



Optimizing Crystals-Dilithium implementation in 16-bit MSP430 environment utilizing hardware multiplier

DongHyun Shin, YoungBeom Kim, Seog Chung Seo*

Department of Financial Information Security, Kookmin University, Seoul, South Korea

Received 9 May 2024; received in revised form 7 August 2024; accepted 28 September 2024

Available online 5 October 2024

Abstract

Dilithium was selected as one of NIST standard Post Quantum Digital Signature algorithms and is undergoing standardization as a Module Lattice Digital Signature Algorithm (ML-DSA). However, until now research on optimization in embedded environments has primarily been conducted on ARM architectures, which are the basic benchmark targets. To prepare for future quantum secure Internet of Things environments, performance optimization on resource-constrained must be considered. Thus, in this paper, for the first time, we propose an optimized implementation of Dilithium in the 16-bit MSP430 environment, a low-resource device. We redesign the state-of-the-art optimization strategies for Dilithium to suit the MSP430 environment. By taking full advantage of MSP430's hardware multiplier in the NTT-based polynomial multiplication, we achieve 73.0% and 80.1% of performance improvement for NTT and NTT^{-1} compared to those in the reference implementation, which contributes about 5.5%–7.0%, 15.3%–17.5%, and 7.5%–10.0% of performance improvement compared to Dilithium's public reference implementation for keypair generation, signing, and verification, respectively.

© 2024 The Authors. Published by Elsevier B.V. on behalf of The Korean Institute of Communications and Information Sciences. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

Keywords: Digital signature algorithm; Post-quantum cryptography; Crystals-Dilithium; MSP430

1. Introduction

It has been widely known that the security of currently used public-key based key establishment mechanisms and digital signature algorithms will be broken as the development of quantum computers that can execute Shor's algorithm. For preparing secure IT environment in this quantum computing era, the National Institute of Standards and Technology (NIST) has organized the Post-quantum cryptography (PQC) competition for secure Key Encapsulation Mechanism (KEM) and Digital Signature Algorithm (DSA) and continues efforts towards standardization. Dilithium was selected by NIST as one of DSAs and is undergoing standardization as ML-DSA [1].

To apply PQCs including Dilithium smoothly for various IT services, the performance optimization is required because they typically require more computational overhead compared to conventional public key algorithms. Particularly, antenna

systems are crucial for IT systems in transmitting and receiving signals, and the computational overhead of PQC can directly impact these communications [2]. In other words, performance optimization is more important on embedded devices since they have only limited computing power and memory capacity compared to typical x86 CPUs. However, to date, most of the research on software optimization in embedded devices has been conducted on 32-bit ARM MCUs (Cortex-M3/M4) [3–5] and 64-bit ARM MCUs (ARMv8) [6,7]. To achieve quantum security in future Internet of Things environment composed of various heterogeneous IT devices including sensors, CCTVs, automotive devices and so on., optimization research needs to be conducted on these resource constrained devices.

Thus, in this paper, we propose an efficient implementation of Crystals-Dilithium on 16-bit MSP430 which is a representative MCU for wireless sensor devices. For performance efficiency, we redesign the state-of-the-art optimization strategies for Dilithium to suit the MSP430 environment. In detail, we analyze the characteristics of the MSP430 architecture (especially, hardware multiplier) and design the implementation of layer merging in the process of Number Theoretic Transform (NTT)-based polynomial multiplication

* Corresponding author.

E-mail addresses: dkldkl13@kookmin.ac.kr (D. Shin),

darania@kookmin.ac.kr (Y. Kim), sseo@kookmin.ac.kr (S.C. Seo).

Peer review under responsibility of The Korean Institute of Communications and Information Sciences (KICS).

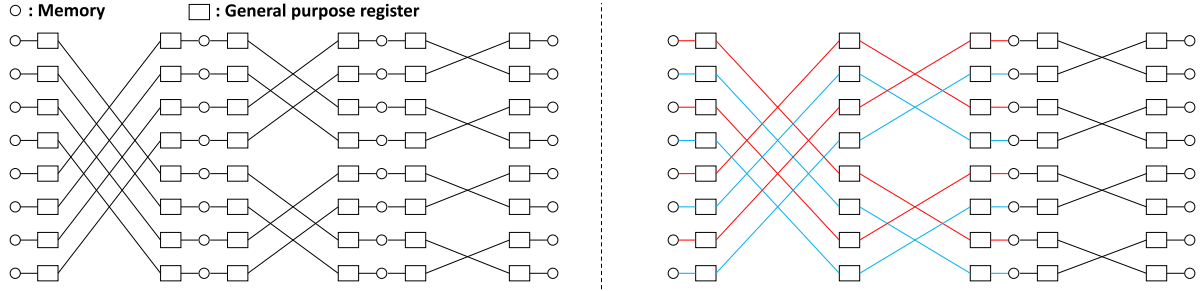


Fig. 1. Computing NTT on $\mathbb{Z}_q[X]/(X^8 + 1)$, (left) computing for every single layer, (right) implementing the initial 2 layers merge.

by appropriately utilizing the multiplier. Furthermore, we significantly reduce the number of shift operations in modulus calculations by leveraging the 16-bit features. As a result, we achieve performance improvements of 73.0% and 80.1% for a NTT and NTT^{-1} , respectively, compared to the public reference implementations of Dilithium. Furthermore, in the signature generation process, we implement a 16-bit NTT using a small q' which is around 416.5% faster than typical 32-bit NTT used in the reference implementation of Dilithium. With the proposed implementation techniques, our software provides about 5.5%–7.0%, 15.3%–17.5%, and 7.5%–10.0% of performance improvement compared to the public Dilithium reference implementation for keypair generation, signing, and verification, respectively. As far as we know, this is the first work of efficient implementation of NIST PQC algorithm on resource-constrained MCUs.

2. Related works

2.1. Previous works on optimizing NTT-based polynomial multiplication in Dilithium

Dilithium [8] is in the process of being standardized as ML-DSA [1], having been selected as a finalist in 2022. Dilithium is designed based on the Module learning-with-errors (MLWE) problem. Similar to other lattice-based algorithms, Dilithium's main computational load is polynomial multiplication. Polynomial multiplication in Dilithium is performed on the polynomial ring $\mathbb{Z}_q[X]/(X^{256} + 1)$, which can be efficiently implemented via the Number Theoretic Transform (NTT)-based multiplication.

NTT is a special case of FFT on the integer domain. It consists of a Forward Transform (NTT), point-wise multiplication, and Inverse Transform (NTT^{-1}) on the integer domain. In particular, the partitioning of NTT and NTT^{-1} on polynomial rings uses Cooley–Tukey (CT) and Gentleman–Sande (GS), the same approach as in FFT. In Kyber, recent work [4] uses CT for NTT^{-1} implementations due to the range of coefficients, but GS is sufficient for NTT^{-1} on Dilithium with 23-bit q .

[9] provided profiling results for Dilithium on Cortex-M4, with Keccak and NTT-based polynomial multiplication accounting for the most operations. For the Cortex-M4, a 32×32 multiplication takes 1 cycle. However, the MSP430 performs at 12 cycles, even with multiplier, so the percentage of NTT increases. Also, unlike the M4, the MSP430 does not

have the barrel shifter, so there is room for optimization in reduction methods.

Also, there has been research on using a smaller q' in Dilithium's signature generation process, taking advantage of the fact that cs_1 and cs_2 are very small values [4]. Since the original q is 23-bit, we typically use a 32-bit NTT. However, using a 16-bit or smaller q' allows us to use a 16-bit NTT, which is more effective on the MSP430.

2.2. Target platform: MSP430

The MSP430 has a total of 12 (r4–r15) general-purpose registers of 16-bit size. It has 16×16 and 32×32 multipliers, which can be used via the mov instruction. For example, the 32×32 multiplier is used as follows registers dedicated to $\text{MPY32}\{\text{L}, \text{H}\}$, $\text{OP2}\{\text{L}, \text{H}\}$, and $\text{RES}\{0, 1, 2, 3\}$ exist. $\text{MPY32}\{\text{L}, \text{H}\}$, $\text{OP2}\{\text{L}, \text{H}\}$ are spaces to store the low 16-bit and high 16-bit of operand1(MPY32) and the low 16-bit and high 16-bit of operand2(OP2), respectively. Each multiplication is performed as soon as the value is written to the $\text{OP2}\{\text{L}, \text{H}\}$ register, and $\text{RES}\{0, 1, 2, 3\}$ register stores the 64-bit result value. Furthermore, by utilizing the MAC register instead of the MPY register, it becomes possible to accumulate the multiplication result into the RES register.

