REDUCED-COMPLEXITY CYCLOTOMIC FFT AND ITS APPLICATION TO REED-SOLOMON DECODING

Ning Chen and Zhiyuan Yan

Department of Electrical and Computer Engineering, Lehigh University, PA 18015, USA
E-mail: {nic6, yan}@lehigh.edu

ABSTRACT

Cyclotomic fast Fourier transform (CFFT) was recently proposed and shown to be efficient for lengths up to 511. In this paper, we propose a novel algorithm to reduce the additive complexity of CFFT. When used in transform-domain Reed-Solomon decoders, our improved CFFT reduces the complexity of the transform portion by up to 72%.

Index Terms— Discrete Fourier transforms, Galois fields, Reed-Solomon codes, Complexity theory, Optimization

1. INTRODUCTION

Due to the widespread applications of Reed-Solomon (RS) codes in various digital communication and storage systems, efficient decoding of RS codes has been an important research topic. All syndrome-based decoding methods for RS codes involve discrete Fourier transform (DFT) over finite fields [1]: partial DFT is used in syndrome evaluation, and transform-domain decoders use inverse DFT to recover transmitted codewords. Thus, efficient FFT algorithms can be used to reduce the complexity of RS decoders. Using the prime-factor FFT algorithm in [2], Truong *et al.* proposed an inverse-free transform-domain RS decoder whose complexity is substantially lower than time-domain decoders [3].

Cyclotomic FFT (CFFT) was recently proposed [4, 5] and shown to be efficient for lengths up to 511 [4]. Furthermore, it was shown that inverse cyclotomic FFT (ICFFT) was efficient for syndrome evaluation of length-255 RS codes [6]. Transpose cyclotomic FFT (TCFFT) was also proposed for syndrome evaluation [5]. To avoid confusion, we call the original form of CFFT in [4] direct cyclotomic FFT (DCFFT).

Though cyclotomic FFT is attractive for its low multiplicative complexity, its additive complexity is very high if implemented directly. In [4], a heuristic algorithm based on decoding in the binary erasure channel [7] was used to reduce the additive complexity.

Realizing that it is actually an NP-complete problem to minimize the additive complexity of CFFT, we propose a novel common subexpression elimination (CSE) algorithm that significantly reduces the additive complexity of CFFT. Compared with direct computation, our CSE algorithm reduces the additive complexity of full DCFFT, TCFFT, and ICFFT by more than 80%, and the additive complexity of our improved DCFFT is 12% smaller than that in [4], the best results of DCFFT known to us. For partial CFFT, our improved TCFFT achieve 20% saving on additive complexity over that in [6]. Using full and partial CFFTs instead of prime-factor FFT algorithm in [3], we reduce the complexity of the DFT and IDFT portions of the transform-domain decoder in [3] by up to 72% for RS codes of lengths up to 1023.

The rest of the paper is organized as follows. In Section 2, we briefly review various cyclotomic FFT algorithms. Section 3 presents our CSE algorithm. Reduced-complexity full and partial CFFTs are discussed in Sections 4.1–4.3. Finally, we apply the improved CFFT to transform-domain RS decoders in Section 4.4.

