\text{Avg}\left(\frac{\#_\text{rs}(\cdot)}{\#_\text{rs}P} : P \in P_{\text{RS}}\right) = \begin{cases} 42.1\% & \text{if } C = \text{Repair}, \\ 40.8\% & \text{if } C = \text{XorRepair}. \end{cases}
\]

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 [103]. 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 little 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(·):

|         | Co(P)/P | Fu(P)/P | Fu(Co(P))/P | Fu(Co(Co(P)))/P |
|---------|---------|---------|-------------|-----------------|
| Avg%    | 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.

|        | Co(P)/P | Fu(P)/P | Fu(Co(P))/P | Fu(Co(Co(P)))/P |
|--------|---------|---------|-------------|-----------------|
| NVar   | 155%    | 100%    | 38.9%       | 24.5%           |
| CCap   | 98%     | 98.7%   | 57.2%       | 39.6%           |

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, . . . , 4K, and the table including 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 \(\frac{CCap(Fu(Co(Co(P))))}{CCap(P)} \sim 199\%\); therefore, we can say that the scheduling heuristic can suppress the side effects of XORRePair to some extent.

We consider why XORRePair significantly deteriorates 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].

7.4 Selecting Adequate Blocksize

As we have seen in §6, the block size \(B\) of the blocking technique is an essential optimization parameter. Although small blocks are supposed to enable the cache to hold all blocks, 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. Since we have already seen the coding performance of \(P_{enc}\) on various blocks, we first see the performance of the uncompressed but fused version \(P_{enc}^F\) of \(P_{enc}\) to check whether or not a similar tendency appears.

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

| Block size (byte) | 64 | 128 | 256 | 512 | 1K | 2K | 4K |
|------------------|----|-----|-----|-----|----|----|----|
| **intelfu**      | 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 NVar(\(P_{enc}^F\)) = 32 and CCap(\(P_{enc}^F\)) = 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 \(B \leq 512\) 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/2 blocks. Generally, the 32K bytes cache with 8 cache associativity has \(\frac{256}{2} = 4K\) cache sets where each cache set can hold 8 cache blocks. Accessing a cache block \(b\) whose start address is \(A(b)\), CPU tries to assign \(b\) to the \(A(b)\) mod 4K)-th cache set. If the cache set is full (i.e., it already has 8 cache blocks), CPU evicts the LRU cache block in the set to memory. Therefore, accessing two blocks \(b_1, b_2\) such that \(A(b_1) \equiv 4K\), \(A(b_2)\) may cause an eviction in a cache set.

If we take the 4K-alignment strategy (i.e., locate all blocks on addresses divisible by 4K), the cache holds at most 8 blocks regardless of the size \(B\). To avoid the worst situation, several approaches have been proposed [64, 79]; however, optimally aligning blocks is a hard problem. We use a simple approach as follows: for an SLP whose constants are \(c_0, c_1, \ldots\), we allocate \(c_i\) so that \(A(c_i) \equiv 4K\) (i.e., \(B\)). We do the same for variables \(v_1, v_2, \ldots\). For example, when \(B = 1K\),

\[A(c_0) \equiv 4K, A(c_1) \equiv 4K 1K, A(c_2) \equiv 4K 2K, A(c_3) \equiv 4K 3K, \]

\[A(c_0) \equiv 4K 0, A(c_1) \equiv 4K 0, A(c_2) \equiv 4K 4K 1K, \ldots\]

This strategy is better than 4K-alignment since accessing to \(c_i\) and \(v_j\) never conflict when \(i \neq j\). In conclusion, 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 \(B\) 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 now 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 reach RAM [46, 55]. Even if the CPU pipeline maximally works, we need 150 + \(\frac{48}{3}\) 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 8-byte loads, then we need (151 \times 8)-cycles. This is the reason why 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 blocks of \(B\) bytes.
We should note that our greedy scheduling heuristic generates
where

Accelerating XOR-based Erasure Coding using Program Optimization Techniques Technical Report, 2021,

obtained by removing

hand, comparing the third and fourth columns, we tell that





where NVar(\(P_{\text{full}}\)) ~ 90 and CCap(\(P_{\text{full}}\)) ~ 170 for all the entries. We should note that our greedy scheduling heuristic generates different programs for each \(B\). However, for all \(B\), NVar(\(\cdot\)) is about 90, and CCap(\(\cdot\)) is about 170. The same is true for the DFS heuristics.

On the basis of the performance, hereafter we set \(B = 1\) on \textit{intel} and \(B = 2\) on \textit{amd} and use the DFS-based scheduling heuristics for comparison with ISA-L and the previous work.

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

### 7.5 Throughput Analysis

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

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{Block size} & \text{intel (greedy)} & \text{amd (greedy)} & \text{AMD (dfs)} & \text{intel (dfs)} \\
\hline
64 & 6.02 & 6.08 & 8.37 & 8.55 \\
128 & 7.61 & 7.32 & 7.15 & 7.58 \\
256 & 1.91 & 3.30 & 4.36 & 6.67 \\
512 & 2.29 & 4.00 & 6.02 & 7.61 \\
1K & 7.58 & 7.64 & 8.55 & 7.64 \\
2K & 8.55 & 7.32 & 7.15 & 7.58 \\
4K & 3.22 & 4.36 & 6.02 & 7.61 \\
\hline
\end{array}
\]

while we note that our scheduling heuristics do not affect the number of XORs and memory accesses, and we represent it by \(\sim\).

