SIMPLER MAGIC: Synthesis and Mapping of In-Memory Logic Executed in a Single Row to Improve Throughput

Rotem Ben-Hur, Ronny Ronen, *Fellow, IEEE*, Ameer Haj-Ali, *Student Member, IEEE*, Debjyoti Bhattacharjee, Adi Eliahu, Natan Peled, and Shahar Kvatinsky, *Senior Member, IEEE*

***Abstract*— In-memory processing can dramatically improve the latency and energy consumption of computing systems by minimizing the data transfer between the memory and the processor. Efficient execution of processing operations within the memory is therefore a highly motivated objective in modern computer architecture. This paper presents a novel automatic framework for efficient implementation of arbitrary combinational logic functions within a memristive memory. Using tools from logic design, graph theory and compiler register allocation technology, we developed SIMPLER (*Synthesis and In-memory MaPping of Logic Execution in a single Row*), a tool that optimizes the execution of in-memory logic operations in terms of throughput and area. Given a logical function, SIMPLER automatically generates a sequence of atomic Memristor-Aided loGIC (MAGIC) NOR operations and efficiently locates them within a single size-limited memory row, reusing cells to save area when needed. This approach fully exploits the parallelism offered by the MAGIC NOR gates. It allows multiple instances of the logic function to be performed concurrently, each compressed into a single row of the memory. This virtue makes SIMPLER an attractive candidate for designing in-memory Single Instruction, Multiple Data (SIMD) operations. Compared to previous work (that optimizes latency rather than throughput for a single function), SIMPLER achieves an average throughput improvement of $435\times$. When previous tools are parallelized similarly to SIMPLER, SIMPLER achieves higher throughput of at least $5\times$, with $23\times$ improvement in area and $20\times$ improvement in area efficiency. These improvements more than fully compensate for the increase (up to 17% on average) in latency.**

***Index Terms*—Memristor, memristive systems, logic design, MAGIC, mMPU, von Neumann architecture, logic synthesis, throughput.**

I. Introduction

A basic assumption that has guided computer architects in the design of almost all modern computing systems is the separation between processing units and data storage units. In almost any computing system today, data is processed by the processor and stored inside the memory. Over the last few decades, computer architects have enjoyed orders of magnitude improvement in computer performance, *e.g.*, processor speedup, reduced power consumption, and the downscale of system dimensions. This trend line was fueled by impressive technological achievements in the two principal computer components, the processor and the memory. Nowadays, however, it seems that both units have reached a scaling barrier, and that data processing performance is now limited mostly by the inevitable need to transfer data. The energy and delay associated with this data transfer are estimated to be several orders of magnitude higher than the cost of the computation itself [1]. This data transfer bottleneck is known as the $memory\ wall$.

Numerous methods for alleviating the memory wall have been explored. The most common method is to integrate several levels of cache memory near the processor. Cache memories can significantly reduce the amount of data transferred between the processor and the memory [2], but do not fully eliminate this need. A relatively more recent (and less prevalent) approach is to integrate processing units within memory elements. The idea of combining processing units within DRAM and SRAM cells was explored in [3]–[5]. However, the potential benefits of in-memory computing were not fully exploited in these works, as they still required data transfer between storage and processing elements. In most common technologies, conventional memory cells are in fact ill-suited for performing direct computations.

The breakthrough in the field of in-memory computing came with the emergence of new memory technologies that can be used to perform logic operations, in addition to their traditional data storage capabilities. Some of these technologies are based on novel electrical elements called memristors [6]. Memristors are used to modulate data into resistance, where high and low resistances represent logical '0' and '1', respectively. Memristors are actually passive elements with very promising capabilities. They can change their resistance as a result of the voltage applied across them, and their high density, non-volatility, low power consumption and CMOS fabrication compatibility [7], [8] offer huge potential improvements over current cell technologies. Furthermore, memristors can be used to perform logic operations, enabling processing within the memory [9]–[12]. Resistive random access memories (RRAM) [13] have paved the way for combining processing and memory, since they allow the same physical entities to be used for both [9]–[12].

An attractive approach for performing logic within a conventional memristive memory array is $stateful\ logic$, where logical states of logic gates are represented by resistance and the logic gates are constructed solely by memristors. The inputs of a logic gate are the logