AcclMT: A Highly Resource-Efficient and Flexible Poseidon Hash-Based Merkle Tree Architecture

Changxu Liu[12], Hao Zhou[12], Lan Yang[12], Yifei Feng[3], Zheng Wu[12], Zhuoyuan Yang[12], Yinlong Li[4], Shiyong Wu[4*] and Fan Yang[12*]

[1] *State Key Laboratory of Integrated Chips and Systems, Fudan University, Shanghai, China*
[2] *School of Microelectronics, Fudan University, Shanghai, China*
[3] *School of Computer Science, Fudan University, Shanghai, China*
[4] *Hardware R&D Center, Shanghai Academy of Future Internet Technology, Shanghai, China*

2025 62nd ACM/IEEE Design Automation Conference (DAC) | 979-8-3315-0304-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/DAC63849.2025.11132911

***Abstract*—Merkle Tree is a fundamental cryptographic primitive in Zero-Knowledge Proof (ZKP) protocols, sharing significant computational workloads with the Number Theoretic Transform (NTT) in zk-STARK schemes. Merkle Tree is a tree structure where nodes are primarily generated through hash computations. Among them, Poseidon Hash, as a ZK-friendly hash function, has emerged as one of the most widely adopted choices. Therefore, hardware acceleration of building Merkle Tree based on Poseidon Hash can significantly enhance the performance of ZKP protocols. We propose AcclMT, a highly resource-efficient and flexible Poseidon Hash-based Merkle Tree architecture. Our design employs hardware-software co-design and optimizes the hashing data flow, resulting in an area-efficient Poseidon Hash engine that improves modular multiplication resource utilization. Furthermore, AcclMT uses these engines alongside hierarchical on-chip cache and optimized task scheduling for building large Merkle Trees. It also supports flexible parameter configurations for various requirements. Experimental results show that our proposed Poseidon Hash engine achieves a 14.3 $\times$ speedup compared to the latest FPGA-based work. By improving resource utilization, it also reduces area usage by 14.8% compared to unoptimized design. AcclMT achieves up to 1665 $\times$ speedup over software implementations in building Merkle tree, with average utilization of 95.9% and 99.2% for the two hash engines.**

***Index Terms*—Zero-Knowledge Proof, Merkle Tree, Poseidon Hash, Hardware Acceleration**

I. INTRODUCTION

The Merkle Tree is a key primitive in Zero-Knowledge Proof (ZKP), a commonly used privacy-preserving algorithm. It is often utilized in zk-STARK-based ZKP [1] schemes for polynomial commitments. Unlike the Multi-Scalar Multiplication-based commitment schemes in zk-SNARK [2], Merkle Trees offer higher computational and spatial efficiency. When building a Merkle Tree, a large sequence of elements is first hashed to generate leaf nodes, which are then repeatedly hashed and combined layer by layer until the root node is obtained. The trend has shifted toward choosing ZK-friendly arithmetic hash functions like Poseidon Hash [3]. Unlike traditional hashes such as SHA, Poseidon Hash offers higher computational efficiency and fewer constraints, making it widely applied in various scenarios [4]–[6]. In zk-STARK and some newer zk-SNARK schemes, hash functions—primarily in the form of Merkle Trees—represent the largest computational workload alongside the Number Theoretic Transform, with the two accounting for over 65% of total computation [7]. Poseidon Hash, being more computationally expensive, may further increase this proportion. Previously, hardware acceleration efforts for ZKP mainly focused on Multi-Scalar Multiplication (MSM) [8]–[11] and the Number Theoretic Transform (NTT) [10]–[12], while neglecting improvements in hash computations and Merkle Tree. According to Amdahl's Law [13], exclusively accelerating MSM and NTT would leave ZK system performance severely bottlenecked by the unoptimized hash operations, which dominate the remaining computation overhead [14].

However, building a Poseidon Hash-based Merkle Tree presents significant challenges. One is efficiently managing the computational workload, as a single Poseidon Hash may require hundreds or thousands of modular multiplications, while a Merkle Tree can involve millions of such computations. Another challenge lies in handling the vast, temporally sparse intermediate data generated during computation. Despite their space efficiency, Merkle Trees can produce several gigabits of intermediate data, whose sparse distribution reduces computational unit utilization. Keeping extensive intermediate data on-chip is expensive, while frequent transfers to off-chip storage further impact efficiency.

To address these challenges, accelerating Poseidon Hash-based Merkle Tree implementations becomes crucial. ZPrize [15] introduces a track for Poseidon Hash hardware acceleration, and Ingonyama develops ICICLE [16], a cryptographic library optimized for hardware platforms, including Poseidon Hash-based Merkle Trees. TRIDENT [17], a Poseidon Hash accelerator for FileCoin on FPGA, delivers significant performance gains over advanced CPUs. Irreducible also proposes the first fully pipelined FPGA architecture for ZKP-friendly Merkle Trees using Poseidon Hash [18].