To maximize performance, we can say that compression, fusion, and scheduling are all necessary. Comparing the first and second columns, we tell that the number of memory accesses is more dominant than that of CCap on the performance. On the other hand, comparing the third and fourth columns, we tell that CCap certainly represents the 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

\[
\begin{align*}
\text{Case 2: Full Optimization.} & \quad \text{As we have seen above, small blocks may degrade the performance of blocked programs. Here we see the coding performance of fully optimized—compressed, fused, and scheduled—version of } P_{\text{enc}}, P_{\text{enc}}^{\text{Full}}, \text{ to check whether or not large blocks better for blocked programs than smaller ones.} \\
\hline
\text{Block size} & \text{intel (greedy)} & \text{amd (greedy)} & \text{AMD (dfs)} & \text{intel (dfs)} \\
\hline
64 & 6.02 & 6.08 & 8.37 & 8.55 \\
128 & 7.61 & 7.32 & 7.15 & 7.58 \\
256 & 1.91 & 3.30 & 4.36 & 6.67 \\
512 & 2.29 & 4.00 & 6.02 & 7.61 \\
1K & 7.58 & 7.64 & 8.55 & 7.64 \\
2K & 8.55 & 7.32 & 7.15 & 7.58 \\
4K & 3.22 & 4.36 & 6.02 & 7.61 \\
\hline
\end{align*}
\]

where NVar(\(P_{\text{full}}\)) ~ 90 and CCap(\(P_{\text{full}}\)) ~ 170 for all the entries. We should note that our greedy scheduling heuristic generates different programs for each \(B\). However, for all \(B\), NVar(\(\cdot\)) is about 90, and CCap(\(\cdot\)) is about 170. The same is true for the DFS heuristics.

On the basis of the performance, hereafter we set \(B = 1\) on \textit{intel} and \(B = 2\) on \textit{amd} and use the DFS-based scheduling heuristics for comparison with ISA-L and the previous work.

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

| \text{Block size} | \text{intel (greedy)} | \text{amd (greedy)} | \text{AMD (dfs)} | \text{intel (dfs)} |
|-------------------|----------------------|---------------------|----------------|------------------|
| 64                | 6.02                 | 6.08                | 8.37           | 8.55             |
| 128               | 7.61                 | 7.32                | 7.15           | 7.58             |
| 256               | 1.91                 | 3.30                | 4.36           | 6.67             |
| 512               | 2.29                 | 4.00                | 6.02           | 7.61             |
| 1K                | 7.58                 | 7.64                | 8.55           | 7.64             |
| 2K                | 8.55                 | 7.32                | 7.15           | 7.58             |
| 4K                | 3.22                 | 4.36                | 6.02           | 7.61             |

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 trend is consistent with \(P_{\text{enc}}\).

### 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 [103]. As the same as [103], we consider three kinds of codec: 4-parities RS\((d, 4)\), 3-parities RS\((d, 3)\), and 2-parities RS\((d, 2)\). We summarize main measures for each codec in Figure 1. These values correspond to the rightmost value of the above tables in §7.5.

![Figure 1: Values of \(\#_{\text{dec}}, \#_{\text{enc}}, \text{NVar}(\cdot), \text{CCap}(\cdot), \) of optimized coding SLPs for various codecs](image)

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\((8, 4)\), although RS\((8, 4)\) is better than RS\((10, 4)\) in terms of the measures in Figure 1. To encode or decode a given data of N-bytes, we run SLPs for \(8 \times 8\) input arrays of \(\frac{N}{256}\) bytes on RS\((8, 4)\). Similarly, we run SLPs for \(8 \times 8\) input arrays of \(\frac{N}{256}\) bytes on RS\((10, 4)\). The difference in Figure 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 \(\frac{N_{\text{dec}}}{256}\) on RS\((8, 4)\) is larger than that \(\frac{N_{\text{dec}}}{256}\) 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 difference between them in our EC library.
Our encoding and decoding matrices, which are sources of $P_{\text{enc}}$ and $P_{\text{dec}}$, 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 MM over finite fields algorithms, 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 $M; i.e., a \cdot b$ is computed by accessing $M[a][b]$. 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 $e_1, e_2 \in \mathbb{F}_p$, the number of 1 in the bitmatrix $e_1$ may be significantly larger than those of $e_2$. In a conclusion, we consider the performance gap between encoding and decoding in our library is intrinsic in XOR-based EC.

