Optimizing Polynomial Multiplication and Fixed-Weight Sampling for HQC on ARM Cortex-M4

Jihoon Jang[1], Hanbeom Shin[1], Suhri Kim[2], Seokhie Hong[3], Donggeun Kwon[4*]

[1] Korea University, Seoul, South Korea, {jhw1031506,newonetiger}@korea.ac.kr
[2] Sungshin Women's University, Seoul, South Korea, suhrikim@sungshin.ac.kr
[3] SmartM2M, Busan, South Korea, shhong@smartm2m.co.kr
[4] Kunsan National University, Gunsan, South Korea, dgkwon@kunsan.ac.kr

Abstract. In this paper, we present an optimized implementation of Hamming Quasi-Cyclic (HQC) on the ARM Cortex-M4. We optimize (i) the polynomial multiplication and (ii) the support expansion in fixed-weight sampling, and (iii) propose an optional caching strategy that reuses the public transforms and hash recomputed under a fixed key. For the polynomial multiplication, the fixed-constant multiplications in the Frobenius additive FFT (FAFFT) butterfly spend nearly half of their instructions on VMOV data movements between general-purpose and floating-point registers rather than arithmetic. Because minimizing the XOR count alone can increase the total instruction count, we propose a dirty-aware register-allocation policy and an XOR-operation reordering that reduce the VMOV count by up to 48.1% while leaving the XOR count unchanged. We apply these to a multiplication that combines prior FAFFT-CRT methods, and for HQC-1 we further find a 34% sparser FAFFT modulus that lowers the CRT reconstruction cost. For fixed-weight sampling, we rewrite the support expansion with predicated execution and 4-way unrolling, lowering the per-word cost of its inner loop from 22 to 6 cycles while remaining constant-time. On the NUCLEO-L4R5ZI board, our implementation reduces key generation, encapsulation, and decapsulation by up to 33.1%, 34.6%, and 29.8% over the faster of the two prior state-of-the-art implementations, and the optional caching yields a further reduction of up to 32.7% and 18.9% for encapsulation and decapsulation.

Keywords: Post-Quantum Cryptography, HQC, Binary polynomial multiplication, Cortex-M4, Additive FFT

1 Introduction

In March 2025, NIST selected the code-based Hamming Quasi-Cyclic (HQC) [AMAB+23] scheme as an additional post-quantum KEM standard to provide algorithmic diversity beyond the lattice-based ML-KEM [Tea25]. Unlike ML-KEM, the previously selected PQC KEM, the security of HQC reduces to the syndrome decoding problem for quasi-cyclic codes, thereby providing diversity that does not rely on lattice-based assumptions. In HQC-1, HQC-3, and HQC-5, polynomial multiplication is performed over the rings $\mathbb{F}_2[x]/(x^n-1)$ with $n = 17{,}669$, 35,851, and 57,637, respectively. Together with polynomial multiplication, sampling and hashing largely determine the overall performance of HQC, while decapsulation additionally incurs the cost of decoding.

*Corresponding author.

Research on accelerating HQC on the Cortex-M4 has progressed actively around NIST's Round 4 selection, from the Frobenius additive FFT implementation of Lee et al. [LJKH25] to the FAFFT-CRT methods that Jang et al. [JLK+25] and Chen et al. [CCC+26] independently proposed to avoid zero-padding overhead, both reducing the transform size and the number of XOR operations. The actual instruction cost, however, is not determined solely by the XOR count. Since the Cortex-M4 provides only a limited number of GP (general-purpose) registers, data transfers between GP and VFP (vector floating-point) registers occur frequently, so an XOR sequence with fewer XOR operations may incur more VMOV instructions under register pressure and thus a larger total instruction count. In addition, fixed-weight sampling requires repeated mask generation when expanding a support into a dense bit vector, and reusing a fixed key across multiple sessions repeats the same polynomial generation and transform. To reduce these costs, we propose implementation-level optimizations tailored to the ARM Cortex-M4.

1.1 Our Contributions

The main contributions of this paper are summarized as follows:

- **Optimizing HQC Polynomial Multiplication.** We combine algorithm-level optimizations of FAFFT-CRT with register-level optimizations of bit-sliced butterfly assembly. First, we integrate truncated basis conversion and the sharing of the forward transform of the common operand $\mathbf{r}_2$ into the FAFFT-CRT implementation of Jang et al. [JLK+25]. We also search for a new FAFFT evaluation set and reduce the Hamming weight of the FAFFT modulus polynomial for HQC-1 from 165, as reported by Chen et al. [CCC+26], to 109, a reduction of 34%, thereby lowering the reconstruction cost. Next, we show that VMOV instructions account for nearly half of all instructions in bit-sliced butterfly assembly, and that minimizing only the XOR count could instead increase the total instruction cost. To address this issue, we propose a new dirty-aware register-allocation policy and an XOR-operation reordering method that uses the costs of load and spill VMOV instructions as its objective function. As a result, while preserving the EOR count of the butterfly operations, our method reduces the number of VMOV instructions over the state-of-the-art implementation by 40.6%, 48.1%, and 37.6% for HQC-1, HQC-3, and HQC-5, respectively.
- **Optimizing Support Expansion in Fixed-Weight Sampling.** We propose rewriting the inner loop of `vect_write_support_to_vector` (WSV), which expands the support generated by HQC sampling into a dense bit vector, using ARM predicated instructions. To avoid data-dependent timing, the baseline implementation distributed by the HQC team spends six ALU operations per index generating a full-width mask, which an AND and an OR then apply to the output word. However, generating and applying this mask takes a substantial portion of the inner-loop cost. We use the Cortex-M4 IT block and `ORREQ` instructions to directly perform the conditional OR operation, thereby allowing the implementation to remain constant-time without a mask. We further combine this approach with 4-way outer-loop unrolling. As a result, our method reduces the per-word cost of the inner loop from approximately 22 cycles to 6 cycles while remaining constant-time. WSV is invoked 9 times during a single KEM cycle, and approximately 1.2 million inner iterations are performed for HQC-5.
- **Improved HQC Performance on Cortex-M4.** Evaluating the implementation of the two methods above on a NUCLEO-L4R5ZI board, we reduce key generation, encapsulation, and decapsulation by 27.7–33.1%, 27.6–34.6%, and 22.0–29.8% relative to the lower cycle count of the two prior state-of-the-art implementations, Chen et

al. [CCC+26] and Jang et al. [JLK+25], for each operation. We further propose an optional caching technique for environments that handle many sessions under the same public key. It precomputes the regenerated polynomial, its FAFFT forward transform, and the public-key hash once and reuses the stored results instead of recomputing them on every encapsulation and decapsulation. In an experiment assuming that the same public key is reused, this technique reduces encapsulation and decapsulation by up to 32.7% and 18.9% compared to the case without caching.

1.2 Related Work

Binary polynomial multiplication is a major performance bottleneck in HQC, and approaches for accelerating it fall broadly into divide-and-conquer methods and FFT-based methods. Implementations such as `gf2x` [BGTZ23] use schoolbook, Karatsuba, and Toom–Cook multiplication [BGTZ08, VzGG96, CGLD13], with the final word-level products computed by carry-less multiplication, which propagates no carries. On platforms equipped with dedicated carry-less multiplication instructions, such as `PCLMULQDQ` on x86 and `PMULL` on Neon, Karatsuba-based methods have been reported to outperform FFT-based methods for operand sizes below approximately 20 kbit [CGKT22]. The ARM Cortex-M4 targeted in this work, however, provides no such instruction, which makes FFT-based multiplication a competitive alternative.

In SIMD environments, Jang et al. combined branch-free Toom–Cook/Karatsuba decomposition with `VPCLMULQDQ`/`PMULL`-based multiplication, reducing the multiplication cycles of HQC-1 and HQC-3 by 17–32% and observing that FFT-based methods may be more advantageous for HQC-5 [JLK+26, DGK20], while Chen et al. combined additive-FFT multiplication with the Chinese Remainder Theorem (CRT) and added a caching strategy for transformed key polynomials [CCPY26].

On the Cortex-M4, Lee et al. first applied FAFFT to HQC with bit-sliced butterflies, but since the polynomial lengths of HQC slightly exceed powers of two, the inputs had to be zero-padded to length 65,536, incurring substantial transform overhead [LJKH25]. A subsequent study [JLK+25] addressed this padding overhead by lowering the transform size to 32,768 through hybrid FAFFT–CRT and FAFFT–Karatsuba methods and, combining radix-16 multiplication with butterfly XOR-sequence reduction and register scheduling, reduced the multiplication cycles of HQC-1 and HQC-3 by 33.4% and 27.2%, respectively, over the preceding work [LJKH25]. In parallel, Chen et al. [CCC+26] reinterpreted the Encode step of FAFFT as a ring homomorphism and proposed an FAFFT+CRT method that multiplies without padding, using only a single 32,768-point FAFFT and an auxiliary multiplication over a small residue ring. This work builds on the implementation of Jang et al. [JLK+25], integrates the optimizations of this concurrent work [CCC+26], and adopts both implementations as performance baselines.

2 Preliminaries

2.1 Notation

Let $\mathbb{F}_2$ be the binary field. A length-n vector over $\mathbb{F}_2$ is identified interchangeably with a polynomial of the ring $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$, and the product of two such elements is defined as their product in $\mathcal{R}$. Vectors and polynomials are written in lowercase boldface ($\mathbf{u}, \mathbf{v}$). For a set S, $s \xleftarrow{\$} S$ denotes that s is drawn at random from S, and $s \xleftarrow{\$} S$ from *seed* denotes pseudorandom sampling determined by *seed*. Let $\omega(\cdot)$ denote the Hamming weight of a vector, and for a positive integer ω let $\mathcal{R}_\omega := \{\mathbf{v} \in \mathcal{R} \mid \omega(\mathbf{v}) = \omega\}$ be the set of vectors of weight ω. For $\mathbf{v} = (v_0, \ldots, v_{n-1})$ with index 0 as the least significant bit and for

$0 \le n' \le n$, define $\mathsf{Trun}(\mathbf{v}, n')$ as the function that discards the top $n - n'$ bits and keeps only the low n' bits.