However, we believe that the above works have limitations that make them unsuitable as reference paradigms or lack practical considerations, as detailed below:

- **Lack of Flexibility and Generalizability:** Their implementations are often tailored to specific protocols, such as Irreducible's approach [18] for Plonky2 and TRIDENT [17] for the FileCoin protocol.
- **Low Resource Utilization:** The most expensive modular multipliers are not fully utilized. We estimate that nearly half of TRIDENT's [17] modular multipliers remain idle most of the time, resulting in reduced area efficiency.
- **Imbalanced Performance and Resource Overhead, with Limited Optimization Strategies:** Irreducible's approach [18] fully unrolls the Poseidon Hash for optimal performance in 64-bit Goldilocks fields. However, for ZKP protocols using larger finite fields, the hardware cost of full unrolling becomes unsustainably high.

In this paper, we present AcclMT, a highly resource-efficient and flexible Poseidon Hash-based Merkle Tree architecture. Our proposed flexible design supports the configuration of several typical parameters, catering to diverse application contexts. We optimize the proposed Poseidon Hash engine through software-hardware co-design, efficiently allocating different types of Poseidon permutations to two

*Corresponding author: yangfan@fudan.edu.cn, wusy@safit.org.cn

sub-engines, achieving higher resource utilization than unoptimized architectures. In Merkle Tree optimization, our approach efficiently decomposes and schedules tasks, using a limited hierarchical on-chip cache to build Merkle Trees of various sizes. Although our design uses a 256-bit data width, it can be adjusted for specific applications. Our key contributions are:

- We propose a Poseidon Hash engine combining a resource-efficient partial-round engine with a resource-intensive full-round engine. We optimize the scheduling of different rounds in Poseidon Hash across two sub-engines, reducing resource consumption and improving modular multiplier utilization. Compared to an unoptimized design with two full-round engines, our approach reduces area by 14.8% with just 7.04% extra time overhead and achieves a 14.3 $\times$ speedup over FPGA design.
- We develop a Merkle Tree architecture with efficient on-chip cache management and high resource utilization. Large-size Merkle Trees are broken into standard tasks requiring minimal cache for hierarchical processing. By efficiently managing around 200KB of limited on-chip cache, our design supports building large-size Merkle Trees. Merging multiple standard tasks further improves Poseidon Hash engine utilization. AcclMT can achieve up to 1665 $\times$ speedup over CPU-based implementation with an average Poseidon Hash utilization of 97.6%.
- Our proposed design natively supports flexible configuration of key parameters for the Poseidon Hash engine and Merkle Tree, enhancing its versatility across a broader range of applications.

II. Background

A. Poseidon Hash Function

Poseidon is a novel arithmetic hash function designed for the ZKP algorithm. While its native computation is more expensive than traditional hash functions like SHA, it offers greater efficiency in circuits, resulting in fewer constraints and reduced complexity for both provers and verifiers. Poseidon Hash employs a sponge/squeeze structure to map arbitrary-length strings over F_p to a fixed-length output, using iterations of the Poseidon permutation. The input, state (S), is divided into rate (R) and capacity (C) bits, with R-sized blocks absorbed into the sponge state. In the squeeze state, part of the state is output as the hash digest. The Poseidon permutation consists of a full round for higher security or a partial round for greater efficiency, with the number of rounds varying by scenario. We present an overview of the Poseidon Hash computation data flow in Fig. 1. In both cases, the algorithm includes three stages: adding round constants (ARC); an S-box layer for nonlinear diffusion using low-degree polynomials; and maximum-distance separable (MDS) matrix multiplication. In partial rounds, all S-box layers except the first are replaced with identity functions.

B. Merkle Tree in zk-STARK

A Merkle Tree (MT) is a tree structure in which each node is constructed using a collision-resistant hash function. It commits to an element sequence, allowing verification of an element's presence. The leaf nodes are obtained by hashing the elements, while non-leaf nodes are derived by concatenating and hashing their child nodes. The root node of the MT serves as a commitment to the sequence, as finding different sequences with the same root is computationally infeasible. Fig. 1 shows a simple schematic of the MT. We focus on binary MT (arity = 2), where each non-leaf node has exactly two child nodes, which is a common structure, though our proposed architecture can be configured to higher arity structures.

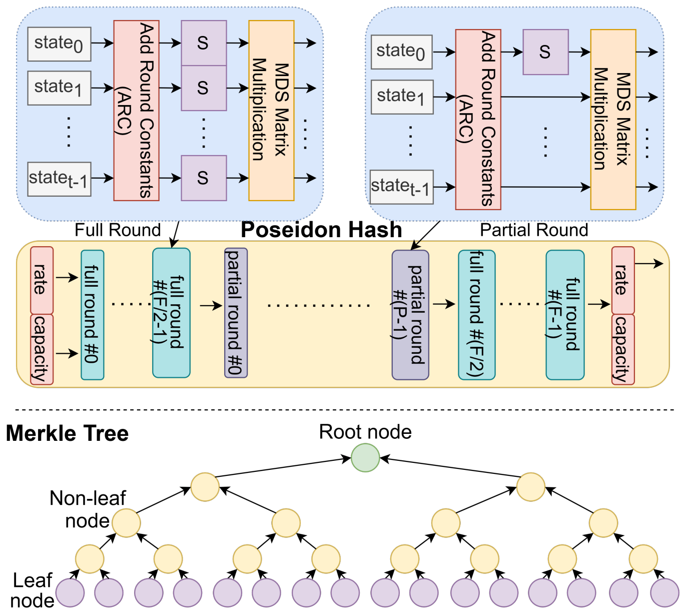

Fig. 1: Flowchart of Poseidon Hash (top) and Merkle Tree (down). For the Poseidon Hash, we assume F full rounds and P partial rounds. The case example features a 4-layer, size-16 Merkle Tree with an arity of 2.

In the zk-STARK protocol, MT is used for polynomial commitments. Fig. 2 outlines the proof generation phase, starting with converting the program into computational integrity (CI) declarations, followed by representing them as polynomials, including trace and constraint polynomials. Multiple constraint polynomials are then combined into a composition polynomial (CP), extended using the low-degree extension (LDE), and evaluated/interpolated via NTT/INTT. The composition and trace polynomials are then committed on the extended domain using MT. After the Deep operation, the polynomial undergoes low-degree testing using the FRI method. During the polynomial folding process, MT is used to commit to the polynomial at each stage. Additionally, this process encompasses querying nodes within the MT and the BCS method [19]. As these topics are not central to this paper, they are not detailed further.

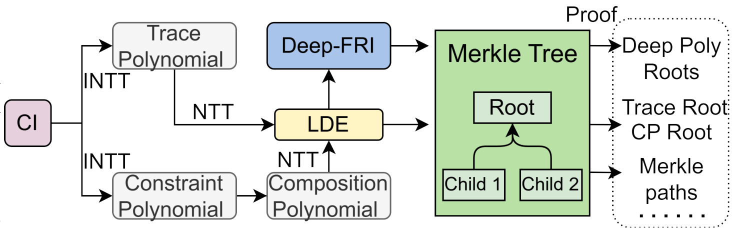

Fig. 2: Flowchart of the Proof Generation Phase in a Typical zk-STARK Protocol.

III. Proposed Design

In this section, we first present the overall architecture of AcclMT. Subsequently, we detail the microarchitecture of the Poseidon Hash engine built with two sub-engines, along with the optimizations of inter-sub-engine data flow. Finally, we present our insights at the Merkle Tree and optimization strategies for on-chip cache and scheduling mechanisms.

A. Overall Architecture

Our proposed design, AcclMT, consists primarily of two Poseidon Hash engines along with several on-chip caches for specific purposes and a dispatch scheduler. Fig. 3 illustrates the overall architecture of AcclMT. The leaf hash engine, a Poseidon Hash engine, receives the raw element sequence and performs hash computations, storing the leaf nodes in the leaf hash buffer. The root hash engine, also a Poseidon Hash engine, constructs the non-leaf nodes through concatenation and hashing, ultimately deriving the root node of the Merkle Tree. The root hash buffer, scratch pad, and hierarchical intermediate data memory store the intermediate results of the root hash engine or provide inputs to it, with the dispatch scheduler managing the scheduling and distribution of inputs and outputs for the root hash engine. Each of these data storage units has distinct roles, and we will elaborate on the employed design strategies in Sec. III-C.

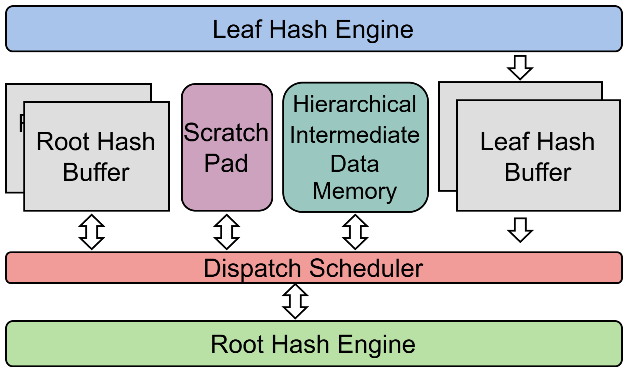

Fig. 3: Overall Architecture of AcclMT.

B. Microarchitecture of Poseidon Hash Engine

We have examined the typical parameters within the Poseidon Hash function, which mainly include the exponent in the S-box (the S-box typically utilizes an exponential function for nonlinear diffusion), the number of states in the Poseidon Hash, and the number of full rounds and partial rounds. Our goal is to develop a highly efficient and flexible Poseidon Hash engine with configurable parameters to enhance versatility.

To achieve this, we first observe the range of these typical parameters and make the following simple insights, focusing on hardware feasibility rather than delving into mathematical principles:

1) The exponent in the S-box is generally greater than or equal to 3, with 3 being almost the most common choice. Some applications may opt for 5 or 7 [3].
2) The number of states in the Poseidon Hash, which is arity + 1, often tends to be multiples of 3. Typical values for arity include 2, 4, 8, and 11 [3], [17].
3) Although the number of full and partial rounds varies across different applications, we observe that the number of partial rounds is generally significantly larger than that of full rounds.

Given these insights, we see great potential for optimizing the Poseidon Hash engine in hardware. Our solutions are as follows: For Insight 1), we design a configurable S-box with three modular multipliers for exponents 3 and 5, and it can also handle exponent 7 with an Initial Interval (II) of 2; higher exponents are not supported due to their rarity. For Insight 2), the Poseidon Hash engine, starting at a state of 3, leverages the vector multiplication and self-adder to support up to the state of 12, accommodating various arities. For Insight 3), we develop a resource-intensive full-round engine for full rounds in full mode and an area-efficient partial-round engine for partial rounds; the full-round engine can also operate in partial mode, enabling both engines to prioritize partial rounds, in line with our observations on round quantities.

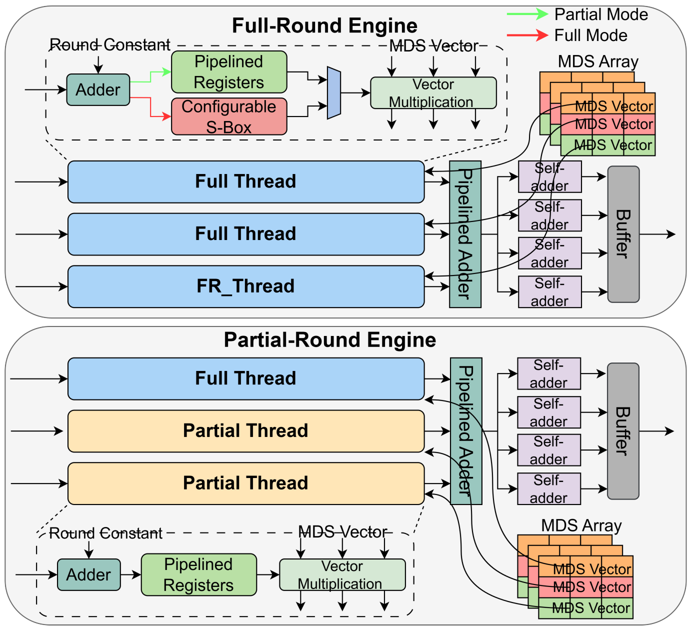

Fig. 4: Microarchitecture of Two Types of Sub-engines in Poseidon Hash Engine. The partial thread utilizes half the modular multiplication resources of the full thread. The full-round engine supports both partial-round and full-round computation, indicated by different colored arrows.

Fig. 4 displays the microarchitectures of our proposed full-round and partial-round engines. The partial-round engine includes two partial threads and one full thread, while the full-round engine comprises three full threads. The partial thread, lacking an S-box, uses half the modular multiplication modules of the full thread. The partial-round engine is dedicated to partial-round computations, while the full-round engine supports full-round computations in full mode and partial-round computations in partial mode. The sub-engine natively supports Poseidon permutation with state = 3. For higher states, an outer product algorithm improves vector-matrix multiplication efficiency. Each thread processes MDS vectors derived from the MDS matrix, and results are aggregated via an adder tree. The self-adder unit handles accumulation and outputs longer vectors in chunks for higher states.

We improve the scheduling mechanism for the two sub-engines to develop the Poseidon Hash engine. It can support arbitrary numbers of full and partial rounds by reusing the full-round engine and partial-round engine through iterative looping. Optimizing the data flow between sub-engines results in higher area efficiency and resource utilization for the Poseidon Hash engine. Fig. 5 compares our Poseidon Hash engine's optimized scheduling with an unoptimized design that uses two full-round engines, as in TRIDENT [17]. The unoptimized design keeps most S-boxes bypassed in partial mode (where partial rounds dominate), leading to considerable modular

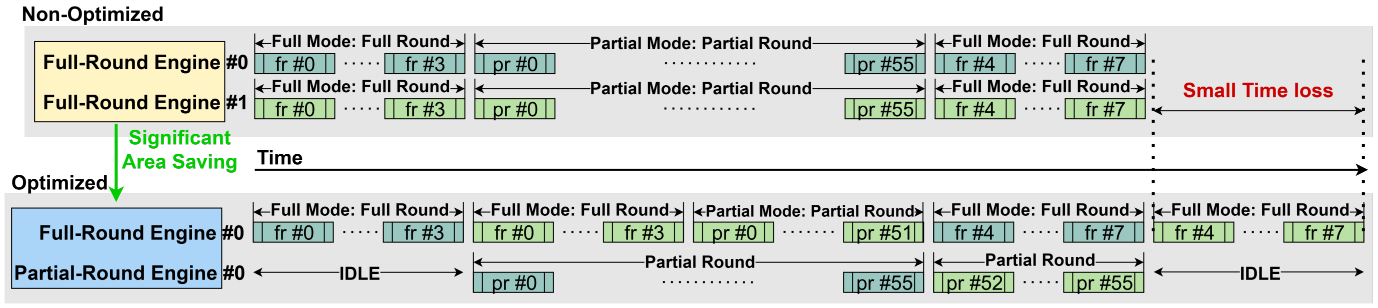

Fig. 5: Optimized Data Flow of Poseidon Hash Engine with Dual Sub-engines. We use typical parameters, with 8 full rounds and 56 partial rounds, as a case. Two different colors are used to illustrate the data flow of Poseidon Hash from different batches. 'pr' stands for partial round, 'fr' represents full round, and the symbol # followed by a number indicates the round sequence number. The partial-round engine consumes only $\frac{2}{3}$ of the modular multiplication resources compared to the full-round engine.

multiplication waste. We find it inefficient to allocate excessive resources to infrequent full rounds. So we assign more full rounds to the full-round engine and allow the more efficient partial-round engine to focus solely on partial rounds, reducing the number of modular multiplication modules.

In our scheduling mechanism, the Poseidon Hash data is divided into two batches. The full-round engine first performs half of the full rounds for both batches, then switches to partial mode for the second batch's partial rounds, while the partial-round engine completes the first batch's partial rounds. The partial-round engine then processes the second batch, with the full-round engine finishing the remaining full rounds in full mode. While the partial-round engine incurs brief idle times, adding minimal latency due to the round count difference, it provides significant area savings, as detailed in Sec. IV-B.

C. Optimization for Merkle Tree

We utilize two Poseidon Hash engines as the Leaf Hash Engine (LHE) and Root Hash Engine (RHE), along with a limited on-chip cache, to construct AcclMT. We observe that during the building of Merkle Tree, two key issues need to be addressed:

1) Merkle Tree sizes are often variable, with larger trees requiring more input and intermediate data. However, relying solely on larger intermediate caches is both inefficient and impractical.
2) The process of building the Merkle Tree involves aggregating multiple child nodes into a parent node. At higher layers of the tree, data distribution becomes sparser, leading to underutilization of the Poseidon Hash engine and reduced computational efficiency.

To address the first challenge, we propose the following solution: as shown in Fig. 6, We use a 128KB Leaf Hash Buffer for LHE-processed values and a 64KB Root Hash Buffer for RHE intermediate results, with a 256-bit data width. These buffers fully support a 4096-size Merkle Tree with arity = 2. To handle larger Merkle Trees in various applications, we decompose them into groups of standard Merkle Trees, each sized at 4096. Instead of computing the entire tree at once, which generates and requires storing gigabits of intermediate data, we pipeline each standard Merkle Tree group, reducing the on-chip cache to about 100KB for a single standard Merkle Tree's intermediate data. The root of the standard Merkle Tree will be stored in an 8.5KB hierarchical intermediate data memory (HIDM). HIDM has two levels: Level 0 stores roots of standard Merkle Trees, which are aggregated when full (corresponding to Merkle Tree layers 13 to 19), with the new root stored in Level 1. When Level 1 is full, a final aggregation is performed for layers 20 to 23, with the final root

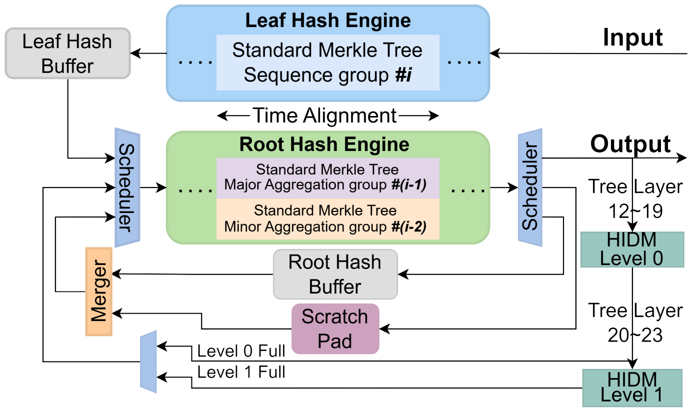

Fig. 6: Optimized Computational Flow of the Merkle Tree in AcclMT. AcclMT can leverage a limited on-chip cache and hierarchical intermediate data memory (HIDM) to efficiently build large-size Merkle Trees through an optimized task scheduling mechanism.

outputted. This hierarchical design supports building the Merkle Trees up to a size of 2^{23}. RHE processes each standard Merkle Tree group by reading data from the leaf hash buffer, and repeatedly writing and accessing intermediate results in the root hash buffer for incremental aggregation. The scheduler manages both RHE's data read and write-back operations.