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

| RS(d, 4) (GB/sec) | Ours | ISA-L v2.30 | Values of [103] |
|------------------|------|-------------|-----------------|
| Enc Dec | 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 [103] |
| 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):

| RS(d, 2) (GB/sec) | Ours | ISA-L v2.30 | Values of [103] |
|------------------|------|-------------|-----------------|
| Enc Dec | 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 | 5.90 |

| RS(d, 2) (GB/sec) | Ours | ISA-L v2.30 | Values of [103] |
|------------------|------|-------------|-----------------|
| Enc Dec | Enc Dec | Enc Dec | Enc Dec |
| RS(8, 2) | 18.79 | 14.59 | 12.99 | 13.34 | 8.13 | 8.07 |
| RS(9, 2) | 18.93 | 14.27 | 11.85 | 12.03 | 8.34 | 8.04 |
| RS(10, 2) | 18.98 | 14.66 | 12.12 | 12.61 | 8.40 | 8.22 |

The column "Values of [103]"] consists of the best throughputs among results of the corresponding parameters in [103] 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 $\cdot \cdot$, $\cdot \cdot \cdot$, and $\cdot \cdot \cdot \cdot$ 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, grammar compression algorithm RePair, the functional program optimization technique deforestation, and the pebble game from program analysis. We extended RePair to our XORRePair to accommodate the cancellative property of XOR. We used the pebble game to model 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 thinking about using the multilevel pebble game introduced by Savage in [88] to accommodate the L2 and L3. As a related cache efficiency topic, we are 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/TechDoc/66568.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, 249–253.
[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 J. Proc. 1995. EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures. IEEE Trans. Comput. 44, 2 (1995), 192–202.
[13] J. Blömér, M. Kalfane, R. Karp, M. Karpinski, M. Lubo, and D. Zuckerman. 1995. An XOR-Based Erasure-Resilient Coding Scheme. ICS 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. Springer, 283–298.
[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. J. ACM 21, 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.
To show the NP-completeness of our problem, we use the classic Technical Report, 2021, Yuya Uezato

A PROOF OF THEOREM 1

Let $A.2$ Build SLP from Graph

Since VCP is NP-complete, OptVCP cannot be solved in polynomial time unless P=NP.

Below we reduce our optimization problem, the minimum memory access problem, to OptVCP.

A.2 Build SLP from Graph

Let $G$ be an undirected graph. Before building an SLP corresponding to $G$, we modify $G$ for our construction and proof as follows:

- For each node $a$ of $G$, we add two fresh nodes $\lambda_a$ and $\mu_a$ and add two edges $(a, \lambda_a)$ and $(a, \mu_a)$.
- We call added nodes $\lambda_a$ and $\mu_a$ local nodes.

The optimization version of VCP, OptVCP, is an optimization problem such that:

- Let $G$ be an undirected graph.
- We then compute the smallest node set $X$ of $G$ that covers $G$.

Since VCP is NP-complete, OptVCP cannot be solved in polynomial time unless P=NP.

We will show that we can extract the minimum cover sets of $G$ (rather than $G_{\text{modif}}$) from the solution of the minimum memory access problem for $P_G$. For example, the following $\text{SLP}_G$ is one of the minimum solutions:

\[
\begin{align*}
g_{ab} &\leftarrow \rho \circ a \circ b; \\
g_{ac} &\leftarrow \rho \circ a \circ c; \\
g_{cd} &\leftarrow \rho \circ c \circ d; \\
g_{\lambda}\lambda &\leftarrow \rho \circ a \circ \lambda; \\
g_{\mu}\mu &\leftarrow \rho \circ a \circ \mu; \\
\vdots \\
g_{d}\lambda &\leftarrow \rho \circ d \circ \lambda; \\
g_{d}\mu &\leftarrow \rho \circ d \circ \mu; \\
\text{ret}(\cdots)
\end{align*}
\]

From the solution, we extract $a$ and $c$ as cover sets of the original graph $G$; actually, $\{a, c\}$ covers $G$.

Technically, we prove the following two lemmas in the subsequent sections.

**Lemma A1.** Let $Q$ be an $\text{SLP}_G$ where $[P_G] = [Q]$. We can effectively normalize $Q$ to $Q'$ such that:

- $\#_M(Q') \leq \#_M(Q)$.
- Every edge $(a, b)$ of $G$ is represented in $Q'$ as follows:

\[
\Gamma_a \leftarrow \rho \circ a; \\
g_{ab} \leftarrow \Gamma_a \circ b
\]
For a node $a$ of $G$, the local edges $(a, \lambda)$ and $(a, \mu)$ are represented in $Q'$ as follows:

$$
\begin{align*}
&\begin{cases}
\text{if } \Gamma_a \text{ is in } Q' \\
\text{if } \Gamma_a \text{ is not in } Q'
\end{cases}
\end{align*}
\begin{align*}
&g_{a\lambda} \leftarrow \Gamma_a \oplus \lambda; \\
&g_{a\mu} \leftarrow \Gamma_a \oplus \mu.
\end{align*}
$$

Using this lemma, we obtain the following two useful properties.

**Lemma A2.** Let $P$ be a normalized $\text{SLP}_\ominus$ in the meaning of Lemma A1. If the size of the set $\{a \in G : \Gamma_a \text{ is in } Q'\}$ is $k$, then

$$3|E| + 8|N| + k = \#_M(Q').$$

where $E$ is the edge sets of $G$ and $N$ is the node sets of $G$.