2.2 Hamming Quasi-Cyclic (HQC)

HQC is an IND-CCA2 KEM based on the hardness of syndrome decoding for quasi-cyclic codes, obtained by applying the Fujisaki–Okamoto transform with implicit rejection to an IND-CPA-secure public-key encryption scheme (HQC-PKE) [AMAB+23]. Its main components are given in Algorithms 1 and 2, where $\mathtt{H}, \mathtt{G}, \mathtt{J}$ are hash functions. Error correction uses a concatenated code $\mathcal{C}$ of length $n_1 n_2$ that serially combines a Reed-Muller (RM) code of length n_2 and a Reed-Solomon (RS) code of length n_1.

Algorithm 1 Main components in HQC-PKE [AMAB+23].

Global parameters: $(n, n_1, n_2, \omega, \omega_r, \omega_e, \mathcal{C})$

$\mathsf{HQC\text{-}PKE.Keygen}(seed_{\mathsf{PKE}})$
- $(seed_{\mathsf{dk}}, seed_{\mathsf{ek}}) \leftarrow I(seed_{\mathsf{PKE}})$
- $(\mathbf{y}, \mathbf{x}) \xleftarrow{\$} \mathcal{R}_\omega \times \mathcal{R}_\omega$ from $seed_{\mathsf{dk}}$
- $\mathbf{h} \xleftarrow{\$} \mathcal{R}$ from $seed_{\mathsf{ek}}$
- $\mathbf{s} \leftarrow \mathbf{x} + \mathbf{h} \cdot \mathbf{y}$
- **return** $\mathsf{ek}_{\mathsf{PKE}} \leftarrow (seed_{\mathsf{ek}}, \mathbf{s})$, $\mathsf{dk}_{\mathsf{PKE}} \leftarrow seed_{\mathsf{dk}}$

$\mathsf{HQC\text{-}PKE.Encrypt}(\mathsf{ek}_{\mathsf{PKE}}, \mathbf{m}, \theta)$
- $\mathbf{h} \xleftarrow{\$} \mathcal{R}$ from $seed_{\mathsf{ek}}$
- $(\mathbf{r}_2, \mathbf{e}, \mathbf{r}_1) \xleftarrow{\$} (\mathcal{R}_{\omega_r}, \mathcal{R}_{\omega_e}, \mathcal{R}_{\omega_r})$ from θ
- $\mathbf{u} \leftarrow \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$
- $\mathbf{v} \leftarrow \mathcal{C}.\mathsf{Encode}(\mathbf{m}) + \mathsf{Trun}(\mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}, n_1 n_2)$
- **return** $\mathbf{c}_{\mathsf{PKE}} \leftarrow (\mathbf{u}, \mathbf{v})$

$\mathsf{HQC\text{-}PKE.Decrypt}(\mathsf{dk}_{\mathsf{PKE}}, \mathbf{c}_{\mathsf{PKE}} = (\mathbf{u}, \mathbf{v}))$
- $\mathbf{y} \xleftarrow{\$} \mathcal{R}_\omega$ from $seed_{\mathsf{dk}}$
- **return** $\mathbf{m} \leftarrow \mathcal{C}.\mathsf{Decode}(\mathbf{v} - \mathsf{Trun}(\mathbf{u} \cdot \mathbf{y}, n_1 n_2))$

Algorithm 2 Main components in HQC-KEM [AMAB+23].

$\mathsf{HQC\text{-}KEM.Keygen}()$
- $seed_{\mathsf{KEM}} \xleftarrow{\$} \{0,1\}^{|seed|}$
- $(seed_{\mathsf{PKE}}, \sigma) \leftarrow \mathtt{XOF}(seed_{\mathsf{KEM}})$
- $(\mathsf{ek}_{\mathsf{PKE}}, \mathsf{dk}_{\mathsf{PKE}}) \leftarrow \mathsf{HQC\text{-}PKE.Keygen}(seed_{\mathsf{PKE}})$
- $\mathsf{ek}_{\mathsf{KEM}} \leftarrow \mathsf{ek}_{\mathsf{PKE}}$
- $\mathsf{dk}_{\mathsf{KEM}} \leftarrow (\mathsf{ek}_{\mathsf{KEM}}, \mathsf{dk}_{\mathsf{PKE}}, \sigma, seed_{\mathsf{KEM}})$
- **return** $\mathsf{ek}_{\mathsf{KEM}}$, $\mathsf{dk}_{\mathsf{KEM}}$

$\mathsf{HQC\text{-}KEM.Encaps}(\mathsf{ek}_{\mathsf{KEM}})$
- $\mathbf{m} \xleftarrow{\$} \{0,1\}^k$, $salt \xleftarrow{\$} \{0,1\}^{|salt|}$
- $(K, \theta) \leftarrow \mathtt{G}(\mathtt{H}(\mathsf{ek}_{\mathsf{KEM}}) \,\|\, \mathbf{m} \,\|\, salt)$
- $\mathbf{c}_{\mathsf{PKE}} \leftarrow \mathsf{HQC\text{-}PKE.Encrypt}(\mathsf{ek}_{\mathsf{KEM}}, \mathbf{m}, \theta)$
- **return** K, $\mathbf{c}_{\mathsf{KEM}} \leftarrow (\mathbf{c}_{\mathsf{PKE}}, salt)$

$\mathsf{HQC\text{-}KEM.Decaps}(\mathsf{dk}_{\mathsf{KEM}}, \mathbf{c}_{\mathsf{KEM}})$
- Parse $\mathsf{dk}_{\mathsf{KEM}} = (\mathsf{ek}_{\mathsf{KEM}}, \mathsf{dk}_{\mathsf{PKE}}, \sigma, seed_{\mathsf{KEM}})$, $\mathbf{c}_{\mathsf{KEM}} = (\mathbf{c}_{\mathsf{PKE}}, salt)$
- $\mathbf{m}' \leftarrow \mathsf{HQC\text{-}PKE.Decrypt}(\mathsf{dk}_{\mathsf{PKE}}, \mathbf{c}_{\mathsf{PKE}})$
- $(K', \theta') \leftarrow \mathtt{G}(\mathtt{H}(\mathsf{ek}_{\mathsf{KEM}}) \,\|\, \mathbf{m}' \,\|\, salt)$
- $\mathbf{c}'_{\mathsf{PKE}} \leftarrow \mathsf{HQC\text{-}PKE.Encrypt}(\mathsf{ek}_{\mathsf{KEM}}, \mathbf{m}', \theta')$
- $\bar{K} \leftarrow \mathtt{J}(\mathtt{H}(\mathsf{ek}_{\mathsf{KEM}}) \,\|\, \sigma \,\|\, \mathbf{c}_{\mathsf{KEM}})$
- **if** $\mathbf{m}' = \perp$ **or** $\mathbf{c}'_{\mathsf{PKE}} \neq \mathbf{c}_{\mathsf{PKE}}$ **then** $K' \leftarrow \bar{K}$
- **return** K'

HQC involves (i) binary polynomial multiplication in the ring $\mathcal{R}$ ($\mathbf{h} \cdot \mathbf{r}_2$, $\mathbf{s} \cdot \mathbf{r}_2$, and $\mathbf{u} \cdot \mathbf{y}$) and (ii) the sampling that generates the fixed-weight vectors $(\mathbf{x}, \mathbf{y}, \mathbf{r}_1, \mathbf{r}_2, \mathbf{e})$ and the public vector $\mathbf{h}$ from a SHAKE-based XOF. Together with hashing, these two operations dominate the performance of HQC, and this paper targets the multiplication and the sampling for optimization.

2.3 Frobenius Additive FFT (FAFFT)

The product of two elements $\mathbf{a}, \mathbf{b}$ of the ring $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$ can be computed with an additive FFT. We first take l to be the smallest power of two greater than $2n - 2$, and

regard the coefficients as elements of a tower field $\mathbb{F}_{2^m}$ with $2^m \geq l$ and m a power of two, viewing $\mathbf{a}, \mathbf{b}$ as polynomials over $\mathbb{F}_{2^m}$. We then (i) evaluate the two polynomials at l points with the additive FFT, (ii) multiply them pointwise, (iii) recover the product polynomial with the inverse additive FFT, and (iv) reduce it into $\mathcal{R}$. Since all input coefficients lie in $\mathbb{F}_2$, the recovered product also has its coefficients in $\mathbb{F}_2$, so mapping it back from $\mathbb{F}_{2^m}$ to $\mathbb{F}_2$ requires no additional computation [LJKH25].

The additive FFT [GM10, Can89, LANH16] recursively evaluates a polynomial $f \in \mathbb{F}_{2^m}[x]$ at many points. For a Cantor basis $\{v_0, \ldots, v_{m-1}\}$ of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$, let $W_k = \langle v_0, \ldots, v_{k-1} \rangle$ $(0 \leq k \leq m, |W_k| = 2^k, W_0 = \{0\})$ be the associated subspaces, and define the subspace polynomials $s_0(x) = x, s_{i+1}(x) = s_i(x)^2 + s_i(x)$. Each s_i is linear over $\mathbb{F}_2$ with $\deg s_i = |W_i| = 2^i$, and $s_k(x) = \prod_{a \in W_k}(x - a)$ vanishes on W_k. The additive FFT then consists of a butterfly recursion that splits f into two halves using s_i, whose key operation at each butterfly step is

$$h_0(X) \leftarrow p_0(X) + s_i(\alpha) \cdot p_1(X), \qquad h_1(X) \leftarrow h_0(X) + p_1(X),$$

where $s_i(\alpha)$ is a fixed constant determined at each butterfly node. Since all operations are carried out over $\mathbb{F}_{2^m}$, they reduce to XORs and carry-less multiplications. In particular, with a Cantor basis, $s_k(x) = x^{2^k} + x$ becomes sparse when k is a power of two, so the low-level butterflies reduce to simple bit operations.