2. CYCLOTOMIC FFT

For a primitive element $\alpha \in \mathrm{GF}(2^m)$, the transform from a polynomial $f(x) = \sum_{i=0}^{n-1} f_i x^i \in \mathrm{GF}(2^m)[x]$ to $\boldsymbol{F} = (f(\alpha^0), f(\alpha^1), \ldots, f(\alpha^{n-1}))^T$ is referred to as a DFT of $\boldsymbol{f} = (f_0, f_1, \ldots, f_{n-1})^T$. Representing the original polynomial $f(x)$ as a sum of linearized polynomials by cyclotomic decomposition [4, 5], cyclotomic FFT $\boldsymbol{F} = \boldsymbol{A}\boldsymbol{L}\boldsymbol{f}' = \boldsymbol{A}\boldsymbol{L}\boldsymbol{\Pi}\boldsymbol{f}$, where $\boldsymbol{A}$ is an $n \times n$ binary matrix, $\boldsymbol{f}' = (\boldsymbol{f}_0'^T, \boldsymbol{f}_1'^T, \cdots, \boldsymbol{f}_{l-1}'^T)$ is a permutation of the input vector $\boldsymbol{f}$, $\boldsymbol{\Pi}$ is a permutation matrix, and $\boldsymbol{L} = \mathrm{diag}(\boldsymbol{L}_0, \boldsymbol{L}_1, \cdots, \boldsymbol{L}_{l-1})$ is a block diagonal matrix with square matrices $\boldsymbol{L}_i$'s on the diagonal. Mapped to normal basis, $\boldsymbol{L}_i$ becomes a quadratic generalized Vandermonde matrix. Thus, the product of $\boldsymbol{L}_i$ and $\boldsymbol{f}_i'$ can be computed as a cyclic convolution over finite fields, for which fast algorithms are available. These fast algorithms can be written in matrix form as $\boldsymbol{L}_i\boldsymbol{f}_i' = \boldsymbol{Q}_i(\boldsymbol{c}_i \cdot \boldsymbol{P}_i\boldsymbol{f}_i')$, where $\boldsymbol{Q}_i$ and $\boldsymbol{P}_i$ are both binary matrices, $\boldsymbol{c}_i$ is a constant vector, and $\cdot$ stands for pointwise multiplications. Hence DCFFT is given by $\boldsymbol{F} = \boldsymbol{A}\boldsymbol{Q}(\boldsymbol{c} \cdot \boldsymbol{P}\boldsymbol{f}')$, where $\boldsymbol{Q} = \mathrm{diag}(\boldsymbol{Q}_0, \boldsymbol{Q}_1, \cdots, \boldsymbol{Q}_{l-1})$ is a block diagonal matrix with $\boldsymbol{Q}_i$'s on the diagonal, $\boldsymbol{P} = \mathrm{diag}(\boldsymbol{P}_0, \cdots, \boldsymbol{P}_{l-1})$ is a block diagonal matrix with $\boldsymbol{P}_i$'s on the diagonal, and $\boldsymbol{c} = (\boldsymbol{c}_0^T, \boldsymbol{c}_1^T, \cdots, \boldsymbol{c}_{l-1}^T)^T$.

Fedorenko [5] constructed an equivalent transform with symmetric transform matrix, which leads to TCFFT as $\boldsymbol{F}' =$

$\boldsymbol{L}^T \boldsymbol{A}'^T \boldsymbol{f}' = \boldsymbol{Q}(\boldsymbol{c} \cdot \boldsymbol{P}\boldsymbol{A}'^T \boldsymbol{f}')$, where $\boldsymbol{F}' = \boldsymbol{\Pi}\boldsymbol{F}$ and $\boldsymbol{A}' = \boldsymbol{\Pi}\boldsymbol{A}$. In [6], the complexity of partial DFT was reduced by using ICFFT as $\boldsymbol{F}'' = \boldsymbol{L}^{-1}\boldsymbol{A}^{-1}\boldsymbol{f} = \boldsymbol{Q}(\boldsymbol{c} \cdot \boldsymbol{P}\boldsymbol{A}^{-1}\boldsymbol{f})$, where $\boldsymbol{F}''$ is also a permutation of $\boldsymbol{F}$. Note that $\boldsymbol{L} = \boldsymbol{L}^T = \boldsymbol{L}^{-1}$.

For CFFT, the number of multiplications is determined by the number of non-one elements in $\boldsymbol{c}$ while the number of additions is determined by the two matrix-vector multiplications in which both matrices are binary. For example, in DCFFT, the matrices are $\boldsymbol{AQ}$ and $\boldsymbol{P}$. Due to the size of $\boldsymbol{AQ}$, direct computation of the matrix-vector product will result in high additive complexity. Next, we will propose an algorithm that significantly reduces the additive complexity of CFFT.

3. CSE ALGORITHM

3.1. Problem Analysis

In this section, we first model the minimization of the additive complexity of CFFT as a matrix transformation, and then provide a simple heuristic description of our CSE algorithm.

