On Analysis of Applications using MAGIC-based In-Memory Computing

Chandrajit Pal*, Lennart Weingarten†, Klaus D. McDonald-Maier*, Rolf Drechsler†,
Sangeet Saha* and Kamalika Datta†
*School of Computer Science and Electronic Engineering, University of Essex, Essex, United Kingdom.
{chandrajit.pal, kdm, sangeet.saha}@essex.ac.uk,
†Institute of Computer Science, University of Bremen, Bremen, Germany.
{len_wei, drechsler, kdatta}@uni-bremen.de

***Abstract*—Executing full applications on RRAM memristor-crossbars for in-memory computing (IMC) using the MAGIC design style is inhibited by the incompatibility between MAGIC's combinational requirement and the sequential logic (e.g., flip-flops) in standard Register-Transfer Level (RTL) generated from High-Level Languages (HLLs). This paper introduces the Application-to-Combinational (A2C) Flow, a novel synthesis methodology that resolves this by generating purely combinational, structural Verilog RTL directly from HLL code. The A2C Flow guarantees a single, stateless dataflow function by eliminating all sequential, behavioural, and iterative constructs through three key transformations: State Linearisation, Algorithmic Control-Flow Flattening, and Temporal-to-Spatial Unrolling. Thereafter, we validate the approach using an in-memory synthesis and verification methodology that assesses resource utilisation and verifies functional correctness against a golden reference using Boolean Satisfiability (SAT). This successfully removes sequential elements, bridging the gap to realise complex application-level kernels on MAGIC-based memristive crossbars.**

***Index Terms*—In-Memory Computing, Memristor-Aided loGIC (MAGIC), Combinational Flow**

I. Introduction

Executing entire applications on RRAM-based memristor-crossbars for *In-Memory Computing* (IMC) is emerging as a defining trend for next-generation computing [1], [2]. The *Memristor-Aided loGIC* (MAGIC) design style is a promising approach, offering a methodology to perform logic operations directly within memory [3], [4]. Current methodologies supporting this style often utilise a mapping tool that takes a *Register-Transfer Level* (RTL) design as input and converts it to a memristor crossbar configuration [2]. However, this translation has largely been limited to benchmark-sized circuits, leaving the mapping of complete high-level algorithms largely unexplored.

This challenge exists primarily since applications are written in *High-Level Languages* (HLLs). Standard HLS maximises clock frequency by inserting flip-flops to shorten critical paths [5], [6]. However, such pipelining is inefficient for memristive MAGIC logic, where state preservation is costly. Proposed A2C inverts this optimisation goal, prioritising *State Minimisation* over frequency. It translates HLLs into massive, unclocked ripple-carry structures [7], [8], ensuring the entire function resolves in a single spatial pass, thereby addressing a synthesis constraint, difficult for standard tools to meet.

In this paper, we first present an *Application-to-Combinational* (A2C) Flow. This novel methodology alters the translation of high-level application code into hardware and then uses an in-memory synthesis and verification tool to evaluate the in-memory resources. The primary challenge in targeting constrained hardware fabrics is the reliance on complex, inference-based behavioural synthesis, which often yields inefficient results. Our primary novelty is the direct transformation of an application's abstract logic into a purely combinational, structural Verilog RTL. This process guarantees the output is a single, dataflow function where the hardware architecture is explicitly defined, not inferred. By eliminating all sequential elements, behavioural operators, and iterative constructs from the generated RTL, our proposed design flow reduces the final synthesis stage to a direct and highly simplified technology mapping, enabling rapid and efficient hardware generation.

The methodology begins by parsing the high-level application code into an *Abstract Syntax Tree* (AST), which captures the program's logic, data dependencies, and control flow. A novel generator then traverses this AST, driven by three key algorithmic transformation processes. First, *State Linearisation* re-architects all stateful variables into a pure dataflow model, explicitly representing the application's entire state as primary inputs and outputs of a single, stateless function. Secondly, Algorithmic *Control-Flow Flattening* directly translates all programmatic branching logic into equivalent combinational, data-selection hardware structures, which effectively converts control-flow dependencies into data-flow routing problems. Finally, *Temporal-to-Spatial unrolling* handles iterative constructs by exploiting hardware parallelism.