**Proof.** For each temporal variables, we need 3-costs for defining $\Gamma_a \leftarrow \rho \oplus a$. This totally require $3k$-costs.

For each local edge of a node $a$,

- if we have $\Gamma_a$, we require 6-costs: $g_{a\lambda} \leftarrow \Gamma_a \oplus \lambda$ and $g_{a\mu} \leftarrow \Gamma_a \oplus \mu$.
- Otherwise, we require 8-costs: $g_{a\lambda} \leftarrow \rho \oplus a \oplus \lambda$ and $g_{a\mu} \leftarrow \rho \oplus a \oplus \mu$.

Totally, for defining all the local edges, we require $6k + 8(|N| - k)$-costs.

For each edge $(a, b)$ of $G$, since we have $\Gamma_a$ or $\Gamma_b$, we require 3-costs $g_{ab} \leftarrow \Gamma_a \oplus b$ or $g_{ab} \leftarrow \Gamma_b \oplus a$.

Entirely, we need the following costs matching with one of the statement:

$$3k + (6k + 8(|N| - k)) + 3|E| = k + 8|N| + 3|E|.$$ 

\hspace{1cm} \Box

Using these lemmas, we obtain the following property.

**Lemma A3.** For a given graph $G$, let $Q$ be the $\#_M$-minimum SLP such that $[Q] = [R_G]$. The following holds on $Q$:

- There is a number $k$ such that $3|E| + 8|N| + k = \#_M(Q)$.
- Furthermore, $k$ is the size of the smallest cover set of $G$.

**Proof.** We normalize $Q$ to $Q'$ using Lemma A1 and then estimate $k$ using Lemma A2. The conditions of Lemma A1 tells we can cover the graph $G$ using $k$ vertices.

Let $X = \{x_1, x_2, \ldots, x_k\}$ be the smallest cover set of $G$. We then can construct an $\text{SLP}_\ominus$ $\mathcal{R}$ such that $\mathcal{R}$ satisfies the conditions of Lemma A1 and $[\mathcal{R}] = [R_G]$ defining variables $\Gamma_{x_1}, \ldots, \Gamma_{x_k}$ as

$$\begin{align*}
\Gamma_{x_1} &\leftarrow \rho \oplus x_1; \\
\Gamma_{x_2} &\leftarrow \rho \oplus x_2; \\
&\vdots \\
\Gamma_{x_k} &\leftarrow \rho \oplus x_k;
\end{align*}
$$

Lemma A2 tells $\#_M(\mathcal{R}) = 3|E| + 8|N| + n$.

From the $\#_M$-minimality of $Q$, $k \leq n$ must hold. On the other hand, from the minimality of $n$, $n \leq k$ must hold; then, we have $k = n$.

\hspace{1cm} \Box

Lemma A3 immediately leads to the intractability of our optimization problem.

Hereafter, we will show Lemma A1.

### A.3 Proof of Lemma A1

**Terminology.** The following is terminology for this section:

- **Temporal variable:** If a variable is not a goal, we call it temporal variable.
- **Goal variable:** If a variable is one of the goal, we call it goal variable.
- **Freely appearing:** If a variable $v$ or a constant $c$ appears in the definition of a temporal variable, we say that $v$ or $c$ freely appears in the program.

**Metavariable Naming Rule.**

- $a, b, \ldots:$ Metavariables for nodes of $G$.
- $t, t_1, t_2, \ldots:$ Metavariables for nodes of $G_{\text{modi}}$.
- $\mathcal{R}, v, \mathcal{R}_1, \mathcal{R}_2, \ldots:$ Metavariables for terms (i.e., constants or variables) of SLP’s.
- $\mathcal{R}_v, \mathcal{R}_2, \ldots:$ Metavariables for a temporal variable of SLP’s.

**Notation.** Let $P$ be an SLP and $v$ be a variable of $P$.

We write $\{v\}$ to denote the value of $v$. To justify this notation, we assume that every SLP of this section is of the SSA (single static assignment) form where each variable is assigned exactly once [6, 87]. We can easily convert a given SLP to an SSA form SLP without changing the semantics and the size. For example,

| non-SSA SLP        | SSA SLP          |
|-------------------|-----------------|
| $v \leftarrow a \oplus b \oplus c;$ | $v_1 \leftarrow a \oplus b \oplus c;$ |
| $v \leftarrow v \oplus d \oplus e;$ | $v_2 \leftarrow v_1 \oplus d \oplus e;$ |
| $v \leftarrow v \oplus f;$       | $v_3 \leftarrow v_2 \oplus f;$       |

**A.3.1 Normalization I.**

Let $v$ be a temporal variable.