We choose a device with more than 8 kB of RAM to run Dilithium. We have selected the MSP430F6779 microcontroller as our target board, which operates at 25 MHz and is equipped with 32 kB of RAM.

3. Proposed optimization approaches for general NTT/ NTT^{-1} in Dilithium

The main computational overhead part in Dilithium is the NTT-based multiplication. We take full advantage of the MSP430 architecture features and multiplier to accelerate performance. Since the q of Dilithium is 23-bit, we use a 32×32 multiplier to compute the 32-bit NTT.

3.1. NTT and NTT^{-1}

Dilithium computes 8 layers with a complete NTT, dividing a polynomial of degree 255 into 256 monomials. Generally, computing one layer requires 256 memory accesses to these coefficients. We redesign the layer merging technique originally proposed for the Cortex-M4 from [4] to fit the MSP430

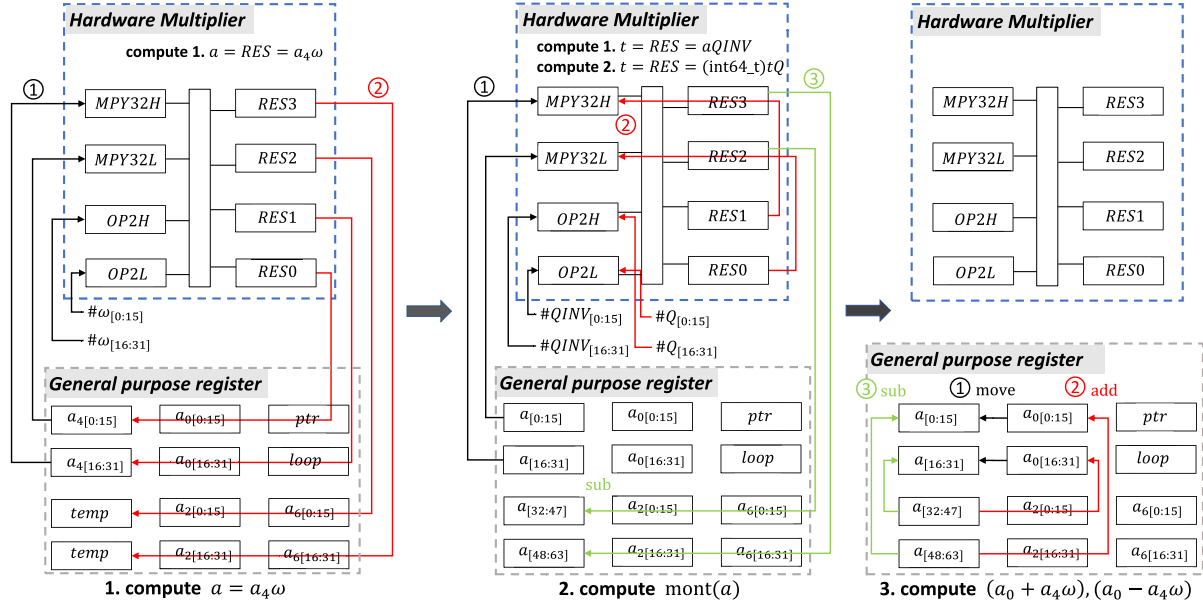


Fig. 2. Our implementation process for CT butterfly. Process compute CT butterfly with merged 2 layer: $(a_0 + a_4\omega) + (a_0 - a_4\omega)$. In here, ω is twiddle factor of NTT.

device, in order to reduce memory accesses. Fig. 1 shows an example of the merging technique. The left side is a single layer on $\mathbb{Z}_q[X]/\langle X^8 + 1 \rangle$, and the right side is a merged initial 2-layer. In Fig. 1, each plot is butterfly way for NTT. Circles, showed in Fig. 1, are memory access to coefficients of polynomial. A general implementation computes all layers sequentially, which requires performing load and store operations at every layer. In a 2-layer merged implementation, we load the coefficients related to both layers, compute all the butterflies for the two layers, and then store them in memory. This reduces memory access overhead.