In standard processing, loops execute sequentially over time, reusing the same logic resources for each step. Our flow transforms this by physically replicating the logic for every iteration and chaining them together in space across the circuit area. This converts a time-based sequence into a physical, spatially cascaded hardware implementation within the generated RTL. The resultant output of the A2C flow is a single, top-level Verilog module that is guaranteed to be purely combinational and structural. This file consists only of I/O declarations, wire definitions, and a flattened set of dataflow assignments that represent the entire functionality of the application. This synthesis-friendly RTL serves as an

explicit architectural model, devoid of complex behavioural constructs. Consequently, the final synthesis tool is no longer required to perform high-level optimisation. Its task is reduced to a direct, straightforward mapping of this structural code into a target netlist of a few simple gates, enabling a high degree of efficiency in hardware generation from high-level applications.

For the in-memory analysis, we use a comprehensive synthesis and verification methodology for MAGIC-based designs, similar to [2], [9]. This approach uses an initial representation of crossbar micro-operations, transforms it into an intermediate representation, and passes it to a verification tool. This tool translates the design into a *Boolean Satisfiability* (SAT) formula (encoded in a DIMACS file), which is then rigorously analysed by the Z3 *Satisfiability Modulo Theory* (SMT) solver to provide the final verification outcome against the golden response.

The rest of the paper is organised as follows. Section II provides the necessary background, Section III proposes the A2C flow framework and the in-memory design verification flow. Experimental results are provided in Section IV, and the conclusion is in Section V.

II. Background

This section briefly describes the MAGIC-based design along with the micro-operation representation.

A. MAGIC-based Design

MAGIC is a logic representation designed for use with memristors, specifically within crossbar arrays. It is stateful logic, where the output of a computation is stored as the resistance state of the device that performed the operation. While MAGIC is theoretically capable of implementing any logic gate, only NOR and NOT gates can be directly realised on the memristor crossbar. These primitive gates are then used to construct more complex functions.

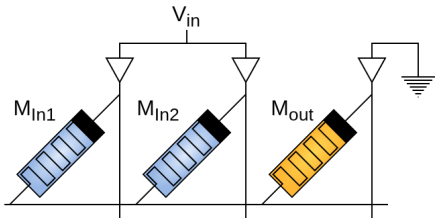


Fig. 1: 2-input Crossbar representation of NOR gate using MAGIC

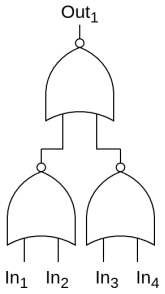
A logic operation, such as the 2-input NOR gate shown in Fig. 1, is executed in two distinct steps:

- In the initialisation step, the output memristor (M_{out}) is set to Logic 1. The input logic values are loaded onto the input memristors (M_{in}). Logic 1 is represented by the *Low-Resistance State* (LRS), and Logic 0 is represented by the *High-Resistance State* (HRS).
- The required programming voltages are applied across the crossbar's columns. This voltage application drives the computation, and the final result of the NOR function is immediately stored as the new resistance state of M_{out}.

B. Micro-Operation File

```
1  name Example
2  input In1,In2,In3,In4
3  output Out1
4
5  Outputs are placed at:
6  out -> 0x5
7
8  #Level 0
9  0 False 0 /In1 1 /In2 2 True
10 1 False 0 /In3 1 /In4 2 True
11 #Level 1
12 0 False 3 0x2 4 1x2 5 True
```

(a) Example Micro-Operation File



(b) Netlist

Fig. 2: Micro-Operation Representation

In this section, we discuss the micro-operation file format, which was introduced in [9]. The crossbar micro-operations specification is a data file containing instructions that define how logic operations are executed on the memristive crossbar. Specifically, these instructions detail the mapping of input and output variables and specify the sequence of MAGIC gate operations to be performed on the crossbar array. Fig. 1 shows an example NOR netlist for which the micro-operation file is provided in Listing Fig. 2a. This has four inputs, one output, and has two levels. The first few lines of the file contain information about input, output and the exact location of the output after execution. The number of columns required in the crossbar is proportional to the number of logic levels in the circuit, as gates in successive levels are mapped to successive sets of columns. Level 0, which is the first level, contains two NOR gates, which are mapped in row 0 and row 1. Level 1, which is the second level, contains one NOR gate, which is mapped in row 0 (Ref. Fig. 2b). After the execution, the output is stored in the 0x5 crossbar location. For more details, it is recommended to refer [9].