The Frobenius additive FFT (FAFFT) reduces the number of evaluation points using the Frobenius map $\phi_2 : a \mapsto a^2$ [LCK+18]. When the input polynomial f has coefficients in $\mathbb{F}_2$,$f(\phi_2(a)) = f(a^2) = f(a)^2 = \phi_2(f(a))$ holds, so once the value $f(a)$ at one point is given, the values $f(a^2), f(a^4), \ldots$ are also fixed. For a set Σ, let $\mathrm{Ord}_{\phi_2}(\Sigma)$ be the smallest integer j with $\phi_2^j(\Sigma) = \Sigma$, and call Σ a Frobenius partition of the evaluation domain Ω when

$$\Omega = \Sigma \cup \phi_2(\Sigma) \cup \cdots \cup \phi_2^{j-1}(\Sigma)$$

is a disjoint union. Li et al. [LCK+18] showed that, for the Cantor basis, $\phi_2(W_i) = W_i$, and for $k = \log_2 l - \log_2 m$ the set $\Sigma = v_{k+m/2} + W_k$ is a Frobenius partition with $\mathrm{Ord}_{\phi_2}(\Sigma) = m$ and $|\Sigma| = 2^k = l/m$. Evaluating f at the l points of Ω therefore reduces to evaluating at the l/m representative points of Σ, and the values at the remaining points are already fixed by the Frobenius relation and are not computed separately. Consequently, the product $\mathbf{c}$ of two polynomials $\mathbf{a}, \mathbf{b}$ is obtained by multiplying $\mathsf{FAFFT}(\mathbf{a})$ and $\mathsf{FAFFT}(\mathbf{b})$ pointwise at the l/m evaluation points and then applying the inverse transform FAFFT^{-1}.

In the Cortex-M4 implementation, this computation is carried out in a bit-sliced form over the tower field $\mathbb{F}_2 \subset \mathbb{F}_{2^2} \subset \cdots \subset \mathbb{F}_{2^{32}}$, and the additions in the butterfly are handled as XORs between bit planes. Among the field multiplications in the butterfly, multiplication by the constant fixed at each level is a fixed $\mathbb{F}_2$-linear map, so it is implemented as a dedicated straight-line XOR sequence [LJKH25], whereas multiplication by the constant that varies with the evaluation point is performed as a general tower-field multiplication. The former is straight-line code consisting of XORs and register moves, and this paper optimizes its instruction scheduling in Section 3.

However, the polynomial length n of HQC is slightly larger than a power of two, so applying the FAFFT directly incurs the overhead of zero-padding up to the next power of two, which is 65,536 for HQC-1. To reduce this, a method was proposed that uses the Chinese Remainder Theorem (CRT) to decompose the polynomial ring into a product of a ring suited to the FAFFT and a small residue ring, so that the multiplication is handled with a single FAFFT of half that size and an auxiliary multiplication in the small residue ring [JLK+25, CCC+26]. Among these, this paper uses the implementation of Jang et al. [JLK+25], with the techniques of Chen et al. [CCC+26] integrated, as the basis for its own optimizations.

2.4 ARM Cortex-M4

The ARM Cortex-M4 is a 32-bit embedded processor based on the ARMv7E-M architecture. It provides the Thumb-2 instruction set and DSP extensions such as single-cycle multiplication and multiply-accumulate (MAC), but it has no carry-less multiplication instruction such as x86's `PCLMULQDQ` or RISC-V's `clmul`. The Thumb-2 instruction set supports predicated execution through `IT` (If-Then) blocks, which lets a condition flag control whether an instruction takes effect without branching. When the condition is false, the instruction still occupies its execution slot but writes no result, which is useful for constant-time implementations that avoid secret-dependent branches.

The ARM Cortex-M4 has 16 GP registers, `r0`–`r15`. Of these, `r13` (SP) and `r15` (PC) are reserved, leaving 14 available for general use. The optional floating-point unit (FPv4-SP) additionally provides 32 single-precision registers `s0`–`s31`, which can serve not only for floating-point operations but also as extra storage. However, data movement between the GP registers and the floating-point registers is possible only through the `vmov` instruction and incurs a cycle cost per move. This `vmov` cost is the key overhead of the bit-sliced FAFFT butterfly, which this paper minimizes in Section 3.

3 Polynomial Multiplication Optimization

This section presents our optimizations to the polynomial multiplication that dominates the cost of HQC. Building on the FAFFT-CRT implementation of Jang et al. [JLK+25], we incorporate the optimizations of Chen et al. [CCC+26] and lower the FAFFT modulus weight further for HQC-1 (Section 3.1). We then minimize the VMOV overhead of the bit-sliced butterfly through register allocation and XOR reordering (Section 3.2), and report the resulting multiplication cycles in Section 3.3.

3.1 Combining FAFFT-CRT Multiplication Techniques

The dominant operation of HQC is the multiplication $\mathbf{c} = \mathbf{a} \cdot \mathbf{b}$ in $\mathcal{R} = \mathbb{F}_2[x]/(x^n - 1)$, where $n = 2^d + r$ for a small r. Two concurrent works, Jang et al. [JLK+25] and Chen et al. [CCC+26], both compute it by combining the FAFFT with the CRT, splitting $\mathcal{R}$ into a 2^d-point FAFFT component and a small residue ring handled by a radix-16 multiplication, which avoids the zero-padding to 2^{d+1} that a direct transform would otherwise require.

We take the implementation of Jang et al. [JLK+25] as our base and incorporate three optimizations of Chen et al. [CCC+26]. First, a *truncated basis conversion*: as each operand fills only part of the transform, the conversion produces the occupied prefix instead of the full 2^d coefficients, namely $16{,}384 + 2{,}048$ for the 32,768-point transform of HQC-1 and $32{,}768 + 4{,}096$ for the 65,536-point transform of HQC-3. Second, a *low-weight FAFFT modulus* that keeps the sparse-dense reconstruction cheap; here we go beyond Chen et al. and find a sparser modulus for HQC-1 (Section 3.1.1). Third, *sharing* the forward transform of the ephemeral operand $\mathbf{r}_2$, which is common to the products $\mathbf{h} \cdot \mathbf{r}_2$ and $\mathbf{s} \cdot \mathbf{r}_2$ in encapsulation and decapsulation, so that it is computed only once.

The residue ring itself is handled by a radix-16 polynomial multiplication, and this is where the two prior works differ. For HQC-1 and HQC-3, the residue product occupies 2,570 and 6,166 bits, that is $\lceil 2570/32 \rceil = 81$ and $\lceil 6166/32 \rceil = 193$ 32-bit words. Jang et al. [JLK+25] multiply the residue at these exact sizes of 81 and 193, whereas Chen et al. [CCC+26] zero-pad them to the implementation-friendly sizes of 96 and 256 while skipping the high-order words that the truncated product does not need. For HQC-1 the padding from 81 to 96 is small, so the truncated multiplication of Chen et al. is faster, but for HQC-3 the padding from 193 to 256 is large enough that the exact multiplication of Jang et al. is faster. We therefore take the residue multiplication of Chen et al. for HQC-1 and that of Jang et al. for HQC-3.

These optimizations apply to HQC-1 and HQC-3, whose parameters admit the CRT split. HQC-5 uses a pure double-size FAFFT with no residue ring.

Beyond the multiplication, we also adopt the RS/RM decoder of Chen et al. [CCC+26] for the decapsulation decoding. Decoding is not a multiplication, but it is another component where the base implementation of Jang et al. [JLK+25] and that of Chen et al. differ. The Reed-Solomon step of HQC recovers the error-locator polynomial from the syndromes, and Chen et al. replace the Berlekamp-Massey solver of the base decoder with an Extended Euclidean Algorithm (EEA). The regular data flow of the EEA aligns with their $\mathbb{F}_{256}$-optimized arithmetic, implemented as unrolled radix-16 matrix-vector products, which makes the decoder faster on the Cortex-M4 at the cost of the larger code discussed in Section 6. Since decoding occurs only in decapsulation, adopting it further reduces the decapsulation cost.

3.1.1 A Low-Weight FAFFT Modulus

In the Hybrid FAFFT-CRT of Section 3.1, the product is reconstructed as $C(x) = C_1(x) + t(x) \cdot \mathcal{P}(x)$, where $\mathcal{P}(x)$ is the vanishing polynomial of the FAFFT evaluation domain. Because $\mathcal{P}(x)$ is fixed, public, and sparse, the reconstruction term $t(x) \cdot \mathcal{P}(x)$ is a constant-time sparse-dense multiplication whose cost is proportional to the Hamming weight $\omega(\mathcal{P})$; the sparser $\mathcal{P}(x)$ is, the cheaper the reconstruction.

The evaluation set that determines $\mathcal{P}(x)$ is not unique, so $\omega(\mathcal{P})$ depends on this choice. Earlier additive-FFT work did not optimize it, leaving $\omega(\mathcal{P}) = 501$ for HQC-1 [JLK+25], whereas Chen et al. [CCC+26] minimized it to 165 for HQC-1 and 199 for HQC-3. We search for the minimizing evaluation set independently for each parameter: for HQC-3 the value 199 is already minimal and we retain it, while for HQC-1 we find an evaluation set with $\omega(\mathcal{P}) = 109$, below the previous 165 (a 34% reduction), which directly lowers the reconstruction cost. HQC-5 uses a pure double-size FAFFT with no residue ring, so this optimization does not apply. The new HQC-1 evaluation set also changes the butterfly's fixed-constant multiplications and hence their XOR sequences; since these have no prior implementation, the HQC-1 reference of Section 3.2 is reconstructed accordingly (marked with an asterisk).

3.2 VMOV Minimization via Register Allocation and XOR Reordering

We show the limitation of the existing framework that minimizes only the XOR count in the assembly implementation of FAFFT butterfly multiplication operations, and propose a novel framework that optimizes the number of data movements between GP registers and VFP registers, namely VMOV instructions, through register allocation and XOR reordering.

