Efficient SIMD SHAKE through Cost-Aware Permutation Scheduling

Hang Zhang[1,2], Ya Shi[1,2], Ying Liu[1,2] and Xianhui Lu[1,2]

[1] Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS, Beijing, China
{zhanghang2024,shiya,liuying,luxianhui}@iie.ac.cn
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

Abstract. Lattice-based cryptography uses SHAKE to expand seeds into byte strings for matrix generation and sampling. SIMD Keccak implementations process multiple hash streams in parallel. Differences in stream lengths and input dependencies can prevent full SIMD utilization. We present a cost-aware method for scheduling Keccak permutations across hash streams during a single execution of a cryptographic algorithm. The method groups ready streams and divides long streams into segments for regrouping. A model of permutation calls, state movement, initialization, and input preparation guides the selection of stream groups, segment lengths, and state placements. We evaluate ML-KEM and HAETAE on an Apple M5, a Cortex-A72, and an Intel Core i5-11500, using Keccak kernels that process two, four, or eight streams per call. ML-KEM achieves speedups of up to 1.31× for encapsulation and 1.44× for decapsulation, while HAETAE achieves up to 1.28× for signing and 1.26× for verification.

Keywords: SHAKE · Pseudorandom generation · Keccak · SIMD · Permutation scheduling · ML-KEM · HAETAE

1 Introduction

SHAKE-based pseudorandom generation is a major computational task in lattice-based cryptography. Schemes expand seeds into byte strings for matrix generation, noise sampling, and signature sampling [Nat15, Nat24, CCD+24]. Their samplers convert these bytes into coefficients with the required distributions. Polynomial arithmetic has received extensive NTT and SIMD optimization [BHK+22, Sei18, SLL+26], while SHA3 and SHAKE remain substantial computational costs. Kyber's round-three specification reports that additional hashing consumes nearly half of the encapsulation cycles in its AVX2 implementation on Intel processors [ABD+21]. For HAETAE, profiling on an Intel Core i7-10700K shows that SHAKE256 accounts for almost 60% of signing time in the reference implementation [CCD+24].

Parallel hashing is already effective in hash-based signatures. SPHINCS+ implementations process independent hashes in parallel during tree computations [BC23]. Lattice-based schemes use fixed-output hashing (e.g., public-key hashing) and SHAKE-based pseudorandom generation for matrix expansion and sampling. These computations use the same Keccak permutation [Nat15], so their independent permutation steps can share SIMD calls once their inputs are available. The scheduling problem is to choose which streams advance together and for how long, given their different lengths and dependencies.

For example, the original AVX2 implementation of ML-KEM-512 decapsulation handles some ready hashes through single-stream calls separately from matrix expansion, missing opportunities to share parallel calls. HAETAE also combines matrix expansion

and message hashing with long SHAKE streams for Gaussian sampling. Differences in stream lengths, dependencies, and the number of ready streams make it difficult to keep SIMD slots fully occupied.

Xu et al. [XHPL26] improve SIMD utilization in Fiat-Shamir with Aborts signatures [Lyu09] by executing different SHA3 algorithms in parallel. For signing, they fill spare slots by precomputing SHAKE output for sampling masking vectors y in later attempts. Its benefit depends on the number of rejected attempts.

We develop a cost-aware method for scheduling Keccak permutations across hash streams during a single execution of a cryptographic algorithm. It groups ready streams and divides long streams into segments for regrouping. Permutation, state-movement, and preparation costs guide the choice of groups, segment lengths, and state placements throughout the search. Search runs offline; deployed implementations execute precomputed schedules. Our HAETAE schedules use the streams required by the current sampling attempt, without precomputing output for later attempts. The method also applies to KEMs, as demonstrated by ML-KEM's matrix expansion and dependent hashes.

We apply our scheduling method to ML-KEM and HAETAE. ML-KEM is standardized by NIST [Nat24], and HAETAE is one of the signature schemes selected by the Korean Post-Quantum Cryptography (KpqC) competition [Kor25]. Their hash workloads also differ: ML-KEM combines short matrix and noise computations with input dependencies, whereas HAETAE requires long SHAKE outputs for Gaussian sampling.

Related work. Extensive research has focused on accelerating the Keccak permutation [BDP+12]. Cabral and López implement parallel SHA3 and SHAKE using AVX2 and AVX-512 [CL14, CL18]. Scalar and SIMD Keccak implementations have also been optimized for AArch64, ARMv7-M, and RISC-V [BK22, HAZ+24, ZYHK25]. XKCP provides parallel state and permutation interfaces [The26].

Lei et al. use parallel SHAKE on AVX-512 to process eight streams for polynomial sampling in Dilithium [LHP+23]. Parallel KEM implementations include SaberX4, which batches four operations [Sin19], and Yu et al.'s Kyber implementation, which batches 16 instances with AVX2 and 32 with AVX-512 [YZH+25].

Results. We evaluate ML-KEM using Keccak kernels that process two streams in parallel on Apple M5, four on Cortex-A72 and AVX2, and eight on AVX-512. HAETAE is evaluated on ARM and AVX2. ML-KEM achieves speedups of up to 1.31× for encapsulation and 1.44× for decapsulation. With identical SHA3 kernels and arithmetic, HAETAE scheduling gives speedups of up to 1.28× for signing and 1.26× for verification on Apple M5. The AVX2 implementation achieves up to 1.24× for signing and 1.21× for verification. We also obtain further AVX-512 key-generation speedups over Xu et al.'s Dilithium implementation. Section 5 gives all parameter-specific results and ablations.

Contributions.

1. **Cost-aware SIMD permutation scheduling.** We select stream groups, segment lengths, and state placements using the combined costs of permutation calls, state movement, initialization, and input preparation.
2. **Efficient SHAKE implementations.** We implement the selected schedules through state preparation, retention, and output delivery, applying the same method to ML-KEM and HAETAE on backends with different parallel widths and state layouts.
3. **General tool for organizing hash computations.** We provide an offline tool that organizes SHA3 and SHAKE computations within cryptographic algorithms into SIMD Keccak schedules. The tool separates algorithm-specific workloads and dependencies

from backend call types and costs, allowing the same scheduling method to be reused across schemes and implementations.

Section 2 introduces the schemes, Keccak, and notation. Section 3 develops the scheduling method and cost model; Section 4 describes the implementations; and Section 5 evaluates schedule quality and performance. Section 6 discusses the results and side-channel analysis.

2 Preliminaries

Notation. Table 1 distinguishes hash functions, indexed streams, and their states. The following subsections introduce the corresponding computations and storage.

Table 1: Notation for hash functions, streams, and Keccak permutation calls.

Symbol	Meaning
f	The Keccak-f[1600] permutation
H, G, J	ML-KEM hash functions: SHA3-256, SHA3-512, and SHAKE256 with a 32-byte output, respectively
A_i, N_i	ML-KEM matrix-expansion and noise-sampling streams, respectively
Y_i	HAETAE SHAKE256 stream supplying a Gaussian sampler
S_i	The complete 200-byte Keccak state of stream i
w	Number of state slots in a parallel permutation call
P_w	One kernel call applying f once to the state in each of w slots

2.1 SHA3, SHAKE, and Keccak

SHA3 and SHAKE use the sponge construction with the 24-round Keccak-f[1600] permutation, denoted by f. Its 1600-bit state occupies 200 bytes and consists of 25 words of 64 bits. The rate determines how many input or output bytes are processed between permutations; the remaining bytes form the capacity. We call one SHA3 or SHAKE evaluation a stream. Table 2 lists the functions used in this work [Nat15].

Table 2: Rate and output length of the SHA3 and SHAKE functions used in this work.

Function	Rate / bytes	Output / bytes
SHAKE128	168	Variable
SHAKE256	136	Variable
SHA3-256	136	32
SHA3-512	72	64

Each stream starts from a zero state. During absorption, input blocks are XORed into the rate portion before applying f; the final block includes the function suffix and padding. After the final absorption permutation, squeezing reads output from the rate portion. Another permutation is needed when further output is required after the current rate-sized block has been exhausted [Nat15].

One complete application of f to a stream's state is a permutation step. Steps of one stream are sequential; steps of different streams can execute independently once their inputs are available. Absorption steps need not produce output blocks. For a 34-byte input, the first 504 output bytes of SHAKE128 require three permutations: the final absorption permutation supplies the first block, and two squeezing permutations supply the next two.

2.2 Hashing in ML-KEM and HAETAE

ML-KEM uses SHAKE128 for matrix expansion and SHAKE256 for centered-binomial noise sampling, with streams distinguished by matrix indices or noise nonces [Nat24]. We label the matrix-expansion streams A_i and the noise-sampling streams N_i, where i is a zero-based stream number within each family. Noise sampling consumes a fixed amount of output; matrix rejection sampling may require further bytes. In schedules, H, G, and J label streams evaluating the corresponding ML-KEM functions in Table 1.