III. Proposed Methodology

A. A2C Design Framework

Fig. 3 details the A2C framework's conversion mechanism, which translates temporal high-level software into the spatial, instruction-less format as required by the MAGIC crossbar. This process resolves the domain incompatibility via three deterministic transformation engines.

Temporal-to-Spatial Unrolling (The Loop Engine): The translation of iterative loops highlights the primary divergence from standard synthesis. Conventional HLS typically prioritises *Resource Sharing* by reusing a single datapath over multiple clock cycles to conserve area. Even when standard tools unroll loops, they often do so partially to balance throughput with size. In contrast, A2C enforces strict *Resource Replication*. As illustrated in Fig. 3, the flow treats iterations not as time steps but as spatial instances: a loop of N iterations is fully flattened into N physically distinct, cascaded hardware blocks. This 'unroll-all' strategy converts the temporal complexity of the software entirely into the structural complexity of the netlist, eliminating the need for

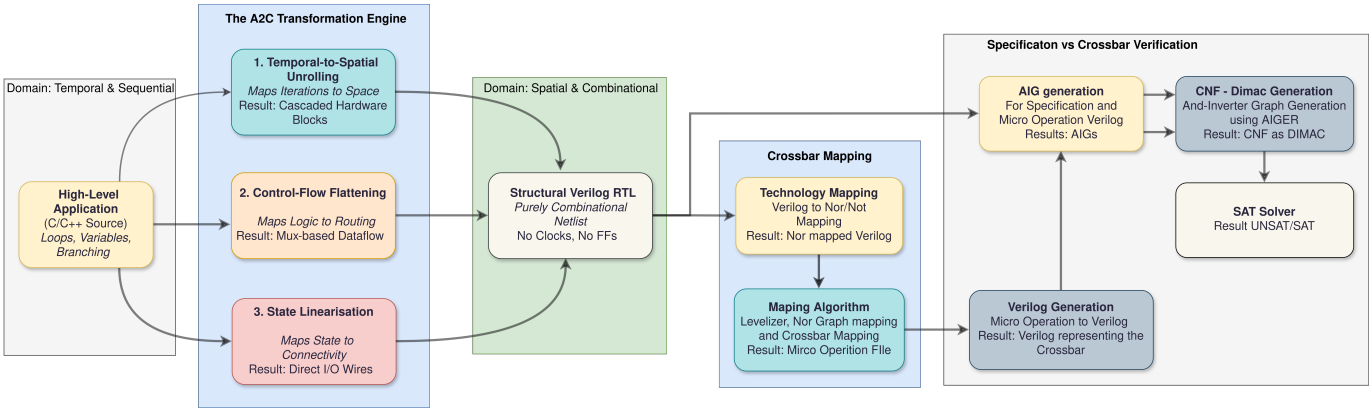


Fig. 3: The A2C Transformational Pipeline and the Verification Process

state machines. This direct trade-off of area for statelessness explains the linear scaling trend observed in Table I.

In contrast, A2C enforces strict resource replication. As illustrated in Fig. 3, the flow treats iterations not as time steps but as spatial instances: a loop of N iterations is fully flattened into N physically distinct, cascaded hardware blocks. Because hardware must be physically instantiated at compile time, the A2C flow currently requires all loops to have a statically determinable maximum bound. For data-dependent loops where the exact number of iterations is unknown beforehand, the loop is unrolled to its maximum possible limit. Our Control-Flow Flattening engine then handles early exit conditions by routing the finalised data through the remaining hardware blocks without further modification. This *unroll-all* strategy converts the temporal complexity of the software entirely into the structural complexity of the netlist, eliminating the need for state machines.