3.2.1 Motivation

In the butterfly stage of FAFFT, terms of the form $s_i(\alpha) \cdot p_1(X)$ are computed repeatedly. When $m = 32$, $s_i(\alpha)$ decomposes as $s_i(\alpha) = w + v$, where $w \in W_k$ and $v \in \{v_{17}, \ldots, v_{k+16}\}$ is fixed per stage and hence common to all nodes of that stage. Therefore, it can be written as

$$s_i(\alpha) \cdot p_1(X) = w \cdot p_1(X) + v \cdot p_1(X).$$

Since $k < 16$, we have $W_k \subseteq W_{16} = \mathbb{F}_{2^{16}}$, so w lies in a proper subfield of $\mathbb{F}_{2^{32}}$ and $w \cdot p_1(X)$ can be computed by multiplications in that subfield rather than in $\mathbb{F}_{2^{32}}$ [CCK21]. In $v \cdot p_1(X)$, on the other hand, v is a fixed constant, so this operation is a multiplication by a fixed constant in $\mathbb{F}_{2^{32}}$. Lee et al. [LJKH25] observed that this is a 32×32 $\mathbb{F}_2$-linear transformation mapping a 32-bit input vector to a 32-bit output vector, and can therefore be represented as a matrix-vector multiplication implementable using only XOR operations.

The butterfly operations discussed in this section are straight-line implementations of multiplications by the fixed constants $v_{17}, \ldots, v_{28}$. Although the corresponding linear transformations are expressed using EOR instructions, the implementation must handle 32 state variables with only 14 available GP registers. Values are therefore transferred between GP registers and dedicated VFP registers using VMOV instructions, and we use the sum of the EOR and VMOV counts as the implementation cost. This register pressure motivates the following observations.

Observation 1: Half of the instructions are data movement rather than arithmetic. As shown in Table 5, across the 12 butterfly assemblies of Jang et al. [JLK+25], the total number of EOR instructions is 2,034, while the total number of VMOV instructions is 1,992. In other words, the number of XOR instructions implementing the $\mathbb{F}_2$-linear transformations is almost the same as the number of data movement instructions. For the implementation corresponding to each fixed constant v, VMOV instructions account for 47–51% of the total instructions. In particular, for three constants (v_{23}, v_{26}, v_{27}), the number of VMOV instructions exceeds the number of EOR instructions. For example, the implementation for v_{26} consists of 179 EOR instructions and 187 VMOV instructions, so it spends more clock cycles on data movement than on the fixed-constant multiplication itself. These results show that, in the actual Cortex-M4 implementation, roughly half of the clock cycles correspond not to the $\mathbb{F}_2$-linear transformation itself but to the overhead caused by register pressure.

Observation 2: Reducing the XOR count does not always reduce the instruction cost. As in Lee et al. [LJKH25] and Jang et al. [JLK+25], we also apply the framework of Xiang et al. [XZL+20] independently to each of the 12 fixed-constant multiplications to further search for shorter XOR sequences. As a result, we find new sequences with fewer XOR operations for five constants $(v_{18}, v_{19}, v_{20}, v_{24}, v_{28})$. We convert these five sequences into assembly implementations with the same functional structure as the implementation of Jang et al. [JLK+25] and compare them with the original implementation. Our search results are shown in Table 1. The number of EOR instructions decreases by 1–3 for each constant, but the number of VMOV instructions increases by 14–36. As a result, all five implementations have larger total instruction counts than those of Jang et al.

Table 1: Results of converting shorter XOR sequences found using the framework of Xiang et al. [XZL+20] into assembly.

	Jang et al. [JLK+25]			New XOR sequence			
v_i	#EOR	#VMOV	Total	#EOR	#VMOV	Total	ΔTotal
v_{18}	153	139	292	150	157	307	+15
v_{19}	166	161	327	165	175	340	+13
v_{20}	172	167	339	170	186	356	+17
v_{24}	180	171	351	178	207	385	+34
v_{28}	183	173	356	181	196	377	+21

These results show that reducing the XOR count does not necessarily reduce the total instruction count. Therefore, optimization on the Cortex-M4 must account for the VMOV cost caused by register pressure. Motivated by these observations, we minimize VMOV instructions by reconsidering the execution order and register allocation while preserving the resulting matrix M.

3.2.2 XOR Sequences and the Implementation Cost Model

Each butterfly function is a linear transformation $M \in \mathrm{GL}(m, \mathbb{F}_2)$ with $m = 32$, which decomposes into a product of type-1 and type-3 elementary matrices. A type-3 matrix $E(i+j)$ adds row j to row i of the identity and corresponds to an in-place XOR, $x_i \leftarrow x_i \oplus x_j$. A type-1 matrix $E(i \leftrightarrow j)$ swaps rows i and j, exchanging the positions of two variables. Thus M can be written as

$$M \;=\; E(i_n + j_n) \cdots E(i_1 + j_1)\, E(i'_s \leftrightarrow j'_s) \cdots E(i'_1 \leftrightarrow j'_1) \;\in\; \mathrm{GL}(m, \mathbb{F}_2). \tag{1}$$

A type-1 elementary matrix only relabels the register holding each variable and therefore incurs no cost in the actual implementation. The type-3 part of (1) corresponds to an ordered sequence of in-place XOR operations applied to the state variables $x_0, \ldots, x_{m-1}$, from $x_{i_1} \leftarrow x_{i_1} \oplus x_{j_1}$ through $x_{i_n} \leftarrow x_{i_n} \oplus x_{j_n}$ in turn. We call such a sequence of operations an *XOR sequence* and denote it by L. Its length n corresponds directly to the number of EOR instructions needed to implement the transformation. The existing framework of Xiang et al. [XZL+20] takes exactly this count as its objective and, among the decompositions that preserve M, searches for an XOR sequence with fewer type-3 elementary matrices under a set of reduction rules.

In the actual implementation, an EOR instruction can operate only on operands held in GP registers, so executing $x_i \leftarrow x_i \oplus x_j$ requires both the source x_j and the destination x_i to reside there. An XOR sequence, however, specifies only the source and destination of each operation, not the registers in which they are placed, and the register constraint of Section 3.2.1 prevents all state variables from residing in GP registers simultaneously; a register-allocation policy is therefore needed to decide which variables to keep at each point. If a required variable is absent, its VFP copy must be brought in, incurring one load VMOV, and if all GP registers are occupied, a resident variable must first be evicted. We call a resident variable *dirty* if its value has been updated since its last load and *clean* otherwise, writing $\mathcal{D}$ for the set of dirty variables. Evicting a clean variable is free, since its VFP copy is still valid, whereas evicting a dirty variable costs one spill VMOV to write its latest value back; the VMOV count of an instruction sequence is therefore the number of loads plus the number of dirty-variable spills.

The EOR count n is fixed once the XOR sequence is chosen, whereas the VMOV count varies with the register-allocation policy. The only variable term of the implementation cost—the sum of the EOR and VMOV counts—is thus the VMOV count, and the same XOR sequence may translate into instruction sequences of different lengths under different policies. Reducing the actual instruction count therefore requires a register-allocation policy that accounts for the cost of moving values between GP and VFP registers.

3.2.3 Reducing VMOV Costs via Register Allocation

We now describe how to allocate registers for a fixed XOR sequence so as to minimize the VMOV count. We first recall the farthest-first policy used in prior work and expose its limitation, and then propose our Spill-Penalized Farthest-First policy.

The Farthest-First Policy and Its Limitation When the XOR sequence is fixed, the order of operations is known in advance, which enables an eviction policy that reduces unnecessary reloads. The prior works of Lee et al. [LJKH25] and Jang et al. [JLK+25] use the farthest-first (FF) policy, which, whenever all GP registers are occupied, evicts the variable whose next use lies farthest in the future. By keeping variables that will soon be reused in GP registers, this policy aims to reduce subsequent load VMOVs.

Let $\mathrm{nextuse}(x, t)$ denote the first time after step t at which variable x is used again, set to ∞ if it is never used again. Let $\mathcal{R}_t$ denote the set of eviction candidates at step t,

excluding the source and destination of the XOR operation about to be executed. The FF policy then selects the eviction target as

$$x_u = \arg\max_{x \in \mathcal{R}_t} \text{nextuse}(x, t).$$

That is, FF evicts the candidate whose next use is farthest away. Since the entire XOR sequence is known in advance, each variable's next-use time can be computed exactly.

However, FF decides the eviction target solely from the next-use time and thus ignores the cost difference due to a variable's dirty status. Evicting a clean variable incurs no VMOV, whereas evicting a dirty variable incurs one spill VMOV to write its latest value back to a VFP register. Consequently, when candidates share the same next-use time, or when a dirty variable's next use is only slightly farther than a clean one's, evicting the dirty variable can make the spill cost outweigh the gain from reduced reloads.

Table 2 illustrates this limitation with $K = 3$ available GP registers. After the second operation, x_0, x_1, and x_2 are resident, with x_0 dirty from having been the destination of both operations and x_1, x_2 clean. The third operation $x_1 \leftarrow x_1 \oplus x_3$ then requires loading x_3, so one resident variable must be evicted; excluding the destination x_1 leaves x_0 and x_2, both of which are reused at the fourth operation and hence share the same next-use time. FF breaks this tie without regard to dirty status and may thus evict x_0, spilling its latest value and reloading it one operation later, which brings the total to 6 VMOVs. Even though FF accounts for the order of future uses, it does not reflect the eviction cost that depends on dirty status.