To merge n layers, we maintain 2^n coefficients which are butterfly input data of n layers. In the Cortex-M4 environment, research was conducted to merge 3 layers using Floating-point Unit(FPU) registers [4]. Since FPU registers of Cortex-M4 incur fewer access overheads compared to memory accesses, they can be used as a cache for merging technique. In contrast, the MSP430 does not have a FPU but includes dedicated registers for multiplier. However, since the multiplier is used in butterflies, it should not be used as a cache like the FPU on the Cortex-M4. Since MSP430 has only 12 general-purpose registers, for using merging technique, we need careful observation and efficient register-scheduling.

Since the q of Dilithium is 23-bit, two registers are required to load one coefficient. To merge two layers, we need to hold 4 coefficients, which requires 8 registers. Additionally, to prevent values from growing too large during the butterfly operations, we use Montgomery reduction, which requires 4 registers. Thus, merging two layers requires a total of 12 registers. This approach is inefficient because it involves repeatedly pushing and popping a polynomial pointer to and from memory. However, we propose a method to perform the computations using 10 registers by carefully observing

the butterfly process and utilizing the hardware multiplier efficiently. The remaining 2 registers are used for a polynomial pointer and a loop counter. Merging 3 layers would require 8 coefficients, which means at least 16 registers would be needed, making it inefficient.

Fig. 2 shows the implementation of merging 2 layers using only 10 registers. To compute the first butterfly, we move a_4 and ω to the multiplier's operand registers ω is handled using an immediate instruction. Once the value is moved to the OP2 register, the multiplication is performed immediately. The multiplication result is then moved to the temporary register and the existing a_4 register. Since a_4 is not used once it is multiplied by ω , it does not need to be preserved. After performing the Montgomery reduction and addition/subtraction, the butterfly computation is complete. For detailed information on Montgomery reduction, please refer to Section 3.2. This process is repeated 4 times to compute the butterflies for the 2 layers and store the coefficients. Our implementation employs CT-butterfly for NTT and GS-butterfly for NTT^{-1} . Even with GS-butterfly in NTT^{-1} , 2 layers can be merged in the same manner. Therefore, we can merge each layer by 7-6, 5-4, 3-2, and 1-0 layers.

3.2. Faster modular arithmetic on MSP430

In general, modulo q computations on 32-bit data in Dilithium use `reduce32`, a variant of the Barrett reduction, and reductions on 64-bit data, such as butterfly and point-wise multiplication in NTT, use the Montgomery reduction. For detailed information on the algorithm, please refer to [10]. These algorithms achieve constant-time implementation by replacing division with shift operations and provide efficient reduction methods. We further accelerate these algorithms by utilizing the 16-bit registers of the MSP430.

Algorithm 1 Faster reduce32

Input: int32_t a := $a_0 || a_1$
Output: int32_t t, with $t \equiv a \pmod{q}$

```

//r4, r5 := a0, a1
1: //t = (int32_t)(a + (1<<22)) >>23
2: mov r5, r6; add r6, #40;
3: rra r6; rra r6; rra r6; rra r6;
4: rra r6; rra r6; rra r6;
5: //t = (int32_t)(a - t * Q)
6: mov #q0, &MPYS32L; mov #q1, &MPYS32H;
7: mov r6, &OP2L; rla r6;
8: subc r6, r6; inv r6; mov r6, &OP2H;
9: sub &RES0, r4; subc &RES1, r5;

```

3.2.1. Faster reduce32

The `reduce32` contains 23 shift operations, since the MSP430 does not have the barrel shifter, it is a lot of computational overhead. Since polynomial coefficients must be stored in 2 registers, implementing 23 shifts simply would require a total of 46 `rra/rrc` instructions. However, we can achieve the same result by performing only seven shift operations on the upper 16-bit data. This means that we can omit the shift operations corresponding to multiples of 16, as shown in Algorithm 1. As a result, we omit 39 shifts. The hardware multiplier performs the multiplication as soon as a value is loaded into the OP2 register. Therefore, in the `poly_reduce`, which calculates `reduce32` for all polynomial coefficients, we can skip the step of moving q to the multiplier for subsequent coefficients after the first one (Algorithm 1, line 6). The q value is already present in the MPY32 register.