To address key issue 2, we optimize the data scheduling mechanism within AcclMT. LHE efficiently handles compactly arranged input data, but as Merkle Tree layers deepen, RHE's inputs and outputs become sparser due to data dependencies, which limits its utilization and creates backpressure, reducing LHE's efficiency. We observe that RHE's pipeline utilization drops with each additional Merkle Tree layer; pipeline bubbles in RHE significantly increase by the fifth layer. As shown in Fig. 6, a 4KB scratch pad mitigates this: when aggregating leaf nodes for each standard Merkle Tree, we use root hash buffer as a data buffer, aggregating only up to 128 nodes, a process we call "major aggregation". The final step to reach the root of standard Merkle Tree is referred to as "minor aggregation". Proceeding directly with minor aggregation introduces pipeline bubbles, greatly reducing the efficiency of the Poseidon Hash engine. To improve this, we temporarily store the previous group's intermediate major aggregation results in a scratch pad and use a merger to combine these intermediate data with the next group's major aggregation sequence, effectively filling pipeline bubbles left by the major aggregation sequence in the RHE. This optimization substantially alleviates sparsity in the RHE pipeline, improving RHE utilization and balancing it with LHE, thereby creating an efficient computational pipeline. Fig. 7 illustrates the optimized data scheduling for the RHE, further boosting the computation efficiency for each standard Merkle Tree. In experiments, the pipeline data flow between RHE and LHE limits minor aggregation to the 11th layer, leaving only the final hash (12th layer). To avoid inefficiency from performing a single hash, both values are stored in HIDM's Level 0, which manages tree layers 12 to 19.

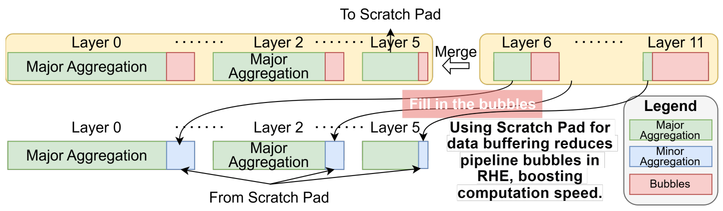

Fig. 7: Optimizing Data Scheduling for the RHE Pipeline. The pipeline combines the Merkle Tree's minor aggregations from the previous group with the major aggregations from the current group, significantly improving RHE utilization.

IV. Evaluation

A. *Experiment Setup & Overhead*

In this section, we present the implementation results and performance analysis of AcclMT. AcclMT is implemented in System Verilog HDL and is based on a typical 256-bit data width. The typical parameters in Poseidon Hash, including the S-box exponent, state size, the number of full and partial rounds, and the size of the Merkle Tree, are all configurable at runtime. We synthesize AcclMT using the TSMC 28nm process library and Synopsys Design Compiler, targeting a frequency of 500 MHz. The area report from the synthesis tool indicates AcclMT has a total area of 20.87 mm^2, with the two Poseidon Hash engines taking up 93.7%, and memory occupying about 1.31 mm^2, which is roughly 6.3%. Our Merkle Tree decomposition strategy allows handling several gigabits of data with just nearly 200KB of on-chip cache.

B. *Evaluation of the Poseidon Hash Engine*

We compare the performance of our proposed Poseidon Hash engine with existing FPGA-based work, TRIDENT [17], and CPU-based work, NEPTUNE [4]. Using the Poseidon Hash configurations from FileCoin [4] as benchmarks, we determine the specific parameters based on Ingonyama's tool [20]. Tab. I details the data, measuring performance by hashes computed per second.

Compared to FPGA-based work, our Poseidon Hash engine offers up to a 14.3 × speedup, achieving a throughput of 14.11M hashes per second. To ensure a fair comparison, excluding frequency differences and focusing solely on clock cycles, the maximum speedup is about 2.85 ×. Notably, TRIDENT uses two units equivalent to our proposed full-round engine. However, during partial round computation, its S-box modules are bypassed and left idle, resulting in lower resource utilization compared to our design. In contrast, We optimize utilization by efficiently distributing tasks between the full-round and partial-round engines, reducing modular multiplication resource consumption by 16.7%. NEPTUNE is an optimized Rust-based open-source implementation of Poseidon Hash for FileCoin. Compared to NEPTUNE, our design offers a speedup of up to 145 ×.

TABLE I: Comparison of Poseidon Hash Computation Speed: AcclMT vs. FPGA and CPU-based Works.

Work		Ours	TRIDENT [17]	NEPTUNE [4]
Platfrom		TSMC 28nm	Xilinx Varium C1100	Intel i7-13700F 32GB
Frequency		500 MHz	100 MHz	2.1 GHz
Hash Rate (hash / sec)	Arity = 2	**14.11M**	0.99M (**14.3 × / 2.85 ×**)[a]	97.00K (**145 ×**)
	Arity = 4	5.93M	-	57.67K (103 ×)
	Arity = 8	3.79M	358K (10.6 × / 2.12 ×)	30.29K (125 ×)
	Arity = 11	2.69M	305K (8.82 × / 1.76 ×)	20.96K (128 ×)

a: The left side of the parentheses indicates the time-based speed up, while the right side represents the clock cycle-based speedup.

Our speedup is optimal at arity = 2, while performance degrades at higher arity levels. This is because, in our Poseidon Hash engine, the two sub-engines support pipelined input and output at arity = 2, resulting in the smallest initiation interval (II). However, at higher arity, the sub-engines are no longer fully pipelined, requiring block processing for input and output, which increases the II and diminishes the overall speedup of the Poseidon Hash engine. Additionally, there is space for further improvements in the scheduling mechanism of the Poseidon Hash engine at higher arity.