Table 2: Register allocation with plain FF under an unfavorable tie-breaking.

t	XOR operation	Before		After		Required `VMOV`	#`VMOV`
		Dirty	Clean	Dirty	Clean		
1	$x_0 \leftarrow x_0 \oplus x_1$	$\emptyset$	$\emptyset$	$\{x_0\}$	$\{x_1\}$	load x_0, load x_1	2
2	$x_0 \leftarrow x_0 \oplus x_2$	$\{x_0\}$	$\{x_1\}$	$\{x_0\}$	$\{x_1, x_2\}$	load x_2	1
3	$x_1 \leftarrow x_1 \oplus x_3$	$\{x_0\}$	$\{x_1, x_2\}$	$\{x_1\}$	$\{x_2, x_3\}$	spill x_0, load x_3	2
4	$x_0 \leftarrow x_0 \oplus x_2$	$\{x_1\}$	$\{x_2, x_3\}$	$\{x_0, x_1\}$	$\{x_2\}$	load x_0 (drop x_3)	1
5	$x_1 \leftarrow x_1 \oplus x_2$	$\{x_0, x_1\}$	$\{x_2\}$	$\{x_0, x_1\}$	$\{x_2\}$	-	0

The Spill-Penalized Dirty-Aware Farthest-First Policy To reflect in victim selection the spill cost that FF overlooks, we propose the Spill-Penalized Dirty-Aware Farthest-First (SPF) policy. SPF retains FF's nextuse-based selection criterion but incorporates into the score the spill cost that may arise when a dirty variable is evicted. For each eviction candidate $x \in \mathcal{R}_t$ at step t, we define the following score.

$$S_p(x, t) = \text{nextuse}(x, t) - p \cdot \mathbf{1}[x \in \mathcal{D}],$$

Here $\mathbf{1}[x \in \mathcal{D}]$ is 1 if x is dirty and 0 otherwise, and p is the penalty imposed on evicting a dirty variable. SPF evicts the candidate with the largest S_p, breaking ties in favor of clean variables. Thus $p = 0$ is not plain FF but FF augmented with a dirty-aware tie-break. When $p > 0$, a dirty variable's score is reduced, so the clean variable is evicted even if the dirty one is used later, as long as the difference is at most p; that is, p is the loss in next-use distance that SPF accepts in order to avoid a dirty spill.

Table 3 applies SPF to the same XOR sequence as the FF example. Just before the third operation, the candidates x_0 and x_2 have equal next-use times, but where FF breaks the tie by evicting the dirty x_0 and incurring a spill (Table 2), SPF evicts the clean x_2. The reload count is unchanged—x_2 is reloaded at the fourth operation, just as FF reloads x_0—so avoiding the spill reduces the total VMOV count from 6 to 5.

Table 3: Register allocation with SPF.

t	XOR operation	Before Dirty	Before Clean	After Dirty	After Clean	Required `VMOV`	#`VMOV`
1	$x_0 \leftarrow x_0 \oplus x_1$	$\emptyset$	$\emptyset$	$\{x_0\}$	$\{x_1\}$	load x_0, load x_1	2
2	$x_0 \leftarrow x_0 \oplus x_2$	$\{x_0\}$	$\{x_1\}$	$\{x_0\}$	$\{x_1, x_2\}$	load x_2	1
3	$x_1 \leftarrow x_1 \oplus x_3$	$\{x_0\}$	$\{x_1, x_2\}$	$\{x_0, x_1\}$	$\{x_3\}$	load x_3 (drop x_2)	1
4	$x_0 \leftarrow x_0 \oplus x_2$	$\{x_0, x_1\}$	$\{x_3\}$	$\{x_0, x_1\}$	$\{x_2\}$	load x_2 (drop x_3)	1
5	$x_1 \leftarrow x_1 \oplus x_2$	$\{x_0, x_1\}$	$\{x_2\}$	$\{x_0, x_1\}$	$\{x_2\}$	-	0

To assess the effect of the penalty p, we measure the VMOV count for the twelve butterfly functions $(v_{17}, \ldots, v_{28})$ of HQC-5 while varying p and keeping the XOR sequences of the implementation of Jang et al. [JLK+25] fixed. As Figure 1 shows, the count is 1,849 at $p = 0$, decreases as p increases, reaches its minimum of 1,804 at $p = 11$, and then rises again, forming a U shape. This reflects a trade-off between avoiding dirty spills and incurring additional clean reloads: a small p makes SPF behave much like FF and underprotect dirty variables, whereas a large p evicts clean variables that will soon be reused and thus increases load VMOVs. We therefore use $p = 11$, which yields the lowest VMOV count, as the default penalty for SPF.

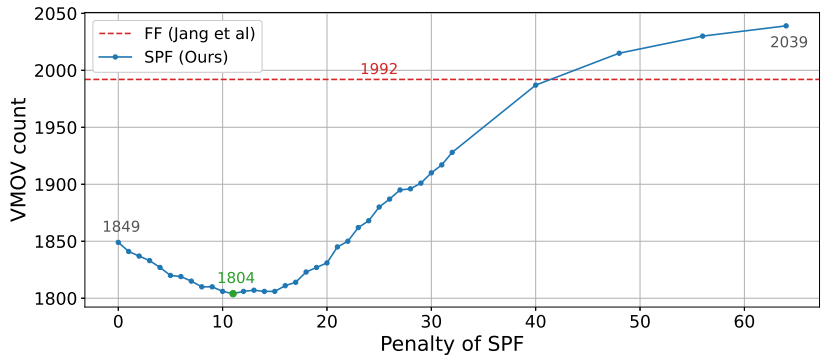

Figure 1: VMOV count as a function of the SPF penalty p over all v_i for HQC-5.

3.2.4 Reducing VMOV Costs via XOR Reordering

The implementation cost model of Section 3.2.2 deterministically computes the VMOV count of an XOR sequence L under a fixed register-allocation policy. In this subsection, among the XOR sequences that preserve the resulting matrix M, we search for a new order that reduces the VMOV cost.

Commutativity and the Search Space The VMOV count depends not only on the register-allocation policy but also on the order of the type-3 operations. We therefore rearrange only the order of operations while preserving the multiset of XOR operations and the resulting matrix M. Property 1 gives three cases in which two type-3 elementary matrices commute: Case 1, where the two operations use disjoint indices, and Cases 2 and 3, where they share the same source and the same destination, respectively.

Property 1 (Xiang et al. [XZL+20], Property 3)**.** For distinct integers i, j, k, l, the

following identities hold.

$$\text{Case 1. } E(k+l)\,E(i+j) = E(i+j)\,E(k+l),$$
$$\text{Case 2. } E(i+j)\,E(k+j) = E(k+j)\,E(i+j),$$
$$\text{Case 3. } E(i+j)\,E(i+k) = E(i+k)\,E(i+j).$$

Two operations are said to *commute* if they satisfy one of the three identities of Property 1. A non-commuting pair must retain its original relative order, and for each such pair, we add an edge from the earlier operation to the later one, forming a dependency graph D. A commuting pair, by contrast, preserves the resulting matrix M under a swap of its order and is therefore a candidate for reordering to reduce the VMOV cost.

Table 4 is a simple example showing that reordering can reduce the VMOV count, using the same XOR sequence as the FF and SPF examples above with $K = 3$ GP registers and the SPF policy. In the original order L, x_2 is loaded at o_2 but evicted at o_3 to make room for x_3, and must therefore be reloaded at o_4. Since $o_2 = E(0{+}2)$ and $o_3 = E(1{+}3)$ use disjoint indices, they commute by Case 1 of Property 1. In the reordered order L', o_3 runs first and loads x_3 while a register is still free, and x_2 is then loaded at o_2 and stays resident through o_4 and o_5, so the reload disappears. Both orders implement the same M, yet L' uses one fewer VMOV, showing that reordering alone—while preserving the resulting matrix M—can change register pressure and thereby lower the VMOV cost.

Table 4: Example of VMOV reduction by commuting two XOR operations.

t	Original order L			Reordered order L'		
	XOR operation	Required `VMOV`	#`VMOV`	XOR operation	Required `VMOV`	#`VMOV`
1	$o_1 : x_0 \leftarrow x_0 \oplus x_1$	load x_0, load x_1	2	$o_1 : x_0 \leftarrow x_0 \oplus x_1$	load x_0, load x_1	2
2	$o_2 : x_0 \leftarrow x_0 \oplus x_2$	load x_2	1	$\mathbf{o_3} : x_1 \leftarrow x_1 \oplus x_3$	load x_3	1
3	$o_3 : x_1 \leftarrow x_1 \oplus x_3$	load x_3 (drop x_2)	1	$\mathbf{o_2} : x_0 \leftarrow x_0 \oplus x_2$	load x_2 (drop x_3)	1
4	$o_4 : x_0 \leftarrow x_0 \oplus x_2$	load x_2 (drop x_3)	1	$o_4 : x_0 \leftarrow x_0 \oplus x_2$	none	0
5	$o_5 : x_1 \leftarrow x_1 \oplus x_2$	none	0	$o_5 : x_1 \leftarrow x_1 \oplus x_2$	none	0
		Total	5		Total	4

Reordering via Simulated Annealing The number of reordering candidates that preserve the resulting matrix M varies greatly with the structure of the dependencies, so an exhaustive search over all possible orders is impractical. Moreover, how much a local change in operation order alters the VMOV count must itself be recomputed by the SPF-based forward simulation. For such a problem—with a large search space and a cost that must be evaluated directly for each candidate—stochastic local search, which repeatedly generates and evaluates candidates, is well suited. We therefore use simulated annealing [KGJV83] to search for XOR sequences with a small VMOV count. A plain greedy search accepts only cost-decreasing moves and is thus prone to getting stuck in local minima, whereas simulated annealing also accepts cost-increasing moves probabilistically, allowing broad exploration early on and convergence around improved candidates later.

We search over the orders of the XOR sequence using the SPF-based VMOV count as the objective. A neighboring candidate is generated by an insertion move that picks one operation and moves it to another position, allowed only when the moved operation commutes with every operation it passes. Since such a move is a composition of swaps of adjacent commuting pairs, the resulting XOR sequence preserves the resulting matrix M. With probability $1/2$ a move swaps two adjacent operations, and otherwise it relocates an operation to a position within distance w; the former fine-tunes the cost, while the latter provides the long-range moves needed to escape local minima. Each new candidate is accepted according to the Metropolis rule: a move is always accepted if it does not

increase the cost, and a move that increases the cost by Δ is accepted with probability $e^{-\Delta/t}$. The temperature t is lowered geometrically from t_0 to t_1.