The optimization problem can be modeled as a linear transform $\boldsymbol{Y} = \boldsymbol{M}\boldsymbol{X}$, where $\boldsymbol{Y}$ and $\boldsymbol{X}$ are n- and n'-dimensional column vectors and $\boldsymbol{M}$ is an $n \times n'$ matrix containing only $1, -1$, and 0. $\boldsymbol{X}$ represents the variable input while $\boldsymbol{M}$ indicates the set of constants. Clearly, such a transform requires only additions and subtractions. It was shown that to minimize the number of additions and subtractions is an NP-complete problem [8].

We now propose a CSE algorithm with polynomial complexity that significantly reduces the additive complexity of CFFT. Although our CSE algorithm does not guarantee to minimize the additive complexity, it may in some cases, especially when the size of the problem is small. Our algorithm exploits two kinds of savings: differential saving and recurrence saving, as defined below.

One can of course reduce the additive complexity by first identifying recurring patterns, which are combinations of non-zero positions, and then calculating them only once. Such a pattern can be defined as a vector. We refer to the number of occurrences of a pattern in $\boldsymbol{M}$ as *pattern frequency*, and define *recurrence saving* of each pattern as its pattern frequency minus 1. After identifying a pattern, instead of simply eliminating it, we replace the pattern with a new element which stands for the result of the identified pattern. This process can be described in matrix decomposition form: $\boldsymbol{M} = \boldsymbol{M}_R \boldsymbol{M}_{j-1} \dots \boldsymbol{M}_1 \boldsymbol{M}_0$, where $\boldsymbol{M}_i = [\boldsymbol{I} \mid \boldsymbol{G}_i]^T$ and $\boldsymbol{G}_i$ is the identified pattern vector. Thus the transform can be computed in a sequential fashion: first assign $\boldsymbol{X}_0 = \boldsymbol{X}$, then compute $\boldsymbol{X}_{i+1} = \boldsymbol{M}_i \boldsymbol{X}_i$ for $i = 0, 1, \cdots, j-1$, and finally compute $\boldsymbol{Y} = \boldsymbol{M}_R \boldsymbol{X}_j$.

Since multi-bit patterns can be expressed recursively as two-bit patterns whose elements are previously identified patterns, our CSE algorithm looks for only two-bit patterns and keeps track of all previously identified patterns in a matrix to reduce computational complexity.

In CFFT, the elements of $\boldsymbol{X}$ are over characteristic-2 fields. For these fields, 1 and -1 are identical and hence additions are the same as subtractions. It also has a special property: if two rows of $\boldsymbol{M}$ both have 1's on the same columns, their sum will cancel out on those columns. The additional property implies that we can take advantage of *differential saving*. If the difference (or sum) of two rows r_0 and r_1 contains fewer elements than one of the two rows, say r_0, we can obtain r_0 by adding the difference (sum) to r_1, thereby reducing the number of additions. However, these two rows must be computed in a strict order. We call r_1, the row which will be computed first, the parent row and r_0 the child row. We use a new element to represent the result of the parent row. After we replace the child row with the difference row, we append the new element to it. Let the numbers of non-zero elements for an ordered pair of rows (r_p, r_c), in which r_p is the parent row and r_c is the child row, be s_p and s_c, respectively, and let the number of non-zero elements for their difference be s_d. The differential saving for the ordered pair is given by $s_c - s_d - 1$. Since we are only concerned about positive savings, we use $[s_c - s_d - 1]^+ \stackrel{\text{def}}{=} \max\{0, s_c - s_d - 1\}$ in our algorithms.

Note that such an operation introduces no extra computation step to the final result except for imposing a strict order between the rows of $\boldsymbol{M}$. After such a transform, the linear transform becomes $\boldsymbol{Y} = \boldsymbol{M}'\boldsymbol{X}'$, where $\boldsymbol{X}' = (\boldsymbol{X}^T, \boldsymbol{Y}'^T)^T$ and $\boldsymbol{Y}'$ includes all the parent rows.