3.2.2. Faster montgomery reduction

Like `reduce32`, it omits the shift operations. It uses a multiplier and is implemented using only the `mov` and `sub` instructions, completely eliminating the shift operation. The reduction for a polynomial, `poly_reduce32` and `poly_mont` are merged into one function at the assembly stage, effectively removing function call overhead.

3.3. Point-wise multiplication for NTT

After the NTT, the multiplication of two polynomials is performed in the form of a point-wise multiplication, which is a multiplication of 256 monomials. At the assembly level, the 256 multiplications and reductions are implemented in a loop, effectively removing the function call overhead.

4. Proposed optimization approaches for small NTT/NTT⁻¹ in Dilithium

In the signature generation process of Dilithium, the values cs_1 and cs_2 consist of very small values. In [11], research was conducted on applying NTT in NTT-unfriendly rings by using a q' larger than the maximum size of the polynomial coefficients. Following a similar approach, [4] explored the application of small NTT in Dilithium by selecting a smaller q' for cs_1 and cs_2 . Since the absolute values of the coefficients of

Algorithm 2 Faster 32-bit Montgomery reduction

Input: int64_t a := $a_0 || a_1 || a_2 || a_3$
Output: int32_t a, with $a \equiv a \cdot R \pmod{q}$

```

//r4...r7 := a0,...,a3
1: //t = (int32_t)((int64_t)a * QINV)
2: mov r4, &MPYS32L; mov r5, &MPYS32H;
3: mov #q0-1, &OP2L; mov #q1-1, &OP2H;
4: //t = (int32_t)(a - (int64_t)t * Q) >> 32
5: mov &RES0, &MPYS32L;
6: mov &RES1, &MPYS32H;
7: mov #q0, &OP2L; mov #q1, &OP2H;
8: //a = (int32_t)a - t
9: sub &RES2, r6; subc &RES3, r7;

```

cs_1 and cs_2 are smaller than $\tau\eta$, we can choose $q' > 2\tau\eta$ and perform the computation in $\mathbb{Z}_{q'}$. In a manner identical to [4], Dilithium2 and Dilithium5 use the Fermat prime q' as 257, while Dilithium3 uses q' as 769 and applies a small NTT. The small NTT using a Fermat prime is called the Fermat number transform (FNT). The original NTT required a 32-bit NTT computation, utilizing a 32×32 multiplier. However, since small NTT uses 16-bit NTT for both 257 and 769, a 16×16 multiplier can be used, reducing the computational cost by half.

4.1. Small NTT and NTT⁻¹

We used 32-bit NTT because the original q was 23-bit, but in the small NTT, $q' = 9$ -bit (or 10-bit), so we use 16-bit NTT. Using the 16-bit NTT effectively reduces the computational overhead. Also, the 16-bit NTT requires fewer registers to hold the coefficients, so unlike the general NTT, this can be implemented by merging the 3 layers.

Since Dilithium 2 and Dilithium5 are FNT, they replace the multiplication in the initial layer with a shift operation. By detailed range analysis, the butterfly in the initial 2.5 layers skips the reduction, because c and s are generated with very small values. As a result, we implement 7-5, 4-3, and 2-1 layers by merging them. NTT⁻¹ is computed in reverse with the same logic as NTT.

4.2. Faster reduction for small NTT

Butterfly in small NTT, we use Montgomery reduction for 16-bit q' . Similar to Algorithm 2, we omit shift operations and implement using only `mov` and `sub`. Since the MSP430 performs computations in 16-bit units, using 16-bit Montgomery reduction enables us to perform the computations with only half the number of instructions required by Algorithm 2.