Experimental Results Table 5 compares, for each parameter, our implementation against a prior one that uses the same XOR sequences, so the EOR count of each function is identical and only the VMOV count differs. The reference is the implementation of Jang et al. [JLK+25] for HQC-5 and of Chen et al. [CCC+26] for HQC-3. For HQC-1, the new evaluation set we choose for the FAFFT (Section 3.1.1) induces XOR sequences that have no prior implementation to compare against, so we take as the reference these same sequences allocated with the farthest-first policy of Lee et al. [LJKH25] (marked with an asterisk). The number of these fixed-constant multiplication functions is 10, 11, and 12 for HQC-1, HQC-3, and HQC-5, respectively. Keeping the XOR sequences fixed, our optimization reduces only the VMOV count by changing the operation order and register allocation, lowering the total VMOV count by 40.6% (from 1,866 to 1,109) for HQC-1, 48.1% (from 2,396 to 1,243) for HQC-3, and 37.6% (from 1,992 to 1,244) for HQC-5.

Table 5: EOR and VMOV counts per function under XOR reordering.

	inv_i	inv_1	inv_2	inv_3	inv_4	inv_5	inv_6	inv_7	inv_8	inv_9	inv_{10}	inv_{11}	inv_{12}	Total
HQC-1														
EOR		183	187	163	184	189	192	186	187	186	183	–	–	1,840
VMOV	[LJKH25]*	197	185	163	181	185	209	197	199	173	177	–	–	1,866
	Ours	117	110	101	104	113	121	110	112	113	108	–	–	**1,109**
	Δ	−80	−75	−62	−77	−72	−88	−87	−87	−60	−69	–	–	**−757**
HQC-3														
EOR		167	182	180	183	186	192	193	182	187	185	189	–	2,026
VMOV	[CCC+26]	194	214	216	200	238	236	226	204	220	224	224	–	2,396
	Ours	105	114	114	110	115	117	114	109	116	109	120	–	**1,243**
	Δ	−89	−100	−102	−90	−123	−119	−112	−95	−104	−115	−104	–	**−1,153**
HQC-5														
EOR		159	153	166	172	166	160	163	180	180	179	173	183	2,034
VMOV	[JLK+25]	153	139	161	167	165	159	165	171	171	187	181	173	1,992
	Ours	100	88	105	110	103	98	97	104	110	110	111	108	**1,244**
	Δ	−53	−51	−56	−57	−62	−61	−68	−67	−61	−77	−70	−65	**−748**

* Our HQC-1 XOR sequence has no prior implementation; the reference is that same sequence allocated with the farthest-first policy of Section 3.2.3, following the implementation of Lee et al. [LJKH25].

3.3 Multiplication Performance

We benchmark the polynomial multiplication in isolation on the NUCLEO-L4R5ZI at 20 MHz, comparing our implementation against those of Chen et al. [CCC+26] and Jang et al. [JLK+25]. These figures reflect both the multiplication construction of Section 3.1 and the VMOV minimization of Section 3.2, so the comparison captures the joint effect of the two. Table 6 reports the results.

Table 6: Clock cycles of the polynomial multiplication.

Parameter	[CCC+26]	[JLK+25]	Ours	Reduction
HQC-1	1,753,860	1,819,624	1,574,380	−10.2%
HQC-3	4,836,147	4,441,928	3,984,093	−10.3%
HQC-5	6,290,041	6,103,141	6,038,662	−1.1%

Our multiplication is faster than both prior implementations; relative to the faster of the two prior implementations, the multiplication cycles drop by 10.2% for HQC-1, 10.3% for HQC-3, and 1.1% for HQC-5. The improvement is largest for HQC-3, where the CRT decomposition with truncated basis conversion removes most of the forward-transform overhead, and smallest for HQC-5, whose multiplication is a pure double-size FAFFT with no residue ring and therefore benefits least from the construction of Section 3.1.

4 Fixed-Weight Sampling Optimization

HQC samples the fixed-weight sparse vectors $\mathbf{x}, \mathbf{y}$ from a secret seed and $\mathbf{r}_1, \mathbf{r}_2, \mathbf{e}$ as encryption randomness throughout key generation, encapsulation, and decapsulation. This sampling proceeds in two steps. First, `vect_generate_random_support1/2` extracts the support, the ω distinct positions, from an XOF. Then `vect_write_support_to_vector` (WSV) expands that support into a length-n bit vector. This section proposes an optimization that performs the latter scatter operation in constant time on the Cortex-M4.

4.1 Constant-Time Support Expansion

The WSV function is called after each `vect_generate_random_support1/2` to expand a fixed-weight vector's support, and is performed throughout the KEM operations. Concretely, it is called twice for $\mathbf{x}, \mathbf{y}$ during key generation, three times for $\mathbf{r}_1, \mathbf{r}_2, \mathbf{e}$ during encapsulation, and four times during decapsulation, the latter including the re-encryption for the FO transform in addition to recovering the secret vector $\mathbf{y}$.

The support produced by `vect_generate_random_support1/2` is the set of indices where the bit 1 appears in the secret vector. Hence a naive expansion that references each position p_j directly as a memory address index, as in $\mathbf{v}[\,p_j \gg 6\,] \mid= 1 \ll (p_j \,\&\, 63)$, induces a secret-dependent memory access pattern and is exposed to timing and cache side-channel attacks. To prevent this, the official HQC implementation first converts the support into word indices and bit masks, and then scans over every combination of the number of output words $n_{64} = \lceil n/64 \rceil$ and the support weight ω without branching. The implementations of [CCC+26, JLK+25] also adopt this expansion scheme unchanged. Since the preprocessing runs only ω times, the expansion overhead is concentrated mostly in the inner double loop that iterates $n_{64} \times \omega$ times.

The number of conditional bit accumulations inside the double loop, per single expansion and based on the Hamming weight $\omega_r = \omega_e$ of $\mathbf{r}_1, \mathbf{r}_2, \mathbf{e}$, is $277 \times 75 = 20{,}775$ for HQC-1, $561 \times 114 = 63{,}954$ for HQC-3, and $901 \times 149 = 134{,}249$ for HQC-5. As a result, for the HQC-1 performance measured on the NUCLEO-L4R5ZI board, this expansion accounts for about 29.4% of the key-generation cycles, 28.2% of encapsulation, and 21.6% of decapsulation relative to the implementation of [CCC+26], as shown in Table 7.

Table 7: Proportion of the WSV operation in the total cycles for HQC-1, measured on NUCLEO-L4R5ZI relative to the implementation of [CCC+26].

Operation	Expansion cycles	Full operation	Proportion
Key generation	859,628	2,924,928	29.4%
Encapsulation	1,461,071	5,181,432	28.2%
Decapsulation	1,889,926	8,738,801	21.6%

Prior implementations first decompose each support element into a word index $idx_j = p_j \gg 6$ and a bit mask $bit_j = 1 \ll (p_j \,\&\, 63)$, and then, for each output word index i, scan over all j and accumulate using the following portable branchless mask technique:

$$mask = -\big(1 \oplus ((t \mid -t) \gg 31)\big), \quad t = i - idx_j, \qquad \mathbf{v}[i] \mid= bit_j \,\&\, mask. \tag{2}$$

This mask computation guarantees constant-time execution, but generating the mask and applying it at every comparison step inside the loop requires a total of eight integer instructions.

4.2 Optimization Method

We replace the mask computation of (2) with the predicated execution feature of the ARMv7E-M architecture, and process the output words in groups of four to reduce the per-word comparison and load costs. Because 4-way loop unrolling requires many registers, we address it by designing the core loop of the expansion as a dedicated assembly routine.

The pre-decomposition that converts each support position p_j into a word index $idx_j = p_j \gg 6$ and a bit mask $bit_j = 1 \ll (p_j \,\&\, 63)$ is, as before, computed once in a preprocessing step and reused across the inner loop. The complete optimized inner loop of the proposed expansion is given in Algorithm 3. Below we describe it from three perspectives in order, namely predicated scatter, 4-way unrolling, and register allocation.

Algorithm 3 Predicated 4-way scatter inner loop.

```
.Lloop:
    ldmia r12!, {r0, r1, lr}   @ idx, bit_lo, bit_hi  (ptr += 12)
    subs  r0, r3, r0           @ d = i - idx_j
    itt   eq                   @ if (d == 0)   : word i
    orreq r4, r4, r1
    orreq r5, r5, lr
    cmn   r0, #1
    itt   eq                   @ if (d+1 == 0) : word i+1
    orreq r6, r6, r1
    orreq r7, r7, lr
    cmn   r0, #2
    itt   eq                   @ if (d+2 == 0) : word i+2
    orreq r8, r8, r1
    orreq r9, r9, lr
    cmn   r0, #3
    itt   eq                   @ if (d+3 == 0) : word i+3
    orreq r10, r10, r1
    orreq r11, r11, lr
    cmp   r12, r2              @ loop until r12 reaches r2 (end ptr)
    bne   .Lloop
```

Predicated scatter. The key idea is to handle the conditional accumulation, which ORs the bit mask bit_j into the output vector $\mathbf{v}[i]$ only when $i = idx_j$, directly through the conditional instruction `orreq`, without going through a mask-computation step. Once the `subs` instruction computes $i - idx_j$ and sets the Z flag, the following `orreq` executes only at the moment $i = idx_j$ and leaves the value unchanged otherwise. Because this predicated execution introduces no data-dependent branch and each `orreq` costs the same whether or not its condition holds, the instruction flow and cycle count are fixed independently of the concrete value of idx_j, guaranteeing constant time.

4-way unrolling. Using only a single `subs` comparison, we update the four output words $i, i{+}1, i{+}2, i{+}3$ at once. We keep $d = i - idx_j$, the result of `subs`, and execute `cmn d, #1`, `cmn d, #2`, and `cmn d, #3` to check whether idx_j equals $i{+}1, i{+}2, i{+}3$ without any additional subtraction. This reduces the comparison to one per output word, and the load cost for the upper and lower 32-bit halves of idx_j and bit_j, previously incurred three times per support element, is spread evenly across the four words down to 0.75 per word. Meanwhile, since each 64-bit word is accumulated in `hi/lo` halves to fit the Cortex-M4's

32-bit register width, processing four words uses eight registers as accumulators, `r4`–`r11` in Algorithm 3. For HQC parameter sets in which n_{64} is not a multiple of four, the words remaining after the loop, at most three, are finished off by a scalar tail loop functionally identical to (2).