During encapsulation, G derives the shared secret and encryption randomness from the message m and the hash of the encapsulation key ek, giving the dependency chain

$$H(ek) \longrightarrow G(m\|H(ek)) \longrightarrow \text{noise generation.} \tag{1}$$

Matrix expansion is independent of this chain because its seed is contained in ek. During decapsulation, G uses the recovered message and stored key hash, so it waits for decryption. In contrast, $J(z\|ct)$ can start from the stored value z and input ciphertext ct. Final key selection waits for the ciphertext re-encryption check [Nat24]. The $H(ek)$ stream in Figure 1 performs the decapsulation-key hash check [Nat24].

HAETAE signing samples a vector from a discrete hyperball using Gaussian samples, with a separate SHAKE256 stream supplying the sampler for each polynomial component. We label these streams Y_i, indexed from zero. They produce long output sequences; other hashes expand matrices, hash messages, and derive seeds [CCD+24].

A consumer is code that processes hash output, including matrix rejection sampling, centered-binomial sampling, and Gaussian sampling. When more bytes are needed, squeezing continues from the same SHAKE state. A new sampling attempt follows the scheme's seed and nonce rules. Continuation and new attempts are requested after the corresponding sampling or rejection results are known. Polynomial arithmetic is outside the hash scheduling cost.

2.3 Parallel Keccak Calls and State Storage

A parallel Keccak call applies f independently to several stream states. We write P_w for one kernel call with w state slots, each holding a complete 200-byte state. The call applies f to every slot, including slots whose output is unused. With four active streams, P_4 advances each by one permutation step in one call, whereas $4P_1$ performs those steps through four separate calls. The number w describes the kernel's parallelism, rather than the instruction set itself.

A working buffer holds the states used by a parallel call. For example, a four-stream working buffer holds 800 bytes of state data. Changing the stream group may therefore require moving states into or out of that buffer.

In contiguous storage, each stream's 25 words are stored together. In interleaved storage, corresponding words from different streams are stored together. These layouts affect how stream states are moved when groups change. A stream can pause after any complete application of f and resume later if its complete state, byte positions, and any output needed later are preserved. We call this point a permutation boundary.

3 Cost-Aware SIMD Hash Scheduling

Hash streams become ready at different points and require different numbers of permutations. Processing them in fixed groups can therefore leave SIMD slots without useful work. We construct schedules that change groups at permutation boundaries, using permutation, state-movement, and preparation costs to decide how the streams should advance.

Figure 1(a) shows ML-KEM-512 decapsulation. Matrix streams can advance alongside the encapsulation-key hash $H(ek)$ and the fallback-key derivation $J(z\|ct)$. Noise streams wait for the encryption randomness derived by G. Grouping ready work and replacing completed streams changes the illustrated sequence from $5P_4 + 14P_1$ to $8P_4 + P_1$ for the valid-input path with three initial output blocks per matrix stream.

In Figure 1(b), HAETAE-2 has six ready SHAKE256 streams for Gaussian sampling, $Y_0, \ldots, Y_5$, each requiring 35 initial output blocks. Processing four streams and then two uses $70P_4$. Dividing each stream into segments of consecutive permutation steps lets it resume in different four-stream groups, reducing the count to $53P_4$.

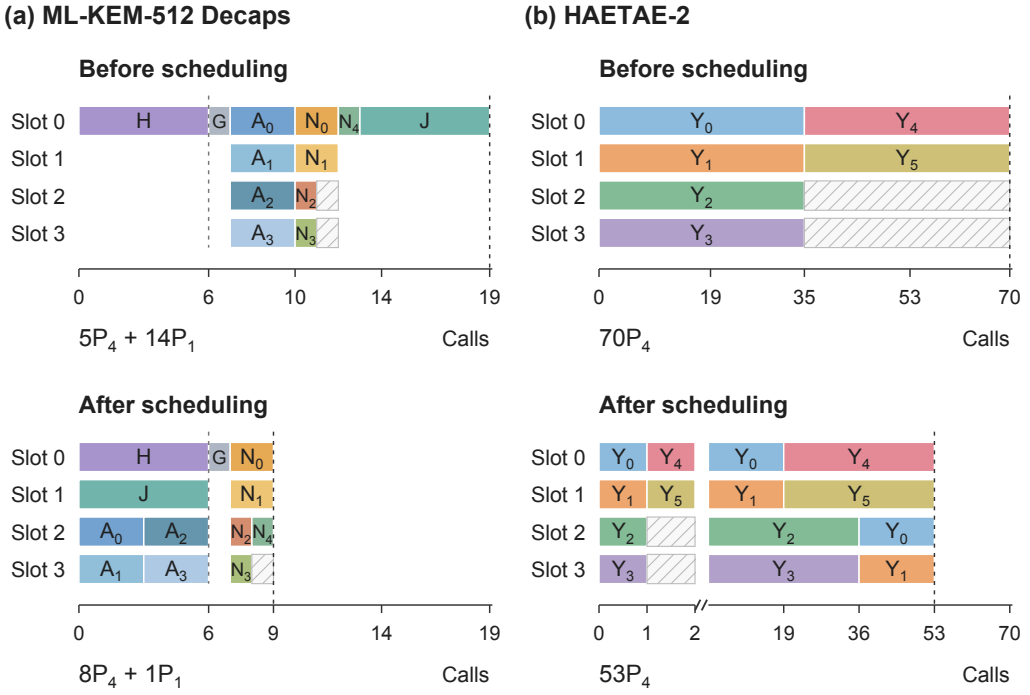

Figure 1: SIMD hash scheduling before and after regrouping: (a) ML-KEM-512 decapsulation; (b) HAETAE-2 Gaussian sampling. Hatched regions mark slots with no required output.

Regrouping can add costs for state movement, initialization, and input preparation. We first explain how states are preserved, then use these costs to guide offline scheduling.

3.1 Stream Switching and Regrouping

A stream need not remain in one group until it finishes. At a permutation boundary, completed streams release slots for other ready work, and unfinished streams can pause before joining a different group.

Regrouping may require state movement through copy, export, or import operations. These operations form the movement cost M_p in Section 3.2. Keeping a state in its slot or an undisturbed working buffer avoids these transfers.

For each stream, regrouping preserves its complete state, byte positions, and output needed later. If its original operations remain unchanged and in order, the next permutation starts from the same state as in separate evaluation. Since f acts independently

on each state, it produces the same next state. Repeating this argument preserves the stream's output.

3.2 Permutation Savings and Scheduling Overheads

A schedule that saves permutation calls may spend more on transfers or preparation. We compare these costs together. For a schedule s on backend p, the modeled cost is

$$\begin{aligned}\widehat{T}_p(s) &= P_p(s) + M_p(s) + I_p(s) \\ &= \sum_k \pi_{p,k} n_k(s) + \sum_m \theta_{p,m} N_m(s) + \sum_h \beta_{p,h} I_h(s).\end{aligned} \tag{2}$$

Here n_k, N_m, and I_h count permutation, state-transfer, and preparation operations; π, θ, and β give their prices in a common time unit. Backend p specifies the platform, implementation, layout, and calling convention. Each price covers one complete priced operation, including all participating states. P_p covers single-state and parallel calls to f, including the kernel's internal state accesses (Section 2.3); M_p covers separate complete-state copy, export, and import operations induced by regrouping (Table 3); I_p covers initialization and non-permutation input preparation, including placement of new states. Preparation prices distinguish single-state and batch calls, shared or distinct seeds, input length, and layout. Output delivery and consumer work are held fixed. A measured call combining preparation and permutation replaces the separate charges for the work it covers and is counted once.

For the before and after schedules s_0 and s in Figure 1(a), the cost difference is

$$\Delta\widehat{T}_p = \widehat{T}_p(s) - \widehat{T}_p(s_0) = 3\pi_{p,4} - 13\pi_{p,1} + \Delta M_p + \Delta I_p. \tag{3}$$

The regrouped schedule has lower estimated cost when this difference is negative.

Segment length affects this trade-off. Consider $v \le w$ ready streams, each advancing through q successive one-step calls. Single-state execution costs $qv\pi_{p,1}$; width-w execution costs $q\pi_{p,w}$. Writing ΔM_p and ΔI_p for the parallel arrangement's overhead differences, it is cheaper when

$$q\big(v\pi_{p,1} - \pi_{p,w}\big) > \Delta M_p + \Delta I_p. \tag{4}$$

A longer segment spreads one-time regrouping costs over more permutation steps. This inequality uses one-step prices. A separately measured multi-step call can replace the corresponding step charges.

Table 3 specifies the transfers charged to M_p for grouping and placement. Temporary saves needed to avoid overwriting unread source states are also counted.