Algorithm 1: Common Subexpression Elimination

1. Identify the pairs of rows with the greatest differential saving, select one pair out of them randomly, replace the child row with the difference between the two rows, and append the element representing the parent row to it. Repeat until there is no differential saving.
2. Identify the two-bit patterns with the greatest recurrence saving, select one out of them randomly, replace all occurrences of the selected pattern with a new element. Go to Step 1 until there is no recurrence saving.

3.2. Randomized Greedy Algorithm

Our CSE algorithm, shown in Algorithm 1, has two steps: Steps 1 and 2 are referred to as the differential saving and recurrence saving steps respectively. In both steps, we use the greedy strategy which choose the transform with the greatest saving. If there are multiple choices, we randomly choose one among them. Note that our choice is merely locally optimal, and does not guarantee the optimal solution. Our goal is to reduce the number of additions as much as possible.

Since differential saving reduces the number of non-zero elements without introducing new patterns which still require additions, differential saving is given higher priority than recurrence saving in our algorithm. Since our algorithm is a randomized algorithm, the result of each run may vary. However, simulation results show that the variance between different runs is quite small even for large problems.

3.3. Cycle Detection

Let (r_p, r_c) be an ordered pair of rows in which r_p is the parent row and r_c is the child row. If the pair is selected, it imposes a restriction that the result of r_p must be computed before that of r_c. All the selected pairs form a directed graph, where the vertices are the row numbers in the pairs and the edges are from the parent rows to the child rows in all pairs. The restrictions imposed by all selected pairs constitute the requirement that the graph must be cycleless. Our differential saving step must maintain this graph and use it to avoid selecting ordered pairs that will result in a cycle in the graph.

Our CSE algorithm uses the recursive procedure in Algorithm 2 to perform cycle detection. If an ordered pair is cycle-introducing, it will not be considered in the random selection of the differential saving step. If all ordered pairs are cycle-introducing, the differential saving step will finish.

Algorithm 2: CycleDetect(r_{pc}, r_{cc})

input : A pair of rows (r_{pc}, r_{cc})
output: If it leads to a cycle, return true; otherwise, return false

```
foreach established pair (r_pi, r_ci) do
    if r_pi = r_cc then
        if r_pc = r_ci then
            return true
        end
        else if CycleDetect(r_pc, r_ci) = true then
            return true
        end
    end
end
return false
```

3.4. Complexity Reduction Improvements

When the size of $\boldsymbol{M}$ is large, the computational complexity of Algorithm 1 could be prohibitive. We propose several improvements to further reduce the runtime of our CSE algorithm.

In Algorithm 1, we restart the differential saving step after each recurrence saving step. But the possibility that new differential savings emerge after we eliminate a pattern for recurrence saving is quite small. In order to reduce the complexity of our CSE algorithm, we do not revisit the differential saving step once the recurrence saving step has started, essentially decoupling the two steps. This not only reduces the runtime by reducing the number of the differential saving steps, but also enables us to further accelerate both steps by space-time trade-off, which will be discussed below.

Now that the differential saving step is stand-alone, it is necessary to avoid repeated exhaustive search. There are only n rows in $\boldsymbol{M}$, so all possible differential saving can be put in an $n \times n$ array $\boldsymbol{D}$, where D_{ij} stands for the differential saving of the ordered pair of rows (r_i, r_j). Such an array can be built with an initial exhaustive search. After it is constructed, at most $2(n-1)$ elements of the array need to be updated after each differential saving step. Whenever one pair of rows is detected to be cycle-introducing, its differential saving will be set to -1 and hence it is excluded from later consideration. Thus the number of possible pairs will decrease continuously, and the search will be increasingly faster.

A similar idea can be used to reduce the runtime of the recurrence saving step. Since elimination of one pattern will only change a small portion of the pattern frequencies, to expedite searches, we store the pattern frequencies and update them after each elimination step.

Algorithm 3: CSE with Runtime Reduction

1. Initialize the differential saving array $\boldsymbol{D}$.
2. Find the cycleless pairs of rows with the greatest differential saving in $\boldsymbol{D}$, randomly choose one, eliminate it, and update $\boldsymbol{D}$. Repeat until there is no positive element in $\boldsymbol{D}$.
3. Initialize the recurrence saving array $\boldsymbol{R}$.
4. Find the patterns with the greatest number in $\boldsymbol{R}$, randomly choose one, eliminate it, and update $\boldsymbol{R}$. Repeat until all elements in $\boldsymbol{R}$ are zero.

Because not all patterns exist and the number of possible patterns will decrease progressively, it will save much space if we keep track of only the possible patterns. But it will involve full searches to update the frequencies and to remove patterns which disappear after each elimination. The complexity of such full searches will increase rapidly when the size $\boldsymbol{M}$ is large. We keep all pattern frequencies in a two-dimensional array $\boldsymbol{R}$, where $\boldsymbol{R}_{ij}$ is the recurrence saving of the two-bit pattern which has non-zero bits on positions i and $i+j+1$. Suppose after the differential saving steps are over, and $\boldsymbol{M}'$ has $\bar{n}$ columns. Initially, $\boldsymbol{R}$ is an $(\bar{n}-1)\times(\bar{n}-1)$ array, where $\boldsymbol{R}_{ij}$ is the recurrence saving of the two-bit pattern which has non-zero bits on positions i and $i+j+1$ for $0 \le i \le \bar{n}-2$ and $0 \le j \le \bar{n}-i-2$. Hence frequency update can be done without search and it is not necessary to remove frequencies. When a new pattern is identified, it forms a new possible pat-

tern by combination with every previously identified patterns. So a new pattern frequency is appended to every row of $\boldsymbol{R}$, and a new row with only one pattern frequency will be appended to the bottom of $\boldsymbol{R}$.

Our CSE algorithm incorporating the above improvements is shown in Algorithm 3. Note our results show that the decoupling of the two steps result in only negligible performance loss. The speed advantage of Algorithm 3 over Algorithm 1 enable us to run Algorithm 3 many more times, enhancing the possibility of obtaining a better result than Algorithm 1 within the same amount of time.

3.5. Modified Differential Saving Update Scheme

During the differential saving step, our CSE algorithm only keeps one copy of each row. Actually one row can have multiple different decompositions, based on differential savings with different rows. To exploit the best differential saving for each row, a modified differential saving update scheme is developed.

Say the pair (r_p, r_c) is selected for differential saving elimination. After elimination, the child row r_c is updated to r'_c. For an arbitrary row r_i, the differential saving of (r_c, r_i) can be higher than (r'_c, r_i). In the modified update scheme, such cases are better handled by not modifying the differential saving D_{ci} and keeping a copy of r_c. Since for different r_i we may need different copies of r_c, an array $\boldsymbol{K}$ whose element K_{ij} keeps a copy of r_j corresponding to D_{ij} is necessary.

If (r'_c, r_i) has bigger saving than D_{ci}, we simply update D_{ci} as before and K_{ij} becomes a empty row since no copy is necessary. If the new saving is equal to D_{ci}, it is randomly chosen which copy to use.

Now we provide an example of Algorithm 3. Suppose we have $\boldsymbol{M} = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$, and the differential saving array $\boldsymbol{D}$ is initialized as $\begin{pmatrix} -1 & 3 & 1 & 0 \\ 2 & -1 & 2 & 0 \\ 1 & 3 & -1 & 0 \\ 0 & 2 & 0 & -1 \end{pmatrix}$, and $\boldsymbol{K}$ is empty. Choosing (r_0, r_1), we have $\boldsymbol{M}' = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$, where the new column represents the result of the first row, and the differential saving array is updated as $\begin{pmatrix} -1 & -1 & 1 & 0 \\ -1 & -1 & 2 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$, and K_{12} becomes $(1, 1, 1, 1, 1, 1)$ to keep the previous copy of r_1. Since (r_0, r_1) is selected, (r_1, r_0) is cycle-introducing and hence its saving is simply set to -1. Choosing (r_1, r_2), $\boldsymbol{M}'$ becomes $\begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$. Note that (r_2, r_0) is cycle introducing so there is no positive differential saving left and $\boldsymbol{K}$ becomes empty, and we enter the recurrence saving step. The recurrence saving array $\boldsymbol{R}$ for $\boldsymbol{M}'$ is initialized as $\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \\ 1 & 0 & 0 & 0 & & \\ 0 & 0 & 0 & & & \\ 0 & 0 & & & & \\ 0 & & & & & \end{pmatrix}$. So the only choice is $(2, 3)$, which corresponds to $X_2 + X_3$. Hence $\boldsymbol{G}_0$ is $(0, 0, 1, 1, 0, 0, 0, 0)^T$ and $\boldsymbol{M}_R$ is $\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$, and the recurrence saving array becomes all zeros. $\boldsymbol{M}_R$ needs 5 additions. The identified pattern $X_2 + X_3$ also needs one addition. So $\boldsymbol{Y} = \boldsymbol{M}\boldsymbol{X}$ can be calculated by 6 additions, whereas a straightforward implementation of $\boldsymbol{Y} = \boldsymbol{M}\boldsymbol{X}$ requires 12 additions. It is easy to verify that the minimum number of additions needed is 6. Hence, our CSE algorithm minimizes the number of additions in this case. Note that if we only use recurrence savings, the result will be 7 additions.

3.6. Computational and Storage Complexity

We can show that the time complexity of Algorithm 3 for an $n \times n'$ matrix $\boldsymbol{M}$ is $\frac{1}{4}n^3n'^2 + O(n^3n') + O(n^2n'^2)$ and its storage complexity is $\frac{1}{8}n^2n'^2 + O(n^2n') + O(nn'^2)$. Thus, our CSE algorithm has polynomial complexity. Due to limited space, the details of the analysis is omitted.

Note that the complexity of our CSE algorithm depends on only $\boldsymbol{M}$. Once a CFFT is determined, it can be used for any input vector. Hence our CSE algorithm is just precomputation and its complexity does not affect that of the CFFT.

4. APPLICATION IN CFFT AND RS DECODING

4.1. Various CFFT

Before we apply our CSE algorithm to various CFFTs, we analyze their properties. We can show that DCFFT and TCFFT have the same additive complexity under direct computation. We can also show that TCFFT and ICFFT are equivalent up to permutation and *always have the same additive complexity*. Due to the limited space, the proofs are omitted.

4.2. Full CFFT

With Algorithm 3, we can easily construct CFFT algorithms with reduced additive complexity for lengths up to 1023. The results are given in Table 1. Note that for direct computation

of CFFT, the sum of the additive complexity of $\boldsymbol{A}$ and $\boldsymbol{Q}$ is less than $\boldsymbol{AQ}$ combined for all lengths, and the smaller numbers are provided in Table 1. The multiplicative complexity is the same for all approaches. Compared with direct computation, the CSE algorithm reduces the additive complexity by more than 80% for all variations of CFFT. Compared to the additive complexity of DCFFT in [4], the best results of DCFFT known to us, the CSE algorithm reduces the additive complexity by 12% for lengths of 255 and 511. The additive complexity of DCFFT of length 1023 is not provided in [4].

Table 1. Complexity of Full CFFT

n	Mult.	Additions			
		DCFFT		TC/IC-FFT	Direct
		CSE	[4]	CSE	Comp.
255	586	6900	7919	7815	37279
511	1014	23424	26643	27299	141710
1023	2827	88002	N/A	105180	536093

4.3. Partial CFFT

When decoding (n, k) RS codes, partial FFT is used to evaluate the first $2t$ syndromes, where $t = \lfloor \frac{n-k}{2} \rfloor$. We label such a partial FFT as $(n, 2t)$. In an $(n, 2t)$ partial FFT, we are only concerned about the first $2t$ elements of $\boldsymbol{F}$. Hence we can eliminate unnecessary rows of $\boldsymbol{AQ}$ in DCFFT and $\boldsymbol{Q}$ in TCFFT/ICFFT, respectively. If there are all-zero columns in the reduced matrices, we can eliminate these columns and the corresponding elements of $\boldsymbol{c}$, and in turn eliminate unnecessary rows of $\boldsymbol{P}$, $\boldsymbol{PA}'^{T}$, or $\boldsymbol{PA}^{-1}$. Since $\boldsymbol{Q}$ is more sparse than $\boldsymbol{AQ}$, it is likely to have more all-zero columns after row elimination, which leads to fewer non-one elements in $\boldsymbol{c}$ and thus fewer multiplications. Thus partial TCFFT and ICFFT tend to have lower multiplicative complexity.