Control-Flow Flattening (The Logic Engine): Software relies on *branching* (i.e. jumps), which requires an instruction pointer, i.e. a sequential element incompatible with MAGIC. The second engine transforms all control dependencies (e.g., if-else) into data routing problems. Instead of conditionally executing code, the generated hardware computes all possible paths and uses multiplexers to select the correct result based on the condition. This ensures the output remains a continuous, unbroken stream of combinational logic.

State Linearisation (The Variable Engine): This final transformation engine, shown in Fig. 3, handles state variables. Standard synthesis infers registers (flip-flops) to store intermediate values. It creates a *Stateless* representation. Because the loops are fully unrolled, every variable assignment becomes a unique wire in the netlist. The *state* is not stored, but is propagated. This linearisation guarantees the result shown in the *Sequential Elements* column of Table I: zero. It ensures the output is not just *low latency*, but completely combinational, which satisfies the strict stateless requirement of the MAGIC synthesis tool.

B. In-Memory Mapping and Verification Framework

In this section, we discuss our in-memory framework that relies on a robust synthesis and verification methodology for MAGIC-based circuits, employing techniques similar to those described in [2], [9]. This framework is divided into two phases. In the first phase, the micro-operation is generated, and in the next phase, verification is performed. The Verilog file generated from the A2C design is first transformed into a NOR netlist using the ABC tool. This netlist is then fed to a mapping tool.

Mapping and Micro-Operation Generation: In this step, the generated NOR netlist is topologically sorted using the *As Late As Possible* (ALAP) scheduling algorithm to assign a distinct level number to every gate. This level assignment is critical because it ensures that data dependencies are rigorously maintained, meaning that a gate is never scheduled until all the inputs it requires, which originate from gates in lower-numbered levels, are fully available. The level numbers assigned by this process then directly dictate the execution strategy on the crossbar. All gates belonging to the same level can be evaluated in parallel, exhibiting the inherent parallel processing capabilities of the memristive array structure. Execution proceeds sequentially across levels, strictly moving from Level 0 to Level 1, and so on, to maintain correct data flow. Once the scheduling and mapping are complete, the entire sequence of parallel gate evaluations, along with their exact placement and execution details, is translated into a set of detailed instructions known as micro-operations. These micro-operations provide step-by-step instructions detailing exactly where gates are mapped within the crossbar. They specify the precise row and column coordinates, as well as the destination where the output is stored.

Verification of the Generated Micro-Operation: As the micro-operation file is generated by a program, we also use a verification methodology to ensure that the generation process is correct. The verification process is initiated by a parser that first reads the micro-operations and translates them into a structural Verilog specification (.v file). This newly generated .v file is then combined with the original input specification (also in Verilog) and submitted to the ABC tool [10] to construct the corresponding *And-Inverter Graphs* (AIGs).

Subsequently, the AIGER tool [11] is used to convert these AIGs into a combined miter circuit. The final step involves converting the miter circuit into (Conjunctive Normal Form) clauses (DIMACS format) and feeding this representation into

the Z3 solver [12]. This solver performs satisfiability checking, reporting UNSAT if the files are logically equivalent and SAT if they are not. Please note that here we evaluate the functional correctness of the micro-operation file, which contains the final crossbar mapping information.

IV. Experimental Results

In this section, we provide the results of our proposed A2C flow transformation and in-memory mapping and verification. For realistic applications, three circuits have been chosen for the experiments [13].

The selected benchmarks represent critical atomic primitives found in data-centric workloads that require massive data parallelism. For instance, $clamp$ targets database query filtering, $max5$ implements the pooling logic in neural networks, and $sum4$ handles pixel accumulation for image processing. When mapped onto crossbars, these primitives enable massive parallelism, allowing the same operation to be performed across gigabytes of memory simultaneously.

- ***clamp*** (Database primitive) limits an input signal to a specific range defined by lower and upper bounds. It outputs the input value if within the range, or saturates at the nearest bound if the limit is exceeded.
- ***sum4*** (Accumulation primitive) computes the sum of four equal bit-width numbers. The output bit-width is sized to accommodate the maximum possible sum without overflow.
- ***max5*** (Pooling primitive) determines the maximum value from a set of five inputs. It uses progressive pairwise comparisons to isolate and output the single largest value.

A. A2C Flow Result

TABLE I: Structural Characterisation of A2C-Generated RTL Metrics indicating the complexity and scale of the RTL generated by the A2C flow before MAGIC mapping.