Grouping also affects I_p. Preparing fresh streams together may permit a batch call supported by the backend; introducing them separately or beside live states may require different preparation calls. The scheduler compares their prices for the streams' prescribed seeds, input lengths, and destination layout. It also counts non-permutation input preparation as the streams advance. Thus grouping and placement can change I_p even when the required inputs and permutation steps remain the same.

The search in Section 3.3 uses these charges when constructing each next action, rather than only scoring completed schedules.

3.3 Cost-Guided Scheduling

The scheduler builds a schedule offline by choosing a ready group, its segment length, and its state placement, then charging the action using Section 3.2. It retains several partial schedules so that the cost of later work can influence which choices survive.

Table 3: State movement induced by regrouping. Preparation of a new stream is counted in I_p.

State placement and lifetime	Transfer to count
State retained in its slot or an undisturbed working buffer	None
State has no further use	Discard; no export
Live state moves to another slot in the same layout	Copy
Live state must survive an overwrite of its slot	Save: copy for contiguous states; export for interleaved states
Saved state resumes in an interleaved slot	Import
State moves between interleaved layouts without a direct conversion	Source export and destination import

Requests and initial state. Each hash request specifies a stream's known permutation steps and dependencies. Requests may also depend on external completion or acceptance events. A request is ready when its dependencies are satisfied and any required external result is confirmed. A prefix that cannot advance until such a result is supplied is returned as a waiting prefix. The backend specifies supported calls, layouts, and prices; the configuration fixes the working buffers and state/output memory budget. A new algorithm supplies its own requests and dependencies to the same search.

Search starts before any requested permutation has run. With first-use preparation, buffers are empty and streams are uninitialized. Selecting A_0, A_1, A_2, H for a P_4 action prepares those states in their assigned slots and charges I_p; the other streams remain uninitialized. A batch call is available only if it supports the selected inputs. With entry preparation, ready states instead start in individual buffers and their preparation contributes to the initial cost.

Groups and segment lengths. At each search state, the scheduler orders ready streams by remaining dependency-chain length, remaining work in either direction, and residence in each working buffer. Secondary priorities use remaining work or chain length; stream order breaks ties. Each ordering proposes its first stream for a single-state call and, with at least two ready streams, up to its first w streams for a supported parallel call. Groups matching backend batch preparation or input calls provide further choices.

For a single-state group, the scheduler tries one step and all remaining steps. For a parallel group, it tries one step, the smallest remaining step count among its members, and balanced lengths for equal workloads. These lengths let a long stream leave one group before finishing, as Figure 1(b) requires.

For the six streams in that figure, one initial step per stream leaves 34 steps each. The overlapping groups are (Y_0, Y_1, Y_2, Y_3), (Y_2, Y_3, Y_4, Y_5), and (Y_4, Y_5, Y_0, Y_1). Each stream appears in two groups, so each group advances by $34/2 = 17$ steps. Every stream receives $1 + 2 \cdot 17 = 35$ steps. The initial two calls and these three segments use $2 + 3 \cdot 17 = 53$ four-way calls.

More generally, for $n > w$ independent ready streams with b known steps each, shift the group by g stream indices:

$$g = \gcd(n, w), \qquad \ell_{\max} = n/g, \qquad r = w/g. \tag{5}$$

The stream indices in group ℓ are

$$C_\ell = \{(\ell g + u) \bmod n : 0 \le u < w\}, \qquad 0 \le \ell < \ell_{\max}. \tag{6}$$

For any stream index, exactly $w/g = r$ offsets in $0, \ldots, w-1$ have the same residue modulo g. Each offset determines one group, so each stream appears in r groups. Advance every stream through h initial steps, then run each group for q steps, where

$$h = b \bmod r, \qquad q = (b-h)/r. \tag{7}$$

This completes $h + rq = b$ steps per stream. Initial steps use groups of at most w; zero-length parts are omitted. Partial groups require initialized unused slots or a supported narrower call. Adjacent cyclic groups share $w - g$ streams, which can retain their slots. For six streams, $g = 2$, $r = 2$, $h = 1$, and $q = 17$. This construction supplies balanced segment lengths and an initial schedule, priced by the same cost model as other schedules.

Pricing and comparing actions. An action combines a proposed group and length with matching preparation/input calls and, for a parallel call, a working buffer. Applying it updates progress, state placement, and physical call counts. Actions that violate dependencies, backend requirements, state preservation, or the memory budget are rejected. If action a extends partial schedule x to x', its accumulated cost is

$$K(x') = K(x) + \Delta P_p(a) + \Delta M_p(a) + \Delta I_p(a). \tag{8}$$

Here $K(x)$ is the cost of the actions already selected, including any entry preparation.

Consider the state after Figure 1(b)'s first two calls. All six streams have 34 steps left; Y_4, Y_5 remain in the working buffer and the other states are saved individually. Remaining-work order proposes Y_0 alone or $Y_0, \ldots, Y_3$ together. Resident-first order proposes Y_4, Y_5, Y_0, Y_1. Table 4 compares their one-step actions; each four-way group also provides 17-step and 34-step choices.

Table 4: Next-call choices from the same six-stream state. Costs use the HAETAE AVX2 component prices measured on the Core i5-11500.

Next call	Stream steps	State transfers	Added cost (cycles)
P_1: Y_0	1	None	1101
P_4: Y_4, Y_5, Y_0, Y_1	4	2 imports	1918
P_4: Y_0, Y_1, Y_2, Y_3	4	2 exports + 4 imports	2126

All six states are initialized, so these actions add no preparation cost. The single-state call costs least but completes less work. The two parallel calls advance the same number of steps, with lower movement cost when resident states are retained. To compare different amounts of progress, the search also estimates the permutation work remaining.

Let $W(x)$ be the remaining stream steps and $D(x)$ the remaining permutation steps on the longest request-dependency chain. The remaining-cost estimate is

$$L_P(x) = \max\left\{W(x)\min\left(\pi_{p,1}, \frac{\pi_{p,w}}{w}\right), D(x)\min(\pi_{p,1}, \pi_{p,w})\right\}. \tag{9}$$

The first term prices total work at the cheapest cost per state step; the second prices the sequential dependency chain at the cheapest call cost. Each is a lower bound under one-step permutation prices, so their maximum is also a lower bound. Partial schedules are ranked by

$$R(x) = K(x) + 2L_P(x). \tag{10}$$

The fixed heuristic weight 2 gives remaining work extra priority; completed schedules are compared by K. In Table 4, retaining the resident streams costs about 817 cycles more than the single-state call but lowers L_P by about 1350 cycles, giving a lower R.

Algorithm 1 Offline cost-guided SIMD scheduling.

Require: Configured initial state x_0, operation prices, B_{beam}, search budget
Ensure: A schedule, a waiting prefix, or $\perp$

$(\mathcal{C}, \mathcal{W}) \leftarrow \text{ConstructSchedules}(x_0)$
$\mathcal{F} \leftarrow [x_0]$
while $\mathcal{F} \neq \emptyset$ and budget remains **do**
  Remove the first schedule x from $\mathcal{F}$
  $A \leftarrow \text{choices}(x)$
  if $A = \emptyset$ and x awaits an external result **then**
    Add x to $\mathcal{W}$
  end if
  for all $a \in A$ **do**
    $x' \leftarrow \text{Apply}(x, a)$ ▷ adds $\Delta P_p + \Delta M_p + \Delta I_p$
    if $x' = \perp$ **then continue**
    if x' is complete **then**
      Add x' to $\mathcal{C}$
    else
      Append x' to $\mathcal{F}$
    end if
  end for
  Sort $\mathcal{F}$ by increasing R
  Keep only the first entry for each equivalent state
  Retain the first B_{beam} entries of $\mathcal{F}$
end while
if $\mathcal{C} \neq \emptyset$ **then**
  return $\text{Replay}(\arg\min_{x \in \mathcal{C}} K(x))$
else if $\mathcal{W} \neq \emptyset$ **then**
  return $\text{Replay}(\arg\min_{x \in \mathcal{W}} K(x))$
else
  return $\perp$
end if

Future movement and preparation enter K when later actions are explored; L_P estimates only permutations.

Building a complete schedule. Algorithm 1 starts from the configured initial state and an empty trace. ConstructSchedules also supplies complete schedules that retain resident streams, prioritize long dependency chains, or use the cyclic construction for equal workloads, all with the same initial state and prices.

Pending schedules are ranked by R, keeping the cheaper partial schedule for equivalent states. Equivalence includes progress, completed dependencies, confirmed results, state locations, initialization status, and preparation counts. At most B_{beam} entries are retained until none remain or the time/expansion budget is reached. Replay checks progress, physical calls, and cost.