We also choose the appropriate permutations of $\boldsymbol{F}$ and $\boldsymbol{f}$ to find more all-zero columns in $\boldsymbol{Q}$ and minimize the number of multiplications, which is identical to the permutation of the basis suggested in [6]. Note that multiple permutations may give the same multiplicative complexity, and we simply choose one randomly in each optimization.

Instead of further reducing multiplications in partial TCFFT and ICFFT, DCFFT can eliminate unnecessary additions by erasing the last $n - 2t$ rows in $\boldsymbol{A}$. Due to the tremendous number of additions in CFFT, it is favorable to use partial DCFFT for less additions. It also makes it easier and faster to run our CSE algorithm since $\boldsymbol{AQ}$ is reduced to a much smaller size.

Partial DCFFT has lower additive complexity, while partial TCFFT and ICFFT have lower multiplicative complexity. To determine which method is better, we use the total number of finite field additions as the metric to compare their complexities. In hardware implementation, a multiplier over $\mathrm{GF}(2^m)$ generated by trinomials requires $m^2 - 1$ XOR and m^2 AND gates [9], while an adder requires m XOR gates. Assuming that XOR and AND gates have the same complexity, the complexity of a multiplier is $2m$ times that of an adder over $\mathrm{GF}(2^m)$. In software implementation, the complexity can be measured by the number of word-level operations [10]. Using the shift and add method as in [10], a multiplication requires $m - 1$ shift and m XOR word-level operations, respectively while an addition needs only one XOR word-level operation. Henceforth we assume that the complexity of a finite field multiplication over $\mathrm{GF}(2^m)$ is $2m - 1$ times as that of an addition. Note that this assumption is in favor of multiplications, which puts CFFT algorithms in a disadvantage since they have reduced multiplicative complexity.

We apply the CSE algorithm to DCFFT and TCFFT to reduce the number of additions. The computational complexities of the $(255, 32)$, $(511, 64)$, and $(1023, 128)$ partial FFTs are compared in Table 2. Among these three lengths, only the result for $(255, 32)$ is available in [6]. Since TCFFT and ICFFT have the same complexity, the CSE algorithm leads to 20% saving on the additive complexity of the $(255, 32)$ cyclotomic partial FFT, compared with that in [6]. For all three partial FFTs, TCFFT minimizes the total number of addition operations.

4.4. Transform-Domain RS Decoding

Replacing the prime-factor FFT with CFFT with reduced complexity proposed above, we propose a new transform-domain decoder as follows:

1. Compute the syndromes by partial CFFT.
2. Use the BMA to obtain the error-locator polynomial.
3. Compute the remaining syndromes by recursive extension using the error-locator polynomial.
4. Compute the error vector by inverse DFT of all syndromes by full CFFT. Finally, the corrected codeword is obtained by adding the received vector and the error vector.

Since our decoder differs from that in [3] only in Steps 1 and 4, we compare the complexity of these two steps in Table 3. We choose partial TCFFT for syndrome evaluation

For Step 4, the saving of CFFT over prime-factor FFT for $(255, 223)$ and $(511, 447)$ RS codes is 25% and 68%, respectively. The advantage of CFFT upon multiplicative complexity is obvious, but due to its large additive complexity CFFT has roughly the same total number of addition operations as prime-factor FFT length 1023. Another advantage of CFFT is its complexity increases proportionally while that of prime-factor FFT varies considerably, depending on the factorization of $2^m - 1$. That is why the saving of CFFT for the $(511, 447)$ RS code is larger than the other two codes.