If $\{v\} = \{x, y\}$ and $v \leftarrow t_1 \oplus t_2 \oplus \cdots$, we convert the definition to the trivial form as follows:

$$v \leftarrow t_1 \oplus t_2 \oplus \cdots \Rightarrow v \leftarrow x \oplus y;$$

If $\{v\} = \{x, y, z\}$ and $v \leftarrow t_1 \oplus t_2 \oplus t_3 \oplus \cdots$, we convert the definition to the trivial form as follows:

$$v \leftarrow t_1 \oplus t_2 \oplus t_3 \oplus \cdots \Rightarrow v \leftarrow x \oplus y \oplus z;$$

These conversion do not change the semantics of the program and increase the size of the program.

**A.3.2 Normalization II.**

If the program normalized by Normalization-I has a variable $\Gamma_a$ of the form:

$$\Gamma_a \leftarrow \rho \oplus a;$$

using $\Gamma_a$, we rewrite the definitions of $g_{a\lambda}$ and $g_{a\mu}$ as follows:

$$g_{a\lambda} \leftarrow \Gamma_a \oplus \lambda; \quad g_{a\mu} \leftarrow \Gamma_a \oplus \mu;$$

The following holds after these normalization steps.

**Proposition A1.** If $\Gamma_a$ is not in the (normalized) program, for $\lambda_a$ (also $\mu_a$), either one of the following holds:

- $g_{a\lambda}$ is defined as $g_{a\lambda} \leftarrow \rho \oplus a \oplus \lambda$; or
- $\lambda$ freely appears.

**Proof.** We assume that the former does not hold and then consider the following subcases about $g_{a\lambda}$.
The meaning of the term Technical Report, 2021, Yuya Uezato

following replacements rules:

have multiple components.

graph

the definition of a

Proof.

We show (1). The other cases are shown by the same

On the basis of this lemma, we introduce one notation.

Movable occurrence: For a local node \( \lambda \), if its appears in

\( \gamma_{\lambda} \leftarrow v @ \lambda \) where \( \{ v \} = \{ \lambda, a \} \): This contradicts to the as-

\( \gamma_{\lambda} \leftarrow v @ x (x \neq \lambda) \) or \( \gamma_{\lambda} \leftarrow v @ \rho \) where \( \{ v \} = \{ \lambda, \ldots \} \): To
define \( v, \lambda \) must freely appear because any other goal does
not have \( \lambda_a \).

\( \gamma_{\lambda} \leftarrow v @ g_{xy} \) where \( \{ v \} = \{ \lambda, \ldots \} \): To define \( v, \lambda \) must freely
appear.

\( \gamma_{\lambda} \leftarrow v_1 @ v_2 \) where \( \{ v_1 \} = \{ \lambda, \ldots \} \): To define \( v_1, \lambda \) must freely
appear.

On the basis of this lemma, we introduce one notation.

Movable occurrence: For a local node \( \lambda \), if its appears in

\( \gamma_{\lambda} \leftarrow v @ \lambda \) or \( \lambda \) freely appears as \( v \leftarrow \lambda @ \cdots \), we call such
cases of \( \lambda \) movable occurrences.

The meaning of the term movable will be justified below:

A.3.3 Normalization III: Deleting Cancellation.
Let \( P \) be an SLP normalized by the steps Normalization-\( \{ I, II \} \).
From \( P \), we build an SLP \( Q \) where \( Q \) does not have the XOR can-
cellation and \( \#_M(\gamma) \leq \#_M(P) \).

We introduce an auxiliary function unfold to (recursively) unfold
the definition of a temporal variable:

\[
\begin{align*}
\text{unfold}(x) &= \{ x \} & \text{if } x \text{ is a constant} \\
\text{unfold}(g_{xy}) &= \{ g_{xy} \} \\
\text{unfold}(v) &= \bigoplus_{i=1}^n \text{unfold}(t_i) & \text{if } v \leftarrow t_1 \oplus t_2 \oplus \cdots \oplus t_n
\end{align*}
\]

It should be noted that we do not unfold the definition of a goal for
our construction.

Using constants and goals that freely appear in \( P \), we define a
graph \( \mathcal{D} \) as follows:

(1) First, we define a graph \( \mathcal{D} \) as a complete graph whose nodes
are constants that freely appear in \( P \).

(2) Next, for each freely appearing goal \( g_{xy} \) in \( P \), we add the
dge \( (x, y) \) to \( \mathcal{D} \).

We note the obtained graph \( \mathcal{D} \) may not be connected; i.e., it may
have multiple components.

Notations.

\( \mathcal{D}_a \): We call the unique component of \( \mathcal{D} \) that contains freely
appearing constants \( \mathcal{D}_a \).

\( \Rightarrow \): We write \( a \Rightarrow b \) to the path between \( a \) and \( b \) of \( \mathcal{D} \).

This graph \( \mathcal{D} \) has the following useful properties.

Proposition A2. Let \( v \) be a temporal variable of \( P \).

(1) If \( \{ v \} = \{ x, y \} \), then \( x \Rightarrow y \).

(2) If \( \{ v \} = \{ x_1, x_2, x_3, x_4 \} \), then this consists of two paths \( n_1 \Rightarrow n_2 \) and \( n_3 \Rightarrow n_4 \) where \( \{ x_1, x_2, x_3, x_4 \} = \{ n_1, n_2, n_3, n_4 \} \).