In Algorithm 1, $\mathcal{F}$ holds pending schedules, $\mathcal{C}$ complete schedules, and $\mathcal{W}$ prefixes waiting for external results. Each partial schedule x records its execution state, cost, and call sequence. $\text{Apply}(x, a)$ creates a successor without modifying x, appends a to its call sequence, and charges its cost; unsupported actions return $\perp$.

The deployed implementation executes the selected call sequence or constant tables. Refills and retries use predefined continuation paths; runtime does not run the search or evaluate the cost model.

3.4 Lower Bounds and Schedule Quality

For a complete returned schedule, we evaluate permutation cost against a lower bound and full scheduling cost against a reference schedule. Both comparisons fix the requests, dependencies, backend calls and prices, and memory budget; Section 5.3 reports their evaluation scopes separately.

Permutation cost. All steps are priced separately as $P_p^{\text{step}}(s) = \pi_{p,1} n_1(s) + \pi_{p,w} n_w(s)$, including those within combined calls. A q-step assembly function contributes q to the count of its width. Let $B_p(W, D)$ minimize this one-step cost subject to enough capacity for the total work and enough successive calls for the longest dependency chain:

$$\begin{aligned} B_p(W, D) = \min_{n_1, n_w \in \{0,1,\dots\}} & \left(\pi_{p,1} n_1 + \pi_{p,w} n_w\right) \\ \text{with} \quad & n_1 + w n_w \geq W, \qquad n_1 + n_w \geq D. \end{aligned} \tag{11}$$

Here W and D are the total permutation steps and longest dependency-chain length. Every executable schedule satisfies these capacity and chain constraints, so the minimum is a lower bound.

We compute transitive dependencies, including paths through zero-step requests and external completion events. Among requests with positive work, we connect pairs for which neither precedes the other. The connected components form stages ordered by dependency: transitivity makes the precedence direction between components consistent. All requests in one stage finish before any in the next stage can start.

For each request in a stage, choose a dependency chain ending at it with the most permutation steps, breaking ties by a fixed topological order. Let c be its number of steps and F all requests in the stage that depend on it. Write W_F and D_F for the work and longest dependency chain in F, both zero when F is empty. The chain and F require at least $d = c + \max\{D_F, \lceil W_F / w \rceil\}$ calls. An a-step stream outside the chain and F can share at most n_w of these calls. Counting their union gives the constraints

$$n_1 + n_w \geq d, \qquad n_1 + 2n_w \geq d + a. \tag{12}$$

Let $L_{P,j}$ minimize the same integer call cost with the capacity and chain constraints for stage j and these additional constraints for every chain endpoint and outside stream. The evaluation bound is

$$L_P^{\text{eval}} = \sum_{j=1}^{t} L_{P,j}. \tag{13}$$

A permutation call cannot advance required work in two such stages, so their bounds can be added. For the returned schedule s_h, the relative gap is

$$g_P^{\text{LB}}(s_h) = \frac{P_p^{\text{step}}(s_h) - L_P^{\text{eval}}}{L_P^{\text{eval}}}. \tag{14}$$

Zero proves minimum permutation cost under the specified one-step prices and dependencies; a positive gap can also include looseness in the bound. With four-way calls only, Figure 1(b)'s six streams require at least $\lceil 210/4 \rceil = 53$ calls. Section 3.3's construction reaches this minimum.

Full scheduling cost. We compare the $P_p + M_p + I_p$ cost of the returned schedule s_h with a reference s_{ref}, including s_h in the reference pool:

$$g_{\text{cost}}(s_h) = \frac{\widehat{T}_p(s_h) - \widehat{T}_p(s_{\text{ref}})}{\widehat{T}_p(s_{\text{ref}})}. \tag{15}$$

An exact reference enumerates ordered ready groups, permitted integer segment lengths, working buffers, and priced input calls under the declared placement rules. Non-negative costs permit pruning by accumulated cost; optimality requires excluding every cheaper schedule. A larger budget-limited search instead supplies the cheapest schedule found.

Zero proves minimum modeled cost in the action domain only for an optimal reference; otherwise, no cheaper schedule was found. Both gaps have positive denominators; smaller is better. Planning time includes construction, search, and replay.

4 Efficient Implementation of SIMD Hash Schedules

Section 3 selects stream groups, segment lengths, and state placements. We implement these choices by preparing or moving states at segment boundaries, retaining them across consecutive permutations, and delivering output in stream order. The Keccak round code remains unchanged. Table 5 distinguishes the caller's memory layout from the storage used during execution.

Table 5: State storage in the implemented backends. Width counts complete states. ARM uses 128-bit vector registers, AVX2 uses YMM registers, and AVX-512 uses ZMM registers.

Backend	Width	Caller memory	During consecutive permutations
ARM SHA3 (M5)	2	Two contiguous states	Twenty-five registers, one word from each state per register
ARM NEON (A72)	4	Four contiguous states	NEON pair in registers; other states use scalar loads and stores
mlkem-native AVX2	4	Four contiguous states	Nine YMM registers and sixteen SIMD vectors on the stack
HAETAE AVX2	4	Interleaved state words	Permutation accesses the working buffer; output-block loop stays inside assembly
ML-KEM AVX-512	8	Interleaved state words	Twenty-five ZMM registers

4.1 Preparing and Moving State Slots

New streams are initialized directly in the slots selected by the schedule. A fresh working buffer can be cleared as a whole. When other slots contain live states, only the new slot is cleared. Preparation writes each stream's input, function suffix, and padding into the cleared storage.

For a working buffer of w states, let $S_i[j]$ denote word j of stream i's state and B the array of 64-bit words storing the buffer. Contiguous and interleaved storage use

$$\begin{aligned} &\text{Contiguous:} && B[25i + j] = S_i[j], \\ &\text{Interleaved:} && B[wj + i] = S_i[j], \end{aligned} \qquad 0 \le i < w, \quad 0 \le j < 25. \tag{16}$$

A contiguous state's words occupy consecutive addresses; an interleaved state's words are spread across 25 rows. Figure 2(a) shows the writes needed to prepare one slot while preserving the other states.

ARM Apple M5 and Cortex-A72 both use contiguous states, so saving or moving a live state uses a contiguous 200-byte copy.

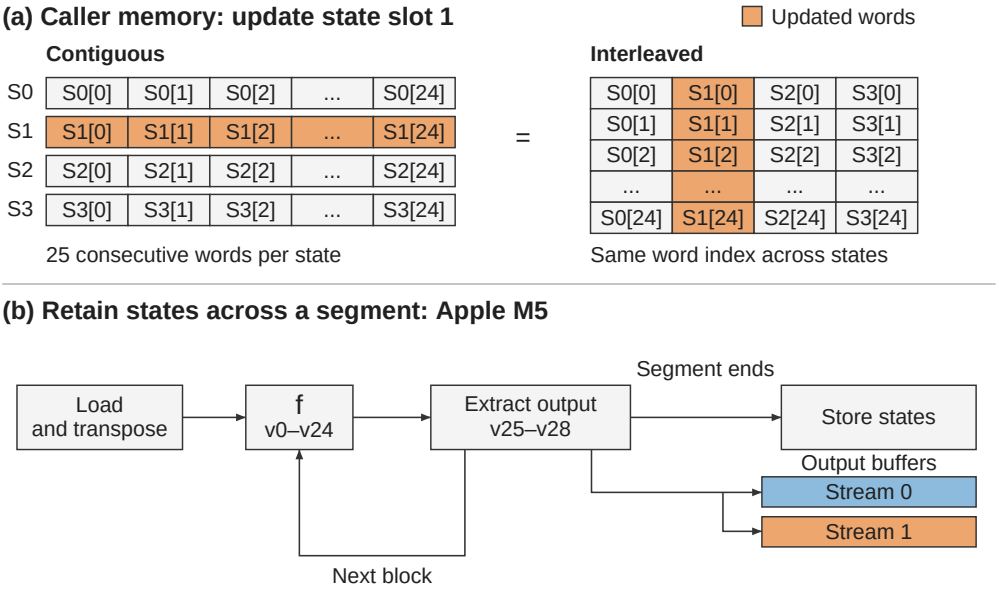

Figure 2: (a) Preparing one state slot while preserving its neighbors; highlighted words are updated. (b) Apple M5 execution within a segment: output extraction uses temporary registers, and complete states are stored when the segment ends.

AVX2 When preparing a whole group of new interleaved states with a shared seed, broadcast instructions copy each seed word into every stream's element, and a vector supplies the distinct nonces. Each state-word vector holds one 64-bit word from each of the four states. Preparing one slot updates its element in all 25 vectors. Input bytes are XORed into the corresponding elements without changing the other live states. The HAETAE AVX2 functions that initialize four states from a seed are not inlined into the sampler.