As a result, we choose the 16-bit Montgomery Reduction. After a small NTT⁻¹, we use the `reduce32` on q' to get back to $\mathbb{Z}_q$.

4.3. Pointwise multiplication for small NTT

Unlike general NTT, small NTT is an incomplete NTT. It performs 7 layers, and the point-wise multiplication after NTT

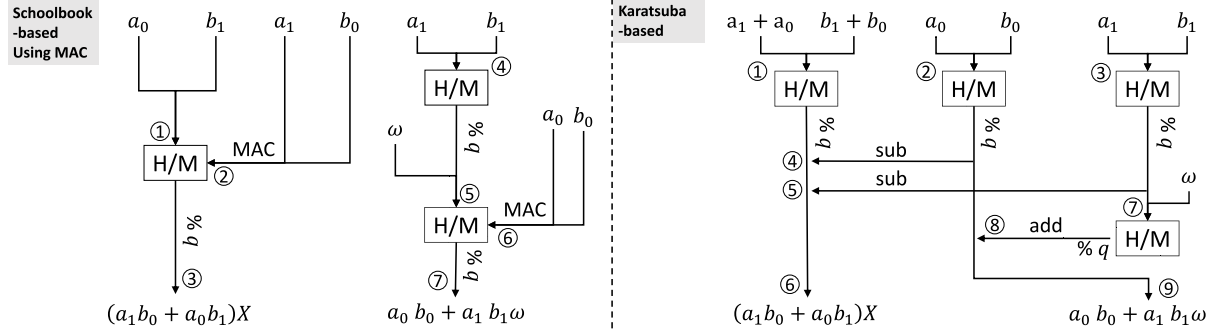


Fig. 3. Multiplication for degree one polynomial on $\mathbb{Z}_q[X]/\langle X^2 + \omega \rangle$, (left) schoolbook-based multiplication using MAC, (right) Karatsuba-based multiplication. In here, ω is twiddle factor of NTT.

is a multiplication for degree one polynomials. The simplest implementation for degree one polynomial is schoolbook-based multiplication. It consists of five multiplications, five modular reductions are also required to prevent the coefficients from continuing to grow.

The MSP430 computes add and sub in a single cycle. However, a 16×16 multiplication typically needs four mov, for a total of 12 cycles. Modular reduction also involves multiple multiplications, so the key to optimizing multiplication for degree one polynomial is to reduce the number of multiplications and modular reductions. Therefore, we can consider both schoolbook-based multiplication using accumulated reduction and Karatsuba multiplication, which reduces the number of multiplications.

Fig. 3 shows the multiplication process of a schoolbook-based implementation using the MAC instruction and a Karatsuba-based implementation. The schoolbook-based multiplication first calculates a_0b_1 . Then, by utilizing the MAC and skipping the step of moving the value from the RES register, it uses a_0b_1 as the input to compute $a_1b_0 + a_0b_1$. By using MAC as the operand register instead of MPY, we can multiply with accumulation. Then, the value of $a_1b_0 + a_0b_1$ is reduced modulo q in a single step. Since `poly_reduce` is performed after the NTT transformation, multiplying with accumulation followed by reduction does not result in overflow. Using the same method, we compute $a_0b_0 + a_1b_1\omega$ with MAC. Karatsuba-based multiplication computes $a_1b_0 + a_0b_1$ using the formula $(a_1 + a_0) \cdot (b_1 + b_0) - a_1b_1 - a_0b_0$. The values a_1b_1 and a_0b_0 are reused when computing $a_0b_0 + a_1b_1\omega$. Karatsuba-based multiplication involves reusing values, which limits the applicability of MAC. The Karatsuba-based multiplication needs 5 multiplications and four reductions. However, the school-based multiplication using MAC implementation needs only 5 multiplications and 3 reductions. In addition, two multiplications are accumulated, skipping the move to the general-purpose register. As a result, the schoolbook-based implementation using MAC saves 27 cycles compared to the Karatsuba-based implementation.

5. Results

5.1. Benchmarking setup

As mentioned in Section 2.2, we use IAR EW compiler with the highest optimization option (`high(speed)`). We

Table 1

Cycle counts (cc) for NTT, basemul(point-wise multiplication), NTT^{-1} , small NTT, small NTT^{-1} on the MSP430.