Register allocation. The eight accumulators, the two upper/lower halves of the loaded bit mask, the temporary result d, and the streaming pointer to the packed support table together with the end pointer that marks loop termination must all stay resident in registers. The number of values that must be simultaneously live thus reaches thirteen, whereas the Cortex-M4 provides only fourteen general-purpose registers. At the `-O3` level the compiler cannot allocate these together with the registers needed for control operations such as the loop counter and memory base-address computation, causing a bottleneck. To resolve this limitation, we implement the core expansion loop of WSV as a hand-written dedicated assembly routine and allocate all fourteen available general-purpose registers of the Cortex-M4 directly. The concrete techniques are as follows. First, we design a packed table that interleaves each support element as $\{idx, bit_{\mathrm{lo}}, bit_{\mathrm{hi}}\}$. A single memory pointer and one `ldmia` instruction then handle the data load, removing the overhead of multiple pointers occupying registers. Second, temporaries that stay fixed within the loop body or are rarely used, such as the start address of the output buffer, are temporarily held on the stack to free up register space.

Constant-time guarantee. In addition to the branchless predication above, the loop bounds n_{64} and ω depend only on public system parameters, and the memory access is data-independent, loading the idx_j and bit_j data and writing $\mathbf{v}$ in a fixed sequential order. Thus the total execution time is independent of the input support.

4.3 Results

Measuring the expansion on the NUCLEO-L4R5ZI board under the same conditions as Table 7, the proposed predicated 4-way expansion reduces the expansion cycles by about 73.5% to 73.7% over the prior implementation, as shown in Table 8. In detail, for HQC-1 it reduces key generation from 859,628 to 227,379 cycles, encapsulation from 1,461,071 to 384,121 cycles, and decapsulation from 1,889,926 to 496,851 cycles, corresponding to speedups of about 3.78×, 3.80×, and 3.80×, respectively. This stems from replacing the prior eight mask instructions with one `subs`/`cmn` and two `orreq` conditional operations per word, and from loading each support element once with `ldmia` and reusing it across the four words. At the instruction level as well, the prior innermost loop takes about 22 cycles per support-element and output-word pair including the loads and mask computation, whereas the proposed method processes four words at once and lowers this to about 6 cycles per word. A KEM-level evaluation combining this method with the other optimizations is presented in Section 6.

Table 8: WSV cycle comparison for HQC-1.

Operation	[CCC+26]	Ours	Reduction
keypair	859,628	227,379	73.5%
encaps	1,461,071	384,121	73.7%
decaps	1,889,926	496,851	73.7%

5 Caching Strategies

Across repeated invocations under the same key, HQC's encapsulation and decapsulation recompute the same public values from scratch on every call. The public polynomial $\mathbf{h}$ is regenerated from $seed_{\mathsf{ek}}$ via SHAKE at every operation, and the FAFFT forward transforms of $\mathbf{h}$ and $\mathbf{s}$ used in multiplication, together with the public-key hash, are recomputed each time. In a setting that repeats many operations under one fixed key, this recomputation is duplicated between calls. Chen et al. [CCPY26] proposed a memoization technique that stores and reuses such transformed keys for HQC in a SIMD setting. This section ports that idea to the Cortex-M4 and proposes an optional caching that computes these public values once, stores them in a single cache identified by the public key, and reuses them on subsequent operations. The cache resides entirely in local RAM and leaves the transmitted public-key size unchanged, so as a time–space tradeoff its use can be decided at build time.

5.1 Applicability and Scenarios

The benefit of caching is realized when operations repeat under the same key, and the scenario in which this repetition is structurally guaranteed is decapsulation. A device that many peers connect to using the device's fixed public key, such as a Cortex-M4-class access-control reader, fare gate, or payment terminal, repeats decapsulation with its own fixed key on every connection. On such a device the cache is filled on the first call and reused by all subsequent decapsulation calls, and because the key belongs to the device itself the validity of reuse is structurally guaranteed. In contrast, encapsulation caching is confined to specific architectures where operations repeat for the same recipient's public key, and in ordinary session-based communication encapsulation runs only once per session, so cache reuse is hard to expect. Key generation is a one-time operation and is unaffected by caching. Under these scenarios, the measurements in this section assume a steady state in which the same public key is continuously reused.

5.2 Caching Targets and Mechanism

As explained in Section 2, $\mathbf{h}$ is a public polynomial generated deterministically from $seed_{\mathsf{ek}}$. Both encapsulation and decapsulation perform the multiplications $\mathbf{u} = \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$ and $\mathbf{s} \cdot \mathbf{r}_2$ when forming the ciphertext, and decapsulation recomputes them in the re-encryption of the Fujisaki–Okamoto (FO) transform. The forward transforms of $\mathbf{h}$ and $\mathbf{s}$ are therefore needed in both operations. In addition, both operations compute the public-key hash $\mathtt{H}(\mathsf{ek})$, where ek is the KEM encapsulation key, to derive the shared secret and the encryption seed. This hash and the two forward transforms all depend only on the fixed public key, so they recur identically across operations.

The cached public values are the following four, and only the form of the transform varies with the parameter set.

- **Public key** ek: used as the key that identifies the cache.
- **Public-key hash** $\mathtt{H}(\mathsf{ek})$: used as the hash input for deriving the shared secret and the encryption seed in encapsulation and decapsulation.
- **Transform of h**: all three parameter sets store the forward transform $\mathsf{FAFFT}(\mathbf{h})$. HQC-1 and HQC-3 additionally store the low-order coefficients of $\mathbf{h}$ needed for the auxiliary CRT multiplication.
- **Transform of s**: all three parameter sets store the forward transform $\mathsf{FAFFT}(\mathbf{s})$. Since $\mathbf{s}$ is carried directly in the public key, its low-order coefficients are not stored separately.

All of these items are already externally exposed public information, so caching and reusing them does not compromise security, whereas caching secret-dependent data would risk side-channel leakage. They are simply public values regenerated and transformed from the seed on each call, now kept resident in RAM, trading memory for speed.

The core of the proposed structure is that the `hash_h` function performing the public-key hashing is designed as the single decision point of the cache. Since both encapsulation and decapsulation compute this hash in common before the polynomial multiplication, at that step the incoming public key $\mathtt{ek}'$ is compared against the `ek` stored in the cache by a constant-time comparison over the full data length. On a cache hit, where the entry's validity flag is set ($valid = 1$) and the two keys match, the stored hash result is returned and the existing transform data is judged valid. On a cache miss, where they do not match, the new public key and hash value are written to the cache and the internal polynomial transforms are marked as an empty, not-yet-updated state. When the first subsequent multiplication then detects that the transform cache is empty, the polynomial $\mathbf{h}$ is regenerated from the public-key seed and forward-transformed, $\mathbf{s}$ is also forward-transformed, the cache is filled, and the valid bit is set. While the valid bit is set, the multiplication uses the cached transforms of $\mathbf{h}$ and $\mathbf{s}$ by direct reference without any additional condition check.

The proposed caching mechanism preserves the constant-time property of the entire operation. The public-key comparison accumulates bitwise results over the full key length without any conditional, and no secret-dependent branch or memory access occurs during cache lookup or data loading. We confirmed that the code with caching enabled passes the official KAT vectors. The speed and memory effects of the caching are evaluated in Section 6.

6 Evaluation

This section evaluates the speed performance and memory overhead of the proposed HQC optimizations on a Cortex-M4, and compares them with the two prior state-of-the-art implementations, [CCC+26] and [JLK+25]. All measurements were taken on an STM32 NUCLEO-L4R5ZI development board, which carries an ARM Cortex-M4 core with 2 MB of Flash and 640 KB of SRAM, using the pqm4 benchmarking framework [KPR+]. The CPU clock was set to 20 MHz so that measurements ran with zero wait states. To ensure a fair comparison with the prior implementations, we used the same arm-none-eabi-gcc 10.3.1 compiler with the `-O3` option.

6.1 Speed Performance

Table 9 compares the cycle counts of the HQC with our optimizations against the prior state-of-the-art implementations. Here the proposed method is the optimization without caching, incorporating the FFT butterfly register-scheduling optimization of Section 3 and the 4-way-unrolled support-expansion method of Section 4.

Relative to the fastest prior implementation for each operation, the proposed method reduces cycles by 27.7% to 33.1% for Keypair, 27.6% to 34.6% for Encaps, and 22.0% to 29.8% for Decaps. The additional reduction from the optional caching is treated separately in Section 6.2.

6.2 Evaluation of the Caching Strategy

In addition to the algorithmic optimizations described above, to isolate the standalone benefit of the caching proposed in Section 5, we compare the cycle counts of HQC with

Table 9: Cycle counts of HQC implementations on Cortex-M4.

Parameter	Operation	[CCC+26]	[JLK+25]	Ours	Reduction
HQC-1	Keypair	2,924,928	2,993,695	2,113,726	−27.7%
	Encaps	5,181,432	5,798,993	3,753,019	−27.6%
	Decaps	8,738,801	9,959,877	6,815,280	−22.0%
HQC-3	Keypair	8,053,779	7,659,978	5,262,127	−31.3%
	Encaps	14,425,876	14,667,541	9,428,668	−34.6%
	Decaps	22,872,448	23,352,058	16,053,394	−29.8%
HQC-5	Keypair	12,701,978	12,513,535	8,369,531	−33.1%
	Encaps	21,890,091	23,735,473	14,510,838	−33.7%
	Decaps	34,769,358	38,191,910	25,097,439	−27.8%

and without caching under the repeated-same-key scenario. The memory effect of caching is discussed together with the overall memory usage in Section 6.3.

As shown in Table 10, in the steady state where the same public key is reused, caching further improves performance. Encapsulation cycles decrease by about 30.3%, 26.0%, and 32.7% for HQC-1, HQC-3, and HQC-5 relative to the case without caching, and decapsulation by about 16.7%, 15.3%, and 18.9%. The absolute cycle savings of the two operations are similar because caching removes the common fixed cost of regenerating the public polynomial $\mathbf{h}$, forward-transforming $\mathbf{h}$ and $\mathbf{s}$, and computing the public-key hash $\mathtt{H}(\mathsf{ek})$. Key generation is not a caching target, so only encapsulation and decapsulation are compared.