(3) If \( \{ v \} = \{ \rho, x, y, z \} \), then this consists of one path \( n_1 \Rightarrow n_2 \) and one point \( n_3 \in \mathcal{D}_a \), where \( \{ x, y, z \} = \{ n_1, n_2, n_3 \} \).

Proof. We show (1). The other cases are shown by the same
argument.

Let \( X = \text{unfold}(v) \). We reduce \( X \) repeatedly applying one of the
following replacements rules:

(1) Choose \( x \in \mathcal{D}_a \) and \( y \in \mathcal{D}_a \) from \( X \), remove them from \( X \),
then update \( X \leftarrow X \oplus \{ (x, y) \} \). Remark \( (x, y) \in \mathcal{D} \).

(2) Choose \( x \in \mathcal{D}_a \) and \( (x, y) \in \mathcal{D} \) from \( X \), remove them from \( X \),
then update \( X \leftarrow X \oplus \{ (y, x) \} \). Remark \( (y, x) \in \mathcal{D} \).

(3) Choose \( (x, y) \in \mathcal{D} \) from \( X \) and \( (x, z) \) from \( X \), remove them from \( X \),
then update \( X \leftarrow X \oplus \{ (y, z) \} \). Remark \( (y, z) \in \mathcal{D} \).

When we cannot apply any rule, then \( X \leftarrow (a, b) \); thus, \( a \Rightarrow b \).

Proposition A3. If a goal \( g_{xy} \) is defined by two temporal variables \( v_1, v_2 \) (i.e., \( g_{xy} \leftarrow v_1 @ v_2 \)), then \( x \Rightarrow y \).

Proof. By applying the same argument of the above proposition
for \( X = \text{unfold}(v_1) \oplus \text{unfold}(v_2) \), we have \( x \Rightarrow y \).

Hereafter, we build an SLP \( Q \) that does not use the XOR-
cancellation.

First, we copy \( P \) to be normalized to \( P' \).

Removing nodes that freely appear in \( P' \).

Let \( a_1, a_2, \ldots \) are (non-local) nodes that freely appear in \( P' \).

- If \( \Gamma_{a_i} \) is in \( P' \), we move it to \( Q \).

- Moving means that we remove \( \Gamma_{a_i} \leftarrow \rho @ a_i \) from \( P' \)
and then add it to \( Q \).

- Otherwise, by Proposition A1, we have movable occurrences
of \( \lambda_{a_i} \) and \( \mu_{a_i} \) in \( P' \).

We remove all the free occurrences of \( \Gamma_{a_i} \) and all the movable
occurrences of \( \lambda_{a_i} \) and \( \mu_{a_i} \) from \( P' \). We then add \( \Gamma_{a_i} \leftarrow \rho @ a_i \)
to \( Q \).

Removing goal variables that freely appear in \( P' \).

Let \( g_1, g_2, \ldots \) are goal variables that freely appear in \( P' \).

- Let \( \Gamma_{g_1} \) is \( g_{ab} \).

- Let \( C \) be the component of \( \mathcal{D} \) that the edge \( (a, b) \) belongs to.

- If there is \( x \in C \) such that \( \Gamma_{x} \) is not in \( Q \), by Proposition A1,
we have movable occurrences of \( \lambda_{x} \) and \( \mu_{x} \) in \( P' \).

We remove all the free occurrences of \( g_i \) and all the movable
occurrences of \( \lambda_{x} \) and \( \mu_{x} \).

We then add \( \Gamma_{g_i} \leftarrow \rho @ c \) to \( Q \).

After the above modification to \( Q \), the following property holds.

Proposition A4.

\( \#_M(P') + \#_M(Q) \leq \#_M(P) \).

For each non-local node \( a \in \mathcal{D} \), we have \( \Gamma_a \) in \( Q \).

For each path \( a \Rightarrow b \), we have \( \Gamma_a \) or \( \Gamma_b \) in \( Q \).

For each path \( a \Rightarrow x \) where \( x \) is local, we have \( \Gamma_a \) in \( Q \).

We do not need \( x \in \{ \lambda_{a_i}, \mu_{a_i} \} \).

Removing temporal variables of specific patterns in \( P' \).

Let \( v \) be a temporal variable in \( P \).

Path: If \( \{ v \} = \{ \lambda_{a_i}, \rho \} \) and \( \Gamma_{a_i} \) is not in \( Q \), there are movable
occurrences of \( \lambda_{a_i} \) and \( \mu_{a_i} \) in \( P' \).

(1) First, we remove all the such occurrences.

(2) Next, we remove the left-side occurrence of \( v \) of the defi-
nition of \( v \); i.e.,

\[
\begin{align*}
v & \cdots \leftarrow \cdots \\
\Rightarrow & \cdots
\end{align*}
\]

- Although such the blank is not permitted in our for-
marialization of SLP, we temporarily allow it only on \( P' \)
because we use \( P' \) to build \( Q \) and do not execute \( P' \).
• By these removal, we decrement the size of $P'$ more than three. Recall that the size of $\mathcal{L}_M$ equals to the number of the total occurrences of constants and variables.