An interleaved-state copy selects one source element and replaces one destination element in each of the 25 vectors. Algorithm 2 shows one word update; repeating it at 32-byte address increments copies all 25 state words. The selection indices and destination mask are constructed once from the public slot indices. Export gathers the selected 25 words into contiguous storage; import inserts them into the destination slot. A contiguous state can instead be copied as 200 consecutive bytes or used directly by a single-state kernel.

Algorithm 2 AVX2 word update for an interleaved state-slot copy

Require: Corresponding word vectors at rsi, rdi in disjoint or identical buffers; slots $a, b \in \{0, 1, 2, 3\}$.

Require: $ymm_4 = (2a, 2a+1, 2a, 2a+1, 2a, 2a+1, 2a, 2a+1)$; ymm_5: byte mask selecting the 64-bit element b.

Ensure: Destination element b receives source element a; other destination elements are unchanged.

```
1: vmovdqu ymm0, [rsi]
2: vpermd ymm1, ymm4, ymm0                    ▷ Select source slot
3: vmovdqu ymm2, [rdi]
4: vpblendvb ymm2, ymm2, ymm1, ymm5           ▷ Replace destination slot
5: vmovdqu [rdi], ymm2
```

AVX-512 Word j of slot i is stored at index $8j + i$. Initialization, input updates, copy,

export, and import access these words without modifying neighboring slots; individual state buffers use contiguous storage. Whole-buffer initialization requires fresh storage.

4.2 Retaining States Across Consecutive Permutations

One assembly function executes a segment's consecutive permutations when no input update, slot replacement, or consumer decision is needed between them. Layout conversion occurs before the loop, output is written after each permutation, and final states return in the caller's layout. Figure 2(b) shows this loop on Apple M5; the amount of state retained in registers depends on the backend.

Apple M5 The assembly function loads and transposes the two contiguous states into $v_0, \ldots, v_{24}$, each holding corresponding words from the two streams. Output extraction uses $v_{25}, \ldots, v_{28}$ as temporaries: `trn1` and `trn2` select the respective stream's words without changing the state registers. The next permutation reuses these registers; complete states are written back when the segment ends.

The assembly function accepts a variable block count and a separate output rate for each stream. It can therefore write 168 bytes for one stream and 136 for the other after the same permutation. The block count ends at the first point where either stream needs a consumer result.

Cortex-A72 The four-stream backend combines two-state NEON processing with scalar code for the other two states and uses no ARM SHA3 instructions. Its multi-block function retains the NEON pair in vector registers between blocks; the other states retain the kernel's load/store organization.

AVX2 For ML-KEM matrix expansion, one AVX2 assembly call generates three SHAKE128 blocks per stream. The function keeps the four states in nine YMM registers and sixteen SIMD vectors on the stack. Output conversion uses temporary registers and preserves these values. Entry and exit transpositions therefore occur once for three blocks rather than once per block. The HAETAE interleaved backend provides assembly functions that write either separate byte buffers or SIMD vectors. Its permutation still accesses the working buffer, while the loop over output blocks remains inside the assembly function.

AVX-512 The eight-way function keeps the states in $zmm_0, \ldots, zmm_{24}$ across complete 24-round permutations and stores them on return. Temporary registers extract output; each output pointer advances by its stream's block size. Output buffers must not overlap state storage.

4.3 Delivering Output in Stream Order

Regrouping changes which streams produce output together. Delivery preserves each stream's byte order and supplies the amount of output required by its consumer. Output is buffered when it must survive a later permutation, and incomplete Gaussian records are saved across blocks. Refill blocks are generated after the consumer reports that more bytes are needed.

ARM Both ARM backends deliver output to separate per-stream byte buffers. The Cortex-A72 function accepts four output pointers and a separate output length for each stream. The output-buffer write positions advance independently. Gaussian consumption

and refill follow the consumer's stream order, preserving the squared-norm update order. A single-state refill advances the corresponding contiguous state in place.

For ML-KEM on both ARM backends, CBD2 reads the first 128 state bytes before the slot is reused. Output spanning several blocks is buffered until consumption.

AVX2 For consumers that accept separate byte strings, a four-by-four transpose separates the interleaved output into per-stream buffers [XHPL26]. A SHAKE128 block has 21 words per stream and a SHAKE256 block has 17. The output conversion processes four words at a time and then the final word. Matrix consumers receive their initial batch of 504 bytes for ML-KEM or 672 bytes for HAETAE, rather than being invoked after every partial segment.

For HAETAE's Gaussian consumer, output stays in SIMD vector form. We record where each stream's output blocks were written, then use vector permutations and blends to assemble its 17-byte sampling records. Figure 3(a) shows Y_0 moving from slot 0 to slot 2 in the HAETAE-2 AVX2 schedule. This avoids separating output into byte strings and then rebuilding vectors for the consumer.

Each stream keeps its accepted-sample count and incomplete record bytes. After the 32 sign bytes, the first 136-byte block contains six 17-byte records and two remaining bytes. Combining those two bytes with the next block gives eight records and another two-byte remainder, as Figure 3(b) illustrates. The diagram shows logical byte order; the implementation assembles the consumer's input directly from SIMD vectors and preserves its record grouping and squared-norm update order.

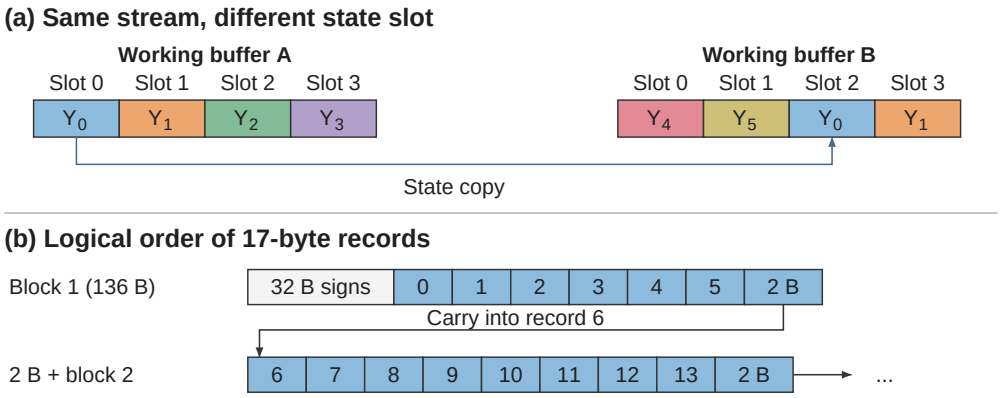

Figure 3: HAETAE-2 AVX2 output organization. (a) State slots change while sampling progress stays with the stream. (b) The first two output blocks in logical record order; numbers identify consecutive 17-byte records.

For the initial batch with only two Gaussian streams, each stream contributes six records. A small byte buffer separates even- and odd-indexed records into four useful SIMD inputs. Subsequent batches remain in vector form. The three parameter sets use the same vector input format with their respective stream assignments.

AVX-512 Output conversion uses unpack and shuffle instructions on temporary registers to form per-stream byte buffers, leaving the state registers unchanged.

In ML-KEM-512 KeyGen, one assembly function executes three consecutive eight-way permutations for four matrix streams and four noise streams. Matrix streams receive three 168-byte blocks. Noise streams receive 136 bytes after the first step and 64 after the second, of which CBD3 uses 56. All eight states advance at every step, but the third step

writes no further noise output. Output writes and consumption follow each stream's byte requirements. Output buffers can be reused after their final required read.

5 Evaluation

We evaluate performance and the modeled cost of the selected schedules.

5.1 Experimental Setup

We benchmark ML-KEM and HAETAE on an Apple M5 running macOS 26.6 and a bare-metal Intel Core i5-11500 running Ubuntu 22.04.5. We use Apple clang 21.0.0 and GCC 11.4.0, respectively, with `-O3`. We also benchmark a Cortex-A72 at 1.8 GHz running Ubuntu 24.04.5 with GCC 11.4.0 and `-O3`. The x86 benchmarks run on CPU 2 with Turbo disabled and a 2.5 GHz frequency policy. All performance tables report the average cycle count over 10,000 executions. Speedup is $T_{\text{baseline}}/T_{\text{ours}}$, using the unrounded average cycle counts.

For ML-KEM, we select baselines by comparing compatible public implementations on Apple M5 and the i5-11500. The fastest identified versions are the actively maintained mlkem-native on Apple M5 and AVX2, and wolfSSL on AVX-512 [pq-26b, wol26]. Our baseline survey also included liboqs's AVX-512VL implementation [Ope26]; wolfSSL had lower cycle counts for all three ML-KEM operations at all three parameter sets on the i5-11500. The pq-crystals Kyber repository is no longer actively maintained; the evaluated version also differs in interface behavior [BDK+26]. For HAETAE, we compare with the official v2.0 and Xu et al.'s AVX2 implementations [HAE23, XHPL26]. For HAETAE on Apple M5, we build on Sim et al.'s code [SLS24], supplying missing dependencies from official HAETAE and Keccak kernels from mldsa-native [pq-26a]. The baseline and scheduled Apple M5 implementations share these dependencies and correctness repairs. No compatible public HAETAE AVX-512 implementation was identified in this survey, so HAETAE results cover ARM and AVX2. For the AVX2 signing comparisons, all implementations use a 32-byte message digest in seed derivation. Dilithium uses the same i5 setup.