Benchmark(BM)	Bit-Width (BW)	A2C Translation Time (s)	RTL Lines of Code (LoC)	Combinational Gate Count (†)	Sequential Elements
$clamp$	8	0.12	350	85	0
	16	0.21	700	170	0
	32	0.43	1400	340	0
	64	0.91	2800	680	0
$sum4$	8	0.15	410	120	0
	16	0.25	820	240	0
	32	0.52	1640	480	0
	64	1.10	3280	960	0
$max5$	8	0.22	520	160	0
	16	0.45	1040	320	0
	32	0.98	2080	640	0
	64	2.05	4160	1280	0

The notable result in Table I is the consistent value of zero in the *Sequential Elements* column across all benchmarks and bit-widths. In standard *High-Level Synthesis* (HLS), increasing the complexity of an operation (e.g., from 8-bit to 64-bit) often requires additional pipeline stages or state registers to meet timing closure. In contrast, our results demonstrate that the State Linearisation engine successfully flattened all internal variables into purely combinational connectivity. This makes the generated RTL strictly dataflow-driven and fully compatible with the stateless execution model of the MAGIC crossbar logic.

Besides, the *Equivalent Gate Count* and LoC metrics exhibit approximately linear correlation with the input Bit-Width (BW). For instance, doubling the BW of the $sum4$ benchmark from 32 to 64 results in an approximate doubling of the gate count (480 to 960 gates). This trend exhibits the structural determinism of the A2C flow. The Temporal-to-Spatial Unrolling mechanism maps algorithmic complexity directly to hardware resources in a predictable $O(N)$ manner.

B. In-Memory Mapping and Verification Result

Table II shows how the crossbar size and latency behave for different input bit-widths for the $clamp$, $sum4$ and $max5$ circuits. In the first column `BW` defines the bit-width for each of the benchmarks, `BM` specifies the name of the benchmark, PI and PO the primary input/output size, `CBS` the crossbar size and `CCBS` the corrected crossbar size. The last four columns present latency metrics: L_R denotes the read, L_W the write. L_E the evaluation, and L_T the total latency. It is important to note that because the generated RTL is entirely unclocked, these latency metrics do not represent standard CPU clock cycles. Instead, they represent MAGIC execution steps. For instance, evaluation latency (L_E) corresponds to the number of topologically sorted logic levels in the design; since all gates within the same level evaluate simultaneously during a single voltage application cycle, each level adds one discrete step to the overall execution time. The table shows that the latency grows linearly for all benchmarks with increasing bit-width.

TABLE II: Mapping Results for Benchmarks

BM	BW	PI	PO	CBS	CCBS	L_R	L_W	L_E	L_T
$clamp$	8	24	8	16x 79	128x128	213	79	28	320
	16	48	16	32x 91	128x128	443	91	32	566
	32	96	32	64x103	128x128	905	103	36	1044
	64	192	32	128x115	128x128	1831	115	40	1986
$sum4$	8	32	10	16x 82	128x128	281	82	28	391
	16	64	18	31x 94	128x128	592	94	32	718
	32	128	34	63x106	128x128	1219	106	36	1361
	64	256	66	127x118	128x128	2472	118	40	2630
$max5$	8	40	8	32x123	128x128	436	123	43	602
	16	80	16	64x141	256x256	892	141	49	1082
	32	160	32	128x159	256x256	1804	159	55	2018
	64	320	64	256x177	256x256	3628	177	61	3866

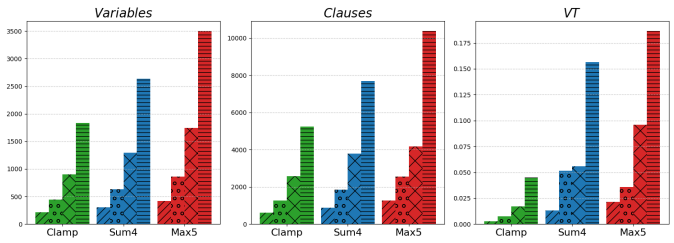


Fig. 4: Verification Results for the considered Benchmarks for 8,16,32 and 64 bit

Fig. 4 presents the verification results for the considered benchmarks. As mentioned in Section III-B, to ensure the functional correctness of our mapping method, the micro-operation file is verified using a Boolean satisfiability (SAT)

approach. The original design file and the generated micro-operation file undergo a series of steps to construct a combined CNF in DIMACS format, which is subsequently fed into the Z3 solver. An UNSAT result from the solver conclusively demonstrates that the generated micro-operations are correct.