Table 10: Cycle counts of the proposed method with and without caching.

Parameter	Operation	No caching	Caching	Reduction
HQC-1	Encaps	3,753,019	2,616,820	−30.3%
	Decaps	6,815,280	5,679,147	−16.7%
HQC-3	Encaps	9,428,668	6,976,592	−26.0%
	Decaps	16,053,394	13,601,480	−15.3%
HQC-5	Encaps	14,510,838	9,769,663	−32.7%
	Decaps	25,097,439	20,356,262	−18.9%

6.3 Memory Usage

Table 11 reports the Flash code size, static RAM, and dynamic stack usage of the proposed method, with and without caching. The proposed method builds on the FAFFT–CRT multiplication of [JLK+25] and additionally adopts the RS/RM decoder of [CCC+26] to accelerate the decoding in decapsulation.

The RAM usage of the proposed method stems from the FAFFT–CRT multiplication path it shares with [JLK+25]. For HQC-1 and HQC-3, which use CRT, the RAM of the proposed method is 14,344 and 37,504 bytes, larger than the implementation of [CCC+26] but equal to or even smaller than that of [JLK+25], which shares the same multiplication. This difference stems from the choice of CRT residue multiplication. HQC-3 uses the radix-16 residue multiplication of [JLK+25], so its RAM matches [JLK+25], whereas HQC-1 adopts the radix-16 residue multiplication of [CCC+26] and is 2,976 bytes smaller than [JLK+25]. For HQC-5, which does not use CRT, all three implementations share the same RAM of 2,824 bytes.

The Flash increase comes from the RS/RM decoder of [CCC+26] adopted to accelerate

decapsulation. This decoder unrolls the radix-16 matrix–vector products to gain speed at the cost of larger code, so relative to [JLK+25] the Flash of the proposed method is larger by 33,736, 35,168, and 162,912 bytes for HQC-1, HQC-3, and HQC-5. The increase is most pronounced for HQC-5 because HQC-5 uses a larger RS code, which enlarges the unrolled matrix–vector products. Compared with the implementation of [CCC+26], which uses the same decoder, the Flash of the proposed method is instead 6,208 bytes smaller for HQC-5. They fit comfortably within the 2 MB Flash and 640 KB SRAM of the target board, and the stack usage stays within a range similar to the two prior implementations.

Caching trades this speedup for additional static memory. As the Ours (caching) rows of Table 11 show, caching increases RAM by 10,792, 21,704, and 40,040 bytes for HQC-1, HQC-3, and HQC-5 due to the static buffer that keeps the transforms resident. Since these transforms were otherwise built on the stack at every call, the peak stack requirement instead decreases by 10,824, 21,660, and 29,944 bytes. That is, caching essentially moves the transforms from the stack to the static region, so the total of static RAM and peak stack is nearly unchanged for HQC-1 and HQC-3 and only slightly higher for HQC-5. Unlike the stack, however, which is occupied only during an operation and released afterward, the cached transforms stay resident in RAM at all times, so in exchange for the lower peak stack they keep occupying that much memory even between KEM operations. Caching is therefore a technique that pays a permanently resident transform cache to gain speed.

Table 11: Memory usage of HQC implementations on Cortex-M4, with and without caching (in bytes).

Parameter	Implementation	Flash (code+data)	RAM (data+bss)	Stack		
				Keypair	Encaps	Decaps
HQC-1	[CCC+26]	268,764	2,824	42,544	52,864	59,664
	[JLK+25]	294,972	17,320	43,752	49,944	56,744
	Ours	328,708	14,344	45,280	55,432	62,232
	Ours (caching)	329,796	25,136	–	36,848	51,408
HQC-3	[CCC+26]	305,652	2,824	84,512	105,736	119,352
	[JLK+25]	340,868	37,504	86,440	99,440	113,056
	Ours	376,036	37,504	86,432	107,496	121,112
	Ours (caching)	377,188	59,208	–	70,248	99,452
HQC-5	[CCC+26]	415,556	2,824	111,704	140,976	162,760
	[JLK+25]	246,436	2,824	111,696	132,864	154,648
	Ours	409,348	2,824	111,688	140,960	162,744
	Ours (caching)	410,052	42,864	–	93,768	132,800

7 Conclusion and Future Work

In this paper, we present optimizations that accelerate HQC on the ARM Cortex-M4. We find a 34% sparser FAFFT modulus for HQC-1, and trim the VMOV count of the bit-sliced butterflies by 37.6–48.1% at an unchanged EOR count via dirty-aware register allocation and XOR reordering. Predicated writes replace the masked support expansion of fixed-weight sampling, lowering its per-word cost from 22 to 6 cycles while remaining constant-time. Together, these reduce all three KEM operations by 22.0–34.6% over the state of the art, and the optional public-key caching cuts encapsulation and decapsulation by up to a further 32.7% and 18.9% under a reused key.

Our VMOV optimizations are not specific to HQC. Work on the linear layers of block ciphers, including [XZL+20], minimizes only the XOR count, yet on small in-order cores

every spill costs an explicit move that this metric does not capture. Our method applies as a post-processing step to any such XOR-count-minimized implementation, such as the AES MixColumns.

Limitations and future work. Optimizing a fixed XOR sequence can miss globally cheaper code, as a slightly longer sequence may admit fewer register transfers; jointly searching over the sequence and the allocation is a natural next step. The optional caching benefits only deployments that repeatedly decapsulate under a fixed key, and keeps its transform cache permanently resident in RAM.

References

[AMAB+23] C. Aguilar Melchor, N. Aragon, S. Bettaieb, L. Bidoux, O. Blazy, J. Bos, J.-C. Deneuville, A. Dion, P. Gaborit, J. Lacan, E. Persichetti, J.-M. Robert, N. Sendrier, P. Véron, and G. Zémor. HQC: Hamming Quasi-Cyclic — Round 4 Specification. https://pqc-hqc.org/, 2023. Specification version 2023-04-30.

[BGTZ08] Richard P Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann. Faster multiplication in gf (2)[x]. In *Algorithmic Number Theory: 8th International Symposium, ANTS-VIII Banff, Canada, May 17-22, 2008 Proceedings 8*, pages 153–166. Springer, 2008.

[BGTZ23] R. P. Brent, P. Gaudry, E. Thomé, and P. Zimmermann. gf2x library. https://gitlab.inria.fr/gf2x/gf2x, 2023. Accessed: Feb. 11, 2023. [Online].

[Can89] David G Cantor. On arithmetical algorithms over finite fields. *Journal of Combinatorial Theory, Series A*, 50(2):285–300, 1989.

[CCC+26] Ming-Shing Chen, Tun-You Chien, Chun-Ming Chiu, Han-Hsuan Lin, Chun-Tao Peng, and Bo-Yin Yang. Additive FFTs for HQC on ARM Cortex-M4, revisited. *IACR TCHES*, 2026(3), 2026.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing bike for the intel haswell and arm cortex-m4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):97–124, Jul. 2021.

[CCPY26] Ming-Shing Chen, Chun-Ming Chiu, Chun-Tao Peng, and Bo-Yin Yang. Accelerating HQC with additive FFT. *IACR TCHES*, 2026(2):520–544, 2026.

[CGKT22] Ming-Shing Chen, Tim Güneysu, Markus Krausz, and Jan Philipp Thoma. Carry-less to BIKE faster. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 22International Conference on Applied Cryptography and Network Security*, volume 13269 of *LNCS*, pages 833–852. Springer, Cham, June 2022.

[CGLD13] Danilo Câmara, Conrado PL Gouvêa, Julio López, and Ricardo Dahab. Fast software polynomial multiplication on ARM processors using the neon engine. In *International Conference on Availability, Reliability, and Security*, pages 137–154. Springer, 2013.

[DGK20] Nir Drucker, Shay Gueron, and Dusan Kostic. Fast polynomial inversion for post quantum QC-MDPC cryptography. In *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 110–127. Springer, 2020.

[GM10] Shuhong Gao and Todd Mateer. Additive fast fourier transforms over finite fields. *IEEE Transactions on Information Theory*, 56(12):6265–6272, 2010.

[JLK+25] Jihoon Jang, Myeonghoon Lee, Donggeun Kwon, Seokhie Hong, Suhri Kim, and Sangjin Lee. Efficient polynomial multiplication for HQC on ARM cortex-m4. Cryptology ePrint Archive, Paper 2025/1939, 2025.

[JLK+26] Jihoon Jang, Myeonghoon Lee, Suhri Kim, Seog Chung Seo, and Seokhie Hong. Faster gf (2)[x] polynomial multiplication using simd instructions. *Journal of Cryptographic Engineering*, 16(1):3, 2026.

[KGJV83] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[KPR+] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[LANH16] Sian-Jheng Lin, Tareq Y Al-Naffouri, and Yunghsiang S Han. FFT algorithm for binary extension finite fields and its application to reed–solomon codes. *IEEE Transactions on Information Theory*, 62(10):5343–5358, 2016.

[LCK+18] Wen-Ding Li, Ming-Shing Chen, Po-Chun Kuo, Chen-Mou Cheng, and Bo-Yin Yang. Frobenius additive fast fourier transform. In *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, pages 263–270, 2018.

[LJKH25] Myeonghoon Lee, Jihoon Jang, Suhri Kim, and Seokhie Hong. Improved frobenius fft for code-based cryptography on cortex-m4. *IEEE Internet of Things Journal*, 12(17):36318–36327, 2025.

[Tea25] NIST PQC Team. Status Report on the Fourth Round of the NIST Post-Quantum Cryptography Standardization Process. NIST IR 8545, 2025.

[VzGG96] Joachim Von zur Gathen and Jürgen Gerhard. Arithmetic and factorization of polynomial over F2. In *Proceedings of the 1996 international symposium on Symbolic and algebraic computation*, pages 1–9, 1996.

[XZL+20] Zejun Xiang, Xiangyoung Zeng, Da Lin, Zhenzhen Bao, and Shasha Zhang. Optimizing implementations of linear layers. *IACR Transactions on Symmetric Cryptology*, 2020(2):120–145, Jul. 2020.