Component prices measure complete permutation, state-transfer, and preparation operations in their backend layouts. A measured preparation-plus-permutation call replaces the separate charges for its work. Model comparisons use the same prices and fixed initial requests on both sides; runtime measurements include any additional refills and retries.

5.2 Performance

Table 6 reports cycles and speedups across all parameter sets. Sign includes natural retries.

In Table 6, ML-KEM encapsulation and decapsulation show larger gains than key generation, consistent with the cost reductions in Figure 5.

Scheduling ablations. The ML-KEM comparisons in Table 6 retain the baseline polynomial arithmetic and Keccak round code, changing hash scheduling and the supporting state preparation, retention, and output delivery described in Section 4. For HAETAE, Table 7 isolates hash scheduling from the other implementation changes. The Apple M5 variants use identical SHA3 kernels, arithmetic, and sampling consumers; the reference executes one stream at a time. The AVX2 KeyGen variants retain the same optimized FFT and secret-sample decoder and differ in hash scheduling. Apple M5 signing and verification retain speedups of 1.25–1.28× and 1.22–1.26×, respectively. AVX2 KeyGen

Table 6: ML-KEM and HAETAE performance in cycles. Speedups (×) compare Ours with each reference row; bold ratios exceed 1.

Operation	Implementation	ML-KEM-512		ML-KEM-768		ML-KEM-1024	
		Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
ARM (Apple M5)							
KeyGen	**Ours**	12 863	–	20 446	–	30 609	–
	mlkem-native	13 358	**1.04**	21 976	**1.07**	31 360	**1.02**
Encaps	**Ours**	12 531	–	19 994	–	30 106	–
	mlkem-native	15 262	**1.22**	24 044	**1.20**	34 292	**1.14**
Decaps	**Ours**	15 644	–	24 664	–	36 391	–
	mlkem-native	19 982	**1.28**	30 813	**1.25**	43 952	**1.21**
AVX2 (Core i5-11500)							
KeyGen	**Ours**	23 048	–	36 101	–	53 522	–
	mlkem-native	23 802	**1.03**	40 444	**1.12**	56 669	**1.06**
Encaps	**Ours**	22 086	–	32 476	–	49 104	–
	mlkem-native	26 788	**1.21**	42 650	**1.31**	60 508	**1.23**
Decaps	**Ours**	24 970	–	38 465	–	57 403	–
	mlkem-native	35 912	**1.44**	55 449	**1.44**	77 361	**1.35**
AVX-512 (Core i5-11500)							
KeyGen	**Ours**	19 529	–	30 406	–	40 785	–
	wolfSSL	22 764	**1.17**	31 635	**1.04**	40 525	0.99
Encaps	**Ours**	19 496	–	27 237	–	38 515	–
	wolfSSL	23 406	**1.20**	31 572	**1.16**	41 877	**1.09**
Decaps	**Ours**	23 887	–	32 732	–	45 590	–
	wolfSSL	33 571	**1.41**	46 381	**1.42**	60 899	**1.34**
Operation	Implementation	HAETAE-2		HAETAE-3		HAETAE-5	
		Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
ARM (Apple M5)							
KeyGen	**Ours**	601 136	–	1 120 097	–	1 413 183	–
	Baseline	610 161	**1.02**	1 140 729	**1.02**	1 444 256	**1.02**
Sign	**Ours**	1 201 646	–	1 535 300	–	2 240 498	–
	Baseline	1 659 206	**1.38**	2 084 581	**1.36**	3 025 345	**1.35**
Verify	**Ours**	79 286	–	123 384	–	158 637	–
	Baseline	102 872	**1.30**	165 842	**1.34**	216 160	**1.36**
AVX2: official v2.0 comparison							
KeyGen	**Ours**	392 992	–	697 181	–	769 912	–
	Official v2.0	1 061 110	**2.70**	1 917 076	**2.75**	2 584 045	**3.36**
Sign	**Ours**	1 558 075	–	1 946 000	–	2 780 738	–
	Official v2.0	1 925 688	**1.24**	2 123 600	**1.09**	2 960 872	**1.06**
Verify	**Ours**	93 132	–	155 689	–	187 125	–
	Official v2.0	112 987	**1.21**	171 699	**1.10**	203 396	**1.09**
AVX2: Xu et al. comparison							
KeyGen	**Ours**	392 992	–	697 181	–	769 912	–
	Xu et al.	610 611	**1.55**	918 609	**1.32**	961 638	**1.25**
Sign	**Ours**	1 558 075	–	1 946 000	–	2 780 738	–
	Xu et al.	1 762 173	**1.13**	2 041 570	**1.05**	2 908 795	**1.05**
Verify	**Ours**	93 132	–	155 689	–	187 125	–
	Xu et al.	96 626	**1.04**	159 801	**1.03**	186 551	0.997

gains a further 1.02–1.03× with these common optimizations fixed. Its larger speedups in Table 6 include those optimizations.

Table 7: HAETAE scheduling ablation with fixed kernels and arithmetic. Mean cycles over 10,000 executions; speedups (×) compare Ours with each reference row.

Operation	Implementation	HAETAE-2		HAETAE-3		HAETAE-5	
		Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
ARM (Apple M5)							
KeyGen	**Ours**	**588 392**	–	**1 096 355**	–	**1 383 956**	–
	Sequential	607 735	**1.03**	1 133 247	**1.03**	1 416 455	**1.02**
Sign	**Ours**	**1 202 666**	–	**1 539 637**	–	**2 229 894**	–
	Sequential	1 543 087	**1.28**	1 927 917	**1.25**	2 811 545	**1.26**
Verify	**Ours**	**78 027**	–	**127 645**	–	**157 777**	–
	Sequential	95 455	**1.22**	157 546	**1.23**	198 231	**1.26**
AVX2 (Core i5-11500)							
KeyGen	**Ours**	**392 102**	–	**704 547**	–	**769 138**	–
	Original grouping	405 629	**1.03**	716 047	**1.02**	791 640	**1.03**

Cortex-A72. We also evaluate the method on a Cortex-A72 using a Keccak backend that processes four hash streams per call through a combination of NEON and scalar instructions. The Apple M5 backend processes two streams per call using ARM SHA3 instructions. We compare the baseline and scheduled implementations using the same Keccak kernels and polynomial arithmetic.

Table 8 reports the results across all three parameter sets. ML-KEM achieves speedups of 1.10–1.17× for encapsulation and 1.15–1.23× for decapsulation; key generation achieves 1.02–1.08×. HAETAE achieves 1.02–1.08× for signing and 1.06–1.08× for verification.

Table 8: ML-KEM and HAETAE performance on Cortex-A72. Mean cycles over 10,000 executions; speedups (×) compare Ours with Baseline.

Operation	Implementation	ML-KEM-512		ML-KEM-768		ML-KEM-1024	
		Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
KeyGen	**Ours**	**58 309**	–	**92 949**	–	**141 488**	–
	Baseline	59 674	**1.02**	100 041	**1.08**	144 206	**1.02**
Encaps	**Ours**	**61 393**	–	**93 095**	–	**142 419**	–
	Baseline	68 507	**1.12**	108 995	**1.17**	157 156	**1.10**
Decaps	**Ours**	**73 741**	–	**113 039**	–	**172 407**	–
	Baseline	90 458	**1.23**	139 264	**1.23**	199 049	**1.15**
Operation	Implementation	HAETAE-2		HAETAE-3		HAETAE-5	
		Cycles	Speedup	Cycles	Speedup	Cycles	Speedup
KeyGen	**Ours**	**1 601 583**	–	**3 003 998**	–	**3 628 620**	–
	Baseline	1 601 679	1.00	3 013 349	1.00	3 629 255	1.00
Sign	**Ours**	**4 679 953**	–	**7 752 541**	–	**11 362 576**	–
	Baseline	5 060 714	**1.08**	7 948 760	**1.03**	11 537 253	**1.02**
Verify	**Ours**	**198 669**	–	**342 228**	–	**424 928**	–
	Baseline	211 864	**1.07**	369 708	**1.08**	449 727	**1.06**

Signing attempts. Figure 4 compares the official release, Xu et al.'s implementation [XHPL26], and ours on the i5. All use the same keys, 64-byte messages, and

32-byte digests. This separate diagnostic executes forced prefixes of exactly 1–12 attempts, continuing after an early acceptance. Each point reports the mean of 10,000 individual calls. Table 6 reports valid signatures with natural retries.