For the considered benchmark, doubling the number of inputs roughly doubles the number of variables and clauses. Importantly, the verification time remains consistently under one second for all benchmarks.

V. Conclusion

The A2C Flow establishes a paradigm of 'C-to-Memory' synthesis for Data-Centric Computing, that resolves the fundamental incompatibility between MAGIC-based architectures and sequential RTL. By transforming temporal software constructs into spatial hardware cascades, utilising the proposed State Linearisation, Control-Flow Flattening, and Temporal-to-Spatial Unrolling, we achieve the purely combinational, stateless representation required for these arrays. Furthermore, our integrated verification framework utilises Boolean Satisfiability (SAT) to rigorously validate functional equivalence between the generated micro-operations and the high-level reference. We firmly believe that this methodology provides the essential infrastructure to automatically compile complex kernels, moving the field past manual component-level design toward next-generation in-memory logic.

Acknowledgment

This work is partly supported by the UK Engineering and Physical Sciences Research Council through grants, EP/Z533749/1, the German Research Foundation (DFG) within the Project PLiM (DR 287/35-1, DR 287/35-2 and by the Federal Ministry of Research, Technology and Space (BMFTR) within the ExaVerse project under grant no. 01IW25003. For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any arising Author Accepted Manuscript version.

References

[1] R. Ben-Hur et al., "SIMPLER MAGIC: Synthesis and Mapping of In-Memory Logic Executed in a Single Row to Improve Throughput," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2434–2447, 2020.

[2] S. Nabipour et al., "Multi-Input MAGIC Synthesis and Verification for In-Memory Computing Design," in *2025 IEEE 55th International Symposium on Multiple-Valued Logic (ISMVL)*. IEEE Computer Society, 2025, pp. 178–183.

[3] S. Kvatinsky et al., "MAGIC—Memristor-aided logic," *IEEE Trans. on Cirt.and Sys. II: Exp. Brief.*, vol. 61, no. 11, pp. 895–899, 2014.

[4] R. Gharpinde, P. L. Thangkhiew, K. Datta, and I. Sengupta, "A Scalable In-Memory Logic Synthesis Approach Using Memristor Crossbar," *IEEE Trans. on VLSI Syst.*, vol. 26, no. 2, pp. 355–366, Feb 2018.

[5] N. Pundir, S. Aftabjahani, R. Cammarota, M. Tehranipoor, and F. Farahmandi, "Analyzing security vulnerabilities induced by high-level synthesis," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 18, no. 3, pp. 1–22, 2022.

[6] S. A. Alam, D. Gregg, G. Gambardella, T. Preusser, and M. Blott, "On the RTL Implementation of FINN Matrix Vector Unit," *ACM Trans. on Embedded Computing Systems*, vol. 22, no. 6, pp. 1–27, 2023.

[7] N. Pundir, F. Farahmandi, and M. Tehranipoor, "Secure high-level synthesis: Challenges and solutions," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2021, pp. 164–171.

[8] S. S. Hoseininasab, "Using HLS to raise the design abstraction level for faster exploration of different CPU Micro-architectures," Ph.D. dissertation, Université de Rennes, 2025.

[9] L. Weingarten et al., "A Complete Synthesis and Verification Approach for MAGIC Based In-Memory Computing," in *DTTIS*, 2025, pp. 1–6.

[10] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 24–40.

[11] A. Biere, K. Heljanko, and S. Wieringa, "AIGER is a format, library and set of utilities for And-Inverter Graphs," available at https://github.com/arminbiere/aiger, 2024.

[12] L. d. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[13] S. Ahmadi-Pour et al., "Messi: Task mapping and scheduling strategy for fpga-based heterogeneous real-time systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 30, no. 3, pp. 1–29, 2025.