Significant Reduction in Area: We conduct experiments to validate that our Poseidon Hash engine, built with a full-round engine and a partial-round engine, achieves higher area efficiency than the baseline with two full-round engines. Based on the four benchmarks from FileCoin shown in Tab. I, our proposal incurs an average time loss of 7.04% relative to the baseline, primarily due to the idling of the partial-round engine, as illustrated in Fig. 5. We also synthesize a baseline design using two full-round engines. Compared to this baseline, our proposed Poseidon Hash engine achieves a 14.8% reduction in area, which closely aligns with our theoretical estimate of 16.7% area reduction. Notably, this area savings is fixed, while the time loss diminishes as the ratio of partial rounds to full rounds increases. When the full round count is 8 and the partial round count is 120 [3], the time loss drops to only 3.55%, nearly negligible, while the reduction of area overhead remains at 14.8%.

C. *Evaluation of the Merkle Tree*

In Tab. II, we evaluate the performance of AcclMT in building Merkle Trees of various sizes, comparing it to CPU-based implementations with an arity = 2, a common parameter. The CSF '24 [21] uses the libff [23] library in C++ and builds Poseidon Hash-based Merkle Trees with sizes up to 2^{18}. We also utilize the Rust-based library from Dusk-network [22] for another benchmark.

The results show that AcclMT achieves up to a 245 × speedup over CSF '24 and a 1665 × speedup compared to Dusk-network. As the size of the Merkle Tree increases, the speedup of AcclMT becomes more pronounced. The execution time also includes initial data loading and final aggregation in HIDM. When the size of the Merkle Tree is small, the time to build the standard Merkle Tree with AcclMT becomes shorter, and the proportion of time spent on initialization and post-processing increases. As the Merkle Tree size increases, the latency from subsequent data loading is masked by the ping-pong mechanism of data transfer, and the proportion of time spent on data aggregation in the HIDM decreases. At larger sizes, the time for initialization and post-processing can be negligible. Thus, AcclMT's execution time grows sublinearly. Compared to CPU implementations, which exhibit linear growth, AcclMT demonstrates

TABLE II: Comparison of Time (ms) Spent by AcclMT and Other Works for Building Merkle Trees of Various Sizes.

Work	Platform	Merkle Tree Size											
		2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}
Ours	TSMC 28nm 500 MHz	0.91	1.22	1.83	3.04	5.45	10.3	19.8	39.0	77.4	154	308	615
CSF'24 [21]	Intel i9-13900KF @ 6.0 GHz	75.8 (83.3 ×)	152 (125 ×)	303 (166 ×)	607 (200 ×)	1213 (223 ×)	2426 (235 ×)	4852 (**245 ×**)	-	-	-	-	-
Dusk-network [22]	Intel Xeon Platinum 8358@ 2.6 GHz	498 (547 ×)	1.01 s (828 ×)	2.00 s (1093 ×)	4.00 s (1316 ×)	7.99 s (1466 ×)	16.0 s (1553 ×)	32.0 s (1616 ×)	64.0 s (1641 ×)	128 s (1654 ×)	256 s (1662 ×)	512 s (1662 ×)	1024 s (**1665 ×**)

greater advantages in building larger Merkle Trees, making it particularly suited for building large-size Merkle Trees.

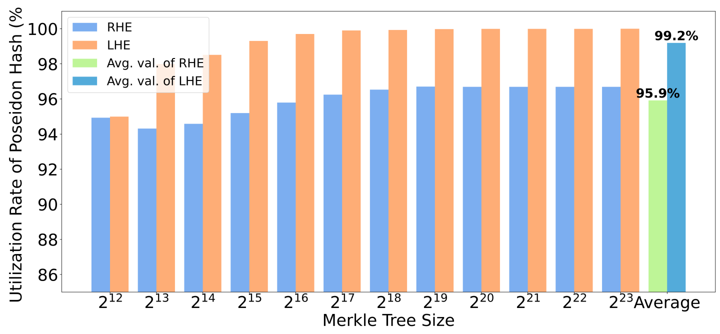

Fig. 8: Utilization Rate of the Leaf Hash Engine (LHE) and Root Hash Engine (RHE) in AcclMT for Building Merkle Trees of Various Sizes.

We also analyze the utilization rates of the LHE and RHE in AcclMT during the building of Merkle Trees of various sizes, as shown in Fig. 8. As the core component of AcclMT, one optimization goal is to improve the Poseidon Hash engine's utilization, which typically correlates with higher computational throughput. The utilization rates of LHE and RHE are quite similar, indicating that data organization for standard Merkle Tree computations is pipelined. Both engines maintain utilization rates above 95%, with RHE reaching 96.7% and LHE nearing 100% as the Merkle Tree size increases. This is expected, as RHE incurs more scheduling overhead, increasing latency. With larger Merkle Trees, both engines' utilization approaches saturation due to the higher computational workload of standard Merkle Tree computations. AcclMT achieves 99.2% and 95.9% average utilization for LHE and RHE, significantly exceeding the pre-optimization average of under 85%.