	Prime q	Ref. [12]	Ourwork
NTT	8 380 417	157 009	90 736
NTT^{-1}	8 380 417	205 407	114 026
basemul	8 380 417	32 262	21 076
FNT	257	–	30 399
FNT^{-1}	257	–	57 804
small NTT	769	–	40 152
small NTT^{-1}	769	–	57 204
basemul(FNT/small)	257/769	–	18 869

use the stack-optimized version of [12], implemented based on [13], as reference. Unfortunately there is no known Dilithium implementation on the MSP430 environment. We merge the assembly code, which incorporates our optimization methodologies, with the C code from [13] and compare the performance on an MSP430F6779. The performance is measured using the cycle counter register provided by IAR EW Workbench for MSP430. For detailed specifications of the board, please refer to Section 2.2.

5.2. Performance analysis

Table 1 shows the implementation results of NTT-based polynomial multiplications. The 32-bit NTT achieves approximately an 80% performance improvement over the reference C code by merging two layers and optimizing the modular reduction. The small NTT uses 16-bit NTT, and by merging three layers, it shows results more than twice as fast as the optimized 32-bit NTT. This is an expected outcome because the cost of 16×16 multiplication is half that of 32×32 multiplication. Furthermore, schoolbook-based multiplication using MAC is faster than Karatsuba-based multiplication and shows faster performance than 32-bit NTT's basemul.

Table 2 presents the results for the overall Dilithium scheme. Our implementation achieves performance improvements of 5.5%–7.0% in key generation, 15.3%–17.5% in signature generation, and 7.5%–10.0% in signature verification compared to the reference C code.

Table 2

Cycle counts (cc), second (s), and stack usage (B) for Keygen, Sign, and Verify of all security level of Dilithium on 16-bit MSP430. 1000 cc is denoted by k. When measuring the average over 10,000 runs as in [4], the MSP430 takes too long. Moreover, with only 100 runs, the abort variance for the Sign function is too large. Therefore, we find seeds with average abort counts and measure performance accordingly. The average abort counts for Dilithium2, 3 and 5 are 4, 5 and 4, respectively. Our implementation uses an additional stack of a polynomial size for small NTT.

Implementation		Reference [12]			Ourwork		
		cycles (k)	second (s)	stack (B)	cycles (k)	second (s)	stack (B)
Dilithium2	K	(–) 13 665k	0.547 s	13 236B	(+7.0%) 12 776k	0.511 s	13 236B
	S	(–) 98 613k	3.945 s	17 148B	(+17.5%) 83 892k	3.356 s	18 172B
	V	(–) 14 014k	0.561 s	16 622B	(+10.0%) 12 749k	0.510 s	16 622B
Dilithium3	K	(–) 24 925k	0.997 s	17 397B	(+5.6%) 23 593k	0.944 s	17 397B
	S	(–) 194 148k	7.766 s	21 540B	(+15.3%) 168 330k	6.733 s	22 564B
	V	(–) 24 733k	0.989 s	19 158B	(+8.1%) 22 870k	0.915 s	19 158B
Dilithium5	K	(–) 41 141k	1.646 s	21 005B	(+5.5%) 38 985k	1.559 s	21 005B
	S	(–) 273 668k	10.947 s	25 810B	(+16.7%) 234 416k	9.377 s	26 834B
	V	(–) 41 492k	1.660 s	23 458B	(+7.5%) 38 580k	1.543 s	23 458B

Dilithium's signature generation process, with an average of approximately 4 rejections, demonstrates the significant advantage of using small NTT. As a result, the Sign process achieves greater performance improvements compared to other processes. According to [14], profiling results for Dilithium on Cortex-M4 showed that hashing accounts for approximately 70% of the total computation time. Despite our implementation achieving over an 80% performance improvement in NTT-based polynomial multiplication, the overall performance improvement of up to 17% suggests that, similar to the Cortex-M4, hashing constitutes a significant computational overhead.