(3) Finally, we add $\Gamma_a \leftarrow \rho \odot a$ to $Q$. It increments the size of $Q$ by three.

Totally, $\#_M(P') + \#_M(Q) \leq \#_M(P)$ still holds.

**Pat2:** If $\{(\lambda_a, \mu_a)\}$ and $\Gamma_a$ is not in $Q$, by the same argument of Pat1, we remove the left side occurrence of $\lambda$ and all the movable occurrences of $\lambda_a$ and $\mu_a$; then, we add $\Gamma_a \leftarrow \rho \odot a$ to $Q$.

**Pat3:** If $\{(\lambda_a, b)\}$ and $\Gamma_a$ is not in $Q$, we remove the definition of $v$ from $P'$ and then add $\Gamma_a \leftarrow \rho \odot a$ to $Q$.

**Pat4:** If $\{(a, b)\}$ and $\Gamma_a$ (or $\Gamma_b$) is not in $Q$, then we remove the left side occurrence of $a$ and all the movable occurrences of $\lambda_a$ and $\mu_a$ (or $\lambda_b$ and $\mu_b$) from $P'$.

If $\Gamma_a$ is not in $Q$, we add $\Gamma_a \leftarrow \rho \odot a$ to $Q$. Otherwise, if $\Gamma_b$ is not in $Q$, we add $\Gamma_b \leftarrow \rho \odot a$ to $Q$. Consequently, we have $\Gamma_a$ and $\Gamma_b$ in $Q$.

**Pat5:** If $\{(a, b, c, d)\}$, by Proposition A2 and A4 there are at most two nodes $x, y \in \{a, b, c, d\}$ such that $\Gamma_x$ and $\Gamma_y$ are not in $Q$.

We remove the left side occurrence of $v$ and all the movable occurrences of $\lambda_a$ and $\mu_a$. We then $\Gamma_a \leftarrow \rho \odot x$ to $Q$. Consequently, there is at most one node $y \in \{a, b, c, d\}$ such that $\Gamma_y$ is not in $Q$.

**Pat6:** If $\{(a, \lambda, b, \lambda_b)\}$, by Proposition A2 and A4, $\Gamma_a$ or $\Gamma_b$ may not exist in $Q$.

If $\Gamma_a$ is not in $Q$, we remove the left side occurrence of $a$ and all the movable occurrences of $\lambda_a$ and $\mu_a$. We then $\Gamma_a \leftarrow \rho \odot a$ to $Q$.

Otherwise, we do the same for $\Gamma_b$.

Consequently, we have $\Gamma_a$ and $\Gamma_b$ in $Q$.

**Pat7:** If $\{(a, \lambda, b, c)\}$, by Proposition A2 and A4, we may do not have $\Gamma_a$ for at most one of $\{a, b, c\}$.

We remove $v, \lambda_a$, and $\mu_a$ and then $\Gamma_a \leftarrow \rho \odot x$.

Consequently, we have $\Gamma_a, \Gamma_b$, and $\Gamma_c$ in $Q$.

**Pat8:** If $\{(\rho, a, \lambda_a, b)\}$, By Proposition A2 and A4, there is at most one node $x \in \{a, b\}$ such that $\Gamma_x$ is not in $Q$.

For such $x$, we remove the left side occurrence of $v$ and all the movable occurrences of $\lambda_a$ and $\mu_a$. We then $\Gamma_x \leftarrow \rho \odot x$ to $Q$.

Consequently, we have $\Gamma_a, \Gamma_b$, and $\Gamma_c$ in $Q$.

After the above construction, we still have:

\[
\#_M(P') + \#_M(Q) \leq \#_M(P).
\]

Transferring the goal of $P$ without changing its size.

Now we define all the goals without the XOR cancellation in $Q$. First, we consider the goals of the form $g_{ab}$.

If $g_{ab} \leftarrow \rho \odot a @ \lambda$: We move it from $P'$ to $Q$.

If $g_{ab} \leftarrow \Gamma_a @ \Gamma_b$: Hereafter, for each subcase, we show that we have $\Gamma_a$ in $Q$. It suffices for our construction because

(1) We delete the definition of $g_{ab}$; it decrements the size of $P'$ by three.

(2) We then add $g_{ab} \leftarrow \Gamma_a @ \lambda$ to $Q$; it increments the size of $Q$ by three.

(3) Totally, $\#_M(P') + \#_M(Q) \leq \#_M(P)$ still holds.

---

**Case $g_{ab} \leftarrow \rho \odot a (\lambda \text{ has been removed to add } \Gamma_a \text{ to } Q):** Clearly we have $\Gamma_a$ in $Q$.