For Step 1, CFFT reduces the number of multiplications at the expense of more additions. Compared with full FFT,

Table 2. Complexity of Partial CFFT

$(n, 2t)$	DCFFT w/ CSE			TCFFT w/ CSE			ICFFT [6]		
	Mult.	Add.	Total	Mult.	Add.	Total	Mult.	Add.	Total
$(255, 32)$	586	2960	11750	149	4012	6247	149	5046	7281
$(511, 64)$	1014	8298	25536	345	16509	22374	N/A	N/A	N/A
$(1023, 128)$	2827	25124	78837	824	60741	76397	N/A	N/A	N/A

Table 3. Complexity of Transform-Domain RS Decoding

(n, k)		CFFT			Prime-Factor [3]		
		Mult.	Add.	Total	Mult.	Add.	Total
$(255, 223)$	Syndrome	149	4012	6247	852	1804	14584
	Inverse Transform	586	6900	15690	1135	3887	20912
	Total	735	10912	21937	1987	5691	35496
$(511, 447)$	Syndrome	345	16509	22374	5265	7309	96814
	Inverse Transform	1014	23424	41203	6516	17506	128278
	Total	1359	39933	63036	11781	24815	225092
$(1023, 895)$	Syndrome	824	60741	76397	6785	15775	144690
	Inverse Transform	2827	88002	141715	5915	30547	142932
	Total	3651	148743	218112	12700	46322	287622

CFFT achieves greater savings for syndrome evaluation. For the $(255, 223), (511, 447)$, and $(1023, 895)$ RS codes, the reduction of complexity in terms of the total number of addition operations is $57\%, 77\%$, and 47%, respectively.

We observe that CFFT always has lower multiplicative complexity than prime-factor FFT, both in full and partial FFT. Though prime-factor FFT requires fewer additions, the advantage of CFFT on the total complexity is significant. For the three listed RS codes, the total saving in the complexities of Steps 1 and 4 is $38\%, 72\%$, and 24%, respectively.

Acknowledgment

The authors would like to thank Professor Peter Trifonov for providing his results of CFFT

5. REFERENCES

[1] R. E. Blahut, "Transform techniques for error control codes," *IBM J. Res. Dev.*, vol. 23, pp. 299–315, 1979.

[2] T. K. Truong, P. D. Chen, L. J. Wang, I. S. Reed, and Y. Chang, "Fast, prime factor, discrete Fourier transform algorithms over $\mathrm{GF}(2^m)$ for $8 \leq m \leq 10$," *Inf. Sci.*, vol. 176, no. 1, pp. 1–26, Jan. 2006.

[3] T. K. Truong, P. D. Chen, L. J. Wang, and T. C. Cheng, "Fast transform for decoding both errors and erasures of Reed-Solomon codes over $\mathrm{GF}(2^m)$ for $8 \leq m \leq 10$," *IEEE Trans. Commun.*, vol. 54, no. 2, pp. 181–186, Feb. 2006.

[4] P. V. Trifonov and S. V. Fedorenko, "A method for fast computation fo the Fourier transform over a finite field," *Probl. Inf. Transm.*, vol. 39, no. 3, pp. 231–238, 2003.

[5] S. V. Fedorenko, "A method of computation of the discrete Fourier transform over a finite field," *Probl. Inf. Transm.*, vol. 42, no. 2, pp. 139–151, 2006.

[6] E. Costa, S. V. Fedorenko, and P. V. Trifonov, "On computing the syndrome polynomial in Reed-Solomon decoder," *Euro. Trans. Telecomms.*, vol. 15, no. 4, pp. 337–342, 2004.

[7] P. V. Trifonov, *Adaptive Coding in Multi-Carrier Systems*, Ph.D. thesis, Saint-Petersburg State Polytechnic University, 2005.

[8] P. Cappello and K. Steiglitz, "Some complexity issues in digital signal processing," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 32, no. 5, pp. 1037–1041, Oct. 1984.

[9] B. Sunar and C. K. Koc, "Mastrovito multiplier for all trinomials," *IEEE Trans. Commun.*, vol. 48, no. 5, pp. 522–527, May 1999.

[10] A. Mahboob and N. Ikram, "Lookup table based multiplication technique for $\mathrm{GF}(2^m)$ with cryptographic significance," *IEE Proc.-Commun.*, vol. 152, no. 6, pp. 965–974, Dec. 2005.