6. Conclusion

In this paper, we proposed for the first time an optimization strategy for Dilithium in the 16-bit MSP430 environment, a low-resource device. we have accelerated the performance of Dilithium by optimizing the NTT-related operations, which is the main computational overhead point of Dilithium. NTT-related operations achieve performance improvements of over 80%, while the overall scheme shows performance enhancements ranging from 5.5% to 17.5%. However, there are still issues with Keccak. To our knowledge, there are no Keccak implementations for the MSP430, including the eXtended Keccak Code Package (XKCP) provided by the Keccak team. Therefore, we believe that optimizing Keccak is necessary to further accelerate Dilithium on the MSP430.

CRedit authorship contribution statement

DongHyun Shin: Writing – original draft. **YoungBeom Kim:** Writing – review & editing, Conceptualization. **Seog Chung Seo:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) ((No. RS-2021-II210796, Research on Foundational Technologies for 6G Autonomous Security-by-Design to Guarantee Constant Quality of Security, 50%) and partly supported by IITP grant funded by MSIT (A study on PQC optimization and security protocol migration to neutralize advanced quantum attacks in Beyond 5G-based next-generation IoT computing environments, 50%) under Grant 2022-00207416).

References

- [1] National Institute of Standards and Technology (NIST), Module-Lattice-Based Key-Encapsulation Mechanism Standard. <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.ipd.pdf>.
- [2] M. Alibakhshikenari, B.S. Virdee, C.H. See, R.A. Abd-Alhameed, F. Falcone, E. Limiti, Super-wide impedance bandwidth planar antenna for microwave and millimeter-wave applications, *Sensors* 19 (10) (2019) 2306.
- [3] E. Alkim, Y.A. Bilgin, M. Cenk, F. Gérard, Cortex-M4 optimizations for {R, M} LWE schemes, *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2020) 336–357.
- [4] A. Abdulrahman, V. Hwang, M.J. Kannwischer, A. Sprenkels, Faster kyber and dilithium on the cortex-M4, in: *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings*, Springer, 2022, pp. 853–871.
- [5] J. Huang, J. Zhang, H. Zhao, Z. Liu, R.C. Cheung, Ç.K. Koç, D. Chen, Improved plantard arithmetic for lattice-based cryptography, *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022 (4) (2022) 614–636.
- [6] H. Becker, V. Hwang, M.J. Kannwischer, B.-Y. Yang, S.-Y. Yang, Neon ntt: Faster dilithium, kyber, and saber on cortex-a72 and apple m1, *Cryptol. ePrint Arch.* (2021).
- [7] Y. Kim, J. Song, T.-Y. Youn, S.C. Seo, et al., Crystals-dilithium on armv8, *Secur. Commun. Netw.* 2022 (2022).
- [8] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, D. Stehlé, Crystals-dilithium: A lattice-based digital signature scheme, *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2018) 238–268.
- [9] D.O.C. Greconici, M.J. Kannwischer, A. Sprenkels, Compact dilithium implementations on cortex-M3 and cortex-M4, *Cryptol. ePrint Arch.* (2020) <http://dx.doi.org/10.46586/tches.v2021.i1.1-24>, URL <https://eprint.iacr.org/2020/1278>. Paper 2020/1278.
- [10] G. Seiler, Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography, *Cryptol. ePrint Arch.* (2018).

- [11] C.-M.M. Chung, V. Hwang, M.J. Kannwischer, G. Seiler, C.-J. Shih, B.-Y. Yang, NTT multiplication for NTT-unfriendly rings: New speed records for saber and NTRU on cortex-M4 and AVX2, *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2021) 159–188.
- [12] V. Hwang, Y. Kim, S.C. Seo, Barrett multiplication for dilithium on embedded devices, *Cryptol. ePrint Arch.* (2023).
- [13] J.W. Bos, J. Renes, A. Sprenkels, Dilithium for memory constrained devices, in: *International Conference on Cryptology in Africa*, Springer, 2022, pp. 217–235.
- [14] M.J. Kannwischer, J. Rijneveld, P. Schwabe, K. Stoffelen, pqm4: Testing and benchmarking NIST PQC on ARM cortex-M4, *Cryptol. ePrint Arch.* (2019).