**Case $g_{ab} \leftarrow v @ a (v) = \{\lambda, \rho\}:** By Pat1, we have $\Gamma_a$ in $Q$.

**Case $g_{ab} \leftarrow \Gamma_a @ \lambda$:** Clearly we have $\Gamma_a$ in $Q$.

**Case $g_{ab} \leftarrow v @ \rho (v) = \{\lambda_a\}:** By Proposition A2 and A4, we have $\Gamma_a$ in $Q$.

**Case $g_{ab} \leftarrow v @ b (v) = \{\lambda_a, b\}:** By Pat2, we have $\Gamma_a$ in $Q$.

**Case $g_{ab} \leftarrow v @ g_{ab} (v) = \{\lambda_a, b\}:** By Prop, we have $\Gamma_a$ in $Q$.

**Case $g_{ab} \leftarrow v @ g_{ab} (v) = \{\lambda_a, b\}:** By Pat3, we have $\Gamma_a$ in $Q$.

**Case $g_{ab} \leftarrow v @ g_{ab} (v) = \{\lambda_a, b\}:** By Pat6, we have $\Gamma_a$ in $Q$.

**Case $g_{ab} \leftarrow v @ g_{ab} (v) = \{\lambda_a, b\}:** By Pat7, we have $\Gamma_a$ in $Q$.

**Case $g_{ab} \leftarrow v_1 @ v_2$ and $v_1, v_2$ are variables:** By Proposition A3 and A4, we have $\Gamma_a$ in $Q$.

Next, we consider the goals of the form $g_{ab}$ where $a$ and $b$ are (non-local) nodes.

If $g_{ab} \leftarrow \rho \odot a @ b$: We copy it to $Q$.

If $g_{ab} \leftarrow \Gamma_a @ \Gamma_b$: Hereafter, for each subcase, we show that we have $\Gamma_a$ in $Q$. As the same as the construction of $g_{ab}$, it suffices for our construction.

If $g_{ab} \leftarrow \rho \odot a @ \Gamma_b$: Clearly, we have $\Gamma_a$ in $Q$.

If $g_{ab} \leftarrow v @ \rho (v) = \{a, b\}: $ By Proposition A2 and A4, we have $\Gamma_a$ or $\Gamma_b$ in $Q$.

If $g_{ab} \leftarrow v @ \lambda_a (v) = \{a, b, \lambda_a\}: $ By Proposition A2 and A4, we have $\Gamma_a$ or $\Gamma_b$ in $Q$.

If $g_{ab} \leftarrow v @ c (v) = \{a, b, c\}: $ By Proposition A2 and A4, we have $\Gamma_a$ or $\Gamma_b$ in $Q$.

If $g_{ab} \leftarrow v @ g_{ab} (v) = \{a, b, \lambda_a\}: $ By Pat3, we have $\Gamma_a$ in $Q$.

If $g_{ab} \leftarrow v @ g_{ab} (v) = \{a, b, c\}: $ By Prop, we have $\Gamma_a$ and $\Gamma_b$ in $Q$.

If $g_{ab} \leftarrow v @ g_{ab} (v) = \{a, b, c\}: $ By Pat4, we have $\Gamma_a$ and $\Gamma_b$ in $Q$.

If $g_{ab} \leftarrow v @ g_{ab} (v) = \{a, b, c\}: $ By Pat7, we have $\Gamma_a$ or $\Gamma_b$ in $Q$.

If $g_{ab} \leftarrow v @ g_{ab} (v) = \{a, b, c\}: $ By Pat5, we have $\Gamma_a$ or $\Gamma_b$ in $Q$.

If $g_{ab} \leftarrow v_1 @ v_2$: By Proposition A3 and A4, we have $\Gamma_a$ or $\Gamma_b$ in $Q$.

By our construction, the following holds in $Q$.

**Lemma A4.**

- $[Q] = [P]$.
- $\#_M(Q) \leq \#_M(P)$.
- Every temporal variable of $Q$ is the form of $\Gamma_a \leftarrow \rho \odot a$ where $a$ is a node of $G$.
- The definition of each goal $g$ of $Q$ forms one of the following:
  - $g \leftarrow \Gamma_a @ x$:
  - $g \leftarrow \rho \odot x @ y$.

A.3.4 Normalization IV: Finalization.

Finally, we prove Lemma A1.
The above lemma Lemma A4 does not ensure that every edge 
\((a, b)\) is represented by \(g_{ab} \leftarrow \Gamma_a \oplus b\) or \(g_{ab} \leftarrow \Gamma_b \oplus a\) because \(Q\) may not have both of \(\Gamma_a\) and \(\Gamma_b\).

In the case, \(Q\) has the following definitions:

\[
\begin{align*}
g_{ab} &\leftarrow \rho \oplus a \oplus b; \\
g_{a\lambda} &\leftarrow \rho \oplus a \oplus \lambda; \\
g_{a\mu} &\leftarrow \rho \oplus a \oplus \mu;
\end{align*}
\]

We change this part as follows without changing \(#_M(Q)\):

\[
\begin{align*}
\Gamma_a &\leftarrow \rho \oplus a; \\
g_{ab} &\leftarrow \Gamma_a \oplus b; \\
g_{a\lambda} &\leftarrow \Gamma_a \oplus \lambda; \\
g_{a\mu} &\leftarrow \Gamma_a \oplus \mu;
\end{align*}
\]

Repeatedly applying this modification, we transform \(Q\) to \(Q'\) that satisfies all the conditions of Lemma A1.