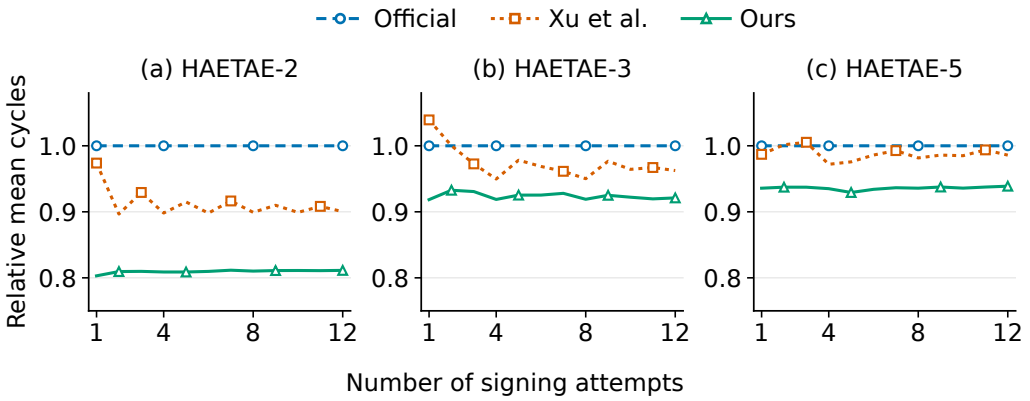

Figure 4: Signing costs relative to the official implementation at the same number of attempts: (a) HAETAE-2, (b) HAETAE-3, and (c) HAETAE-5. Lines show ratios of mean cycles over 10,000 individually timed forced execution prefixes (official = 1).

Across all 12 attempt counts, our implementation uses 18.84–19.70%, 6.73–8.16%, and 6.12–7.07% fewer cycles than the official release for HAETAE-2, -3, and -5. Relative to Xu et al., the reductions are 9.72–17.54%, 3.26–11.62%, and 3.80–6.77%.

Dilithium. We also apply the method to Dilithium [DKL+18], using Xu et al.'s optimized implementation [XHPL26]. AVX2 speedups remain close to one (0.99–1.04×). On AVX-512, KeyGen achieves 1.24–1.43× speedups (Table 9).

Table 9: Dilithium AVX-512 KeyGen: initial calls, cycles, and speedups.

Parameter	Original calls	Scheduled calls	Xu et al.	Ours	Speedup (×)
2	$11P_1 + 22P_4$	$11P_1 + 11P_8$	61 683	49 580	**1.24**
3	$16P_1 + 46P_4$	$4P_1 + 25P_8$	109 393	76 476	**1.43**
5	$21P_1 + 74P_4$	$6P_1 + 40P_8$	170 596	124 723	**1.37**

5.3 Model-based Schedule Quality

Table 10 reports the permutation-cost gaps and modeled cost reductions of the 45 selected schedules.[1] 40 attain the permutation-cost lower bound, including every KeyGen case. The five positive gaps occur in AVX-512 encapsulation and AVX2 signing; 44 schedules have gaps at most 5%.

Permutation cost alone does not determine the choice of schedule. An alternative ML-KEM-1024 Encaps schedule on AVX-512 saves 519 permutation cycles but adds 645 state-movement cycles relative to the selected schedule. Regrouping must preserve partial matrix output in buffers associated with the state slots.

HAETAE AVX2 Sign leaves matrix streams at the ends of the selected groups, exceeding the permutation-cost lower bound by $2\pi_{p,4}$, $\pi_{p,4}$, and $\pi_{p,4} - \pi_{p,1}$ for parameters 2, 3, and 5.

[1]The HAETAE Sign model counts the initial hash requests of a single signing attempt.

Table 10: Permutation-cost gaps and PMI cost reductions relative to s_0 (Equation 17). Counts give permutation steps at each SIMD width, including steps within multi-step functions. Bold zeros attain the permutation-cost lower bound.

	Apple M5 ($w = 2$)			AVX2 ($w = 4$)			AVX-512 ($w = 8$)		
Scheme / operation	n_1/n_2	P-gap (%)	r_{PMI} (%)	n_1/n_4	P-gap (%)	r_{PMI} (%)	n_1/n_8	P-gap (%)	r_{PMI} (%)
ML-KEM-512 KeyGen	7/10	**0.00**	0.68	7/5	**0.00**	0.92	7/3	**0.00**	17.94
ML-KEM-512 Encaps	0/13	**0.00**	26.53	3/6	**0.00**	21.36	6/3	**0.00**	31.97
ML-KEM-512 Decaps	0/16	**0.00**	31.82	1/8	**0.00**	36.95	3/6	**0.00**	46.73
ML-KEM-768 KeyGen	11/16	**0.00**	7.58	11/8	**0.00**	8.55	11/4	**0.00**	6.28
ML-KEM-768 Encaps	0/22	**0.00**	23.12	1/11	**0.00**	27.62	4/7	6.54	28.32
ML-KEM-768 Decaps	1/26	**0.00**	27.74	1/13	**0.00**	36.97	2/9	**0.00**	49.27
ML-KEM-1024 KeyGen	13/28	**0.00**	0.76	13/14	**0.00**	1.18	13/7	**0.00**	0.00
ML-KEM-1024 Encaps	0/35	**0.00**	15.98	0/18	**0.00**	20.66	6/9	2.33	24.48
ML-KEM-1024 Decaps	0/41	**0.00**	23.13	0/21	**0.00**	30.29	3/12	**0.00**	47.24
HAETAE-2 KeyGen	2/18	**0.00**	48.94	2/9	**0.00**	29.68	–	–	–
HAETAE-2 Sign	5/120	**0.00**	48.17	5/65	3.03	26.19	–	–	–
HAETAE-2 Verify	5/20	**0.00**	45.17	5/10	**0.00**	36.21	–	–	–
HAETAE-3 KeyGen	1/40	**0.00**	50.18	1/20	**0.00**	11.16	–	–	–
HAETAE-3 Sign	8/191	**0.00**	48.04	7/97	1.00	15.68	–	–	–
HAETAE-3 Verify	7/42	**0.00**	45.80	7/21	**0.00**	23.26	–	–	–
HAETAE-5 KeyGen	1/53	**0.00**	50.23	1/27	**0.00**	3.50	–	–	–
HAETAE-5 Sign	10/238	**0.00**	48.12	9/120	0.31	11.70	–	–	–
HAETAE-5 Verify	9/56	**0.00**	46.34	9/28	**0.00**	20.70	–	–	–

Fixed initial hash requests. HAETAE: 64-byte messages for Sign and Verify. Dashes denote unsupported backends.

Full-cost reference comparison. For nine ML-KEM AVX2 schedules, we evaluate g_{cost} with fixed requests, input formats, layouts, two working buffers, and a 256 KiB logical storage limit. Only priced preparation calls are allowed. Replay checks physical calls and cost. The reference pool combines the selected schedule, generic constructions, and searches with 64- and 256-state frontiers using Section 3.3's action generation. The larger search permits 100,000 expansions and 10 seconds; planning takes 0.96–8.91 seconds on Apple M5 under macOS. All nine gaps are zero: no cheaper schedule was found in this priced action domain.

Using $\widehat{T}_p$ from Section 3.2, we define the $P_p + M_p + I_p$ (PMI) cost reduction as

$$r_{\mathrm{PMI}}(s) = 1 - \frac{\widehat{T}_p(s)}{\widehat{T}_p(s_0)}, \tag{17}$$

Here s_0 retains the original grouping and call order, including SIMD batching. For HAETAE on Apple M5, s_0 executes streams sequentially with the same SHA3 kernels. r_{PMI} measures the percentage reduction in modeled hash-scheduling cost.

Figure 5 confirms that measured gains follow the PMI trend. Runtime reductions use $1 - 1/\text{speedup}$ with the unrounded speedups in Table 6.

6 Discussion and Conclusion

Correctness analysis. Regrouping leaves stream outputs unchanged (Section 3.1). All modified implementations pass correctness checks and known-answer tests (KATs).