V. CONCLUSION

In this paper, we introduce AcclMT, a highly resource-efficient and flexible Poseidon Hash-based Merkle Tree architecture. We implement the Poseidon Hash engine through hardware-software co-optimization and data flow optimization, making it suitable not only for Merkle Trees but also for various purposes in ZKP. We also optimize the scheduling mechanism to ensure efficient acceleration of Merkle Tree computation using limited on-chip cache. The parameters of the Poseidon Hash engine and Merkle Tree size are configurable, offering AcclMT high flexibility for practical use in diverse ZKP scenarios.

ACKNOWLEDGMENTS

This work was supported in part by the National Key R&D Program of China under Grant 2023YFB2704600 and in part by the Science and Technology Commission of Shanghai Municipality under Project 24BC3201000. The authors would like to thank the reviewers for their valuable feedback.

REFERENCES

[1] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," Cryptology ePrint Archive, Paper 2018/046, 2018. [Online]. Available: https://eprint.iacr.org/2018/046

[2] J. Groth, "On the size of pairing-based non-interactive arguments," in *Proceedings, Part II, of the 35th Annual International Conference on Advances in Cryptology — EUROCRYPT 2016 - Volume 9666*. Berlin, Heidelberg: Springer-Verlag, 2016, p. 305–326.

[3] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for Zero-Knowledge proof systems," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 519–535. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/grassi

[4] Filecoin, "Neptune," https://github.com/argumentcomputer/neptune, accessed: 2024-10-31.

[5] Dusk-network, "Poseidon252," https://github.com/dusk-network/Poseidon252, accessed: 2024-10-31.

[6] Polygon, "Plonky2," https://github.com/0xPolygonZero/plonky2, accessed: 2024-10-27.

[7] Kelly Olson, "ZKP MOOC Lecture 16: Hardware Acceleration of ZKP," 2023, accessed: 2024-10-27. [Online]. Available: https://www.youtube.com/watch?v=ez46At3xTjM

[8] K. Aasaraai, D. Beaver, E. Cesena, R. Maganti, N. Stalder, and J. Varela, "FPGA acceleration of multi-scalar multiplication: CycloneMSM," Cryptology ePrint Archive, Paper 2022/1396, 2022. [Online]. Available: https://eprint.iacr.org/2022/1396

[9] C. Liu, H. Zhou, L. Yang, J. Xu, P. Dai, and F. Yang, "Gypsophila: A scalable and bandwidth-optimized multi-scalar multiplication architecture," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3658259

[10] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, "Pipezk: Accelerating zero-knowledge proof with a pipelined architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 416–428.

[11] T. Lu, C. Wei, R. Yu, C. Chen, W. Fang, L. Wang, Z. Wang, and W. Chen, "cuZK: Accelerating zero-knowledge proof with a faster parallel multi-scalar multiplication algorithm on GPUs," Cryptology ePrint Archive, Paper 2022/1321, 2022. [Online]. Available: https://eprint.iacr.org/2022/1321

[12] C. Wang and M. Gao, "Sam: A scalable accelerator for number theoretic transform using multi-dimensional decomposition," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, pp. 1–9.

[13] O. Shlomovits, "Revisiting paradigm hardware acceleration for zero knowledge proofs," https://www.ingonyama.com/blog/revisiting-paradigm-hardware-acceleration-for-zero-knowledge-proofs, 2023, accessed: 2024-10-29.

[14] H. Zhou, C. Liu, L. Yang, L. Shang, and F. Yang, "Rezk: A highly reconfigurable accelerator for zero-knowledge proof," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–14, 2024.

[15] ZPRIZE, “Accelerating the poseidon hash function,” https://www.zprize.io/prizes/accelerating-the-poseidon-hash-function, accessed: 2024-10-27.

[16] Ingonyama, “Icicle,” https://github.com/ingonyama-zk/icicle/tree/main, 2023, accessed: 2024-11-5.

[17] DatenLord, “Trident: Open-source project for high-performance storage solutions,” https://github.com/datenlord/TRIDENT, accessed: 2024-10-27.

[18] Irreducible, “Poseidon merkle trees in hardware,” https://www.irreducible.com/posts/poseidon-merkle-trees-in-hardware, 2023, accessed: 2024-10-27.

[19] E. Ben-Sasson, A. Chiesa, and N. Spooner, “Interactive oracle proofs,” Cryptology ePrint Archive, Paper 2016/116, 2016. [Online]. Available: https://eprint.iacr.org/2016/116

[20] Ingonyama, “poseidon-hash,” https://github.com/ingonyama-zk/poseidon-hash, accessed: 2024-11-1.

[21] E. Andreeva, R. Bhattacharyya, A. Roy, and S. Trevisani, “On efficient and secure compression modes for arithmetization-oriented hashing,” Cryptology ePrint Archive, Paper 2024/047, 2024. [Online]. Available: https://eprint.iacr.org/2024/047

[22] D. Network, “dusk-poseidon-merkle,” https://github.com/dusk-network/dusk-poseidon-merkle, 2023, accessed: 2024-10-27.

[23] scipr lab, “libff,” https://github.com/scipr-lab/libff, accessed: 2024-11-3.