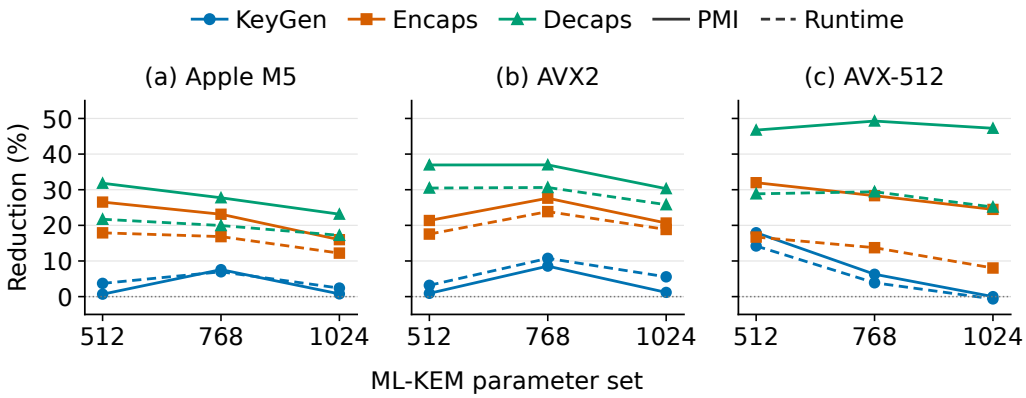

Figure 5: PMI and runtime reductions for ML-KEM on (a) Apple M5, (b) AVX2, and (c) AVX-512.

TIMECOP analysis. We examine secret-dependent branches and memory accesses in AVX2 ML-KEM and HAETAE using TIMECOP-style checks [Nei19] with Valgrind Memcheck. ML-KEM produces no warnings on the tested paths. For HAETAE, all observed secret-dependent branches and memory accesses on the tested scheduled paths arise from mechanisms already present in the original implementation. For ARM and AVX-512, which are not covered by our dynamic checking setup, we use targeted manual source review to check scheduling changes for additional secret-dependent control flow and memory accesses.

We presented a cost-aware method for organizing SHA3 and SHAKE computations within a single cryptographic operation. Grouping ready streams and regrouping segments of long streams reduces unused SIMD work, while state preparation, retention, and output delivery make these schedules efficient on different backends. The implementations achieve speedups of up to 1.31× for ML-KEM encapsulation and 1.44× for decapsulation. With identical kernels and arithmetic, HAETAE scheduling achieves up to 1.28× for signing and 1.26× for verification on Apple M5. These results show that the organization of hash computations is an effective optimization target alongside the Keccak kernels themselves: stream groups, segment lengths, and state placement should be chosen together to reduce the cost of producing the required hash output.

References

[ABD+21] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: Algorithm Specifications and Supporting Documentation. NIST Post-Quantum Cryptography Standardization, Round 3 submission, version 3.02, August 2021. https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.

[BC23] Gustavo Banegas and Florian Caullery. Multi-Armed SPHINCS+. In *ACNS Workshops 2023*, 2023.

[BDK+26] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Kyber: Maintenance Status. Software repository documentation, 2026. https://github.com/pq-crystals/kyber#maintenance-status. Accessed 2026-10-14.

[BDP+12] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview. Technical Report Version 3.2, The Keccak Team, May 2012. https://keccak.team/files/Keccak-implementation-3.2.pdf.

[BHK+22] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, 2022. https://doi.org/10.46586/tches.v2022.i1.221-244.

[BK22] Hanno Becker and Matthias J. Kannwischer. Hybrid Scalar/Vector Implementations of Keccak and SPHINCS+ on AArch64. In Takanori Isobe and Santanu Sarkar, editors, *Progress in Cryptology – INDOCRYPT 2022*, volume 13774 of *Lecture Notes in Computer Science*, pages 272–293. Springer, 2022. https://doi.org/10.1007/978-3-031-22912-1_12.

[CCD+24] Jung Hee Cheon, Hyeongmin Choe, Julien Devevey, Tim Güneysu, Dongyeon Hong, Markus Krausz, Georg Land, Marc Möller, Damien Stehlé, and MinJune Yi. HAETAE: Shorter Lattice-Based Fiat-Shamir Signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):25–75, 2024. https://doi.org/10.46586/tches.v2024.i3.25-75.

[CL14] Roberto Cabral and Julio López. Software implementation of SHA-3 family using AVX2. In *Anais do XIV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg 2014)*, pages 330–333. Sociedade Brasileira de Computação, 2014. https://sol.sbc.org.br/index.php/sbseg/article/download/20146/19974.

[CL18] Roberto Cabral and Julio López. Implementation of the SHA-3 family using AVX512 instructions. In *Anais do XVIII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg 2018)*, pages 25–32. Sociedade Brasileira de Computação, 2018. https://doi.org/10.5753/sbseg.2018.4266.

[DKL+18] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268, 2018. https://doi.org/10.46586/tches.v2018.i1.238-268.

[HAE23] HAETAE team. HAETAE v2.0. Software release, version 2.0, 2023. https://kpqc.cryptolab.co.kr/haetae. Released 2023-11-29; accessed 2026-10-16.

[HAZ+24] Junhao Huang, Alexandre Adomnicăi, Jipeng Zhang, Wangchen Dai, Yao Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Revisiting Keccak and Dilithium Implementations on ARMv7-M. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):1–24, 2024. https://doi.org/10.46586/tches.v2024.i2.1-24.

[Kor25] Korean Post-Quantum Cryptography Research Group. Selected Algorithms from the KpqC Competition Round 2, January 2025. https://kpqc.or.kr/competition_02.html. Announcement dated 2025-01-16; accessed 2026-10-15.

[LHP+23] Douwei Lei, Debiao He, Cong Peng, Min Luo, Zhe Liu, and Xinyi Huang. Faster Implementation of Ideal Lattice-Based Cryptography Using AVX512. *ACM Transactions on Embedded Computing Systems*, 22(5):83:1–83:18, 2023. https://doi.org/10.1145/3609223.

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, 2009. https://doi.org/10.1007/978-3-642-10366-7_35.

[Nat15] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Federal Information Processing Standards Publication 202, National Institute of Standards and Technology, Gaithersburg, MD, USA, August 2015. https://doi.org/10.6028/NIST.FIPS.202.

[Nat24] National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standards Publication 203, National Institute of Standards and Technology, Gaithersburg, MD, USA, August 2024. https://doi.org/10.6028/NIST.FIPS.203.

[Nei19] Moritz Neikes. TIMECOP: Automated Dynamic Analysis for Timing Side-Channels. Software and documentation, 2019. https://www.post-apocalyptic-crypto.org/timecop/. Accessed 2026-10-16.

[Ope26] Open Quantum Safe project. liboqs. Software repository, 2026. https://github.com/open-quantum-safe/liboqs. Commit b196b57aa615; accessed 2026-10-06.

[pq-26a] pq-code-package contributors. mldsa-native. Software repository, 2026. https://github.com/pq-code-package/mldsa-native. Commit 9b0ee84f4cf3; accessed 2026-10-06.

[pq-26b] pq-code-package contributors. mlkem-native. Software repository, 2026. https://github.com/pq-code-package/mlkem-native. Commit 1d7b486c4db3; accessed 2026-10-06.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, Paper 2018/039, 2018. https://eprint.iacr.org/2018/039.

[Sin19] Sujoy Sinha Roy. SaberX4: High-Throughput Software Implementation of Saber Key Encapsulation Mechanism. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 321–324. IEEE, 2019. https://doi.org/10.1109/ICCD46524.2019.00050.

[SLL+26] Shi Ya, Liu Bingqian, Lu Xianhui, Qian Wenfei, Liu Ying, and Wang Kunpeng. Parameter-Aware and Instruction-Driven Dilithium Optimization on AVX2 and NEON. Cryptology ePrint Archive, Paper 2026/1272, 2026. https://eprint.iacr.org/2026/1272.

[SLS24] Minjoo Sim, Minwoo Lee, and Hwajeong Seo. HAETAE on ARMv8. *Electronics*, 13(19):3863, 2024. https://doi.org/10.3390/electronics13193863.

[The26] The Keccak Team. XKCP: eXtended Keccak Code Package. Software repository, 2026. https://github.com/XKCP/XKCP. Accessed 2026-10-06.

[wol26] wolfSSL Inc. wolfSSL. Software repository, 2026. https://github.com/wolfSSL/wolfssl. Commit 2411aae3f74d; accessed 2026-10-06.

[XHPL26] Runqing Xu, Debiao He, Cong Peng, and Min Luo. Hybrid Parallel SHA-3 and its Applications to Fiat-Shamir with Aborts Signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(4):679–704, 2026. https://doi.org/10.46586/tches.v2026.i4.679-704.

[YZH+25] Xuan Yu, Jipeng Zhang, Junhao Huang, Donglong Chen, and Lu Zhou. Multi-way high-throughput implementation of Kyber. In Nicky Mouha and Nick Nikiforakis, editors, *Information Security – ISC 2024, Part II*, volume 15258 of *Lecture Notes in Computer Science*, pages 41–60. Springer Nature Switzerland, 2025. https://doi.org/10.1007/978-3-031-75764-8_3.

[ZYHK25] Jipeng Zhang, Yuxing Yan, Junhao Huang, and Çetin Kaya Koç. Optimized Software Implementation of Keccak, Kyber, and Dilithium on RV{32,64}IM{B}{V}. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):632–655, 2025. https://doi.org/10.46586/tches.v2025.i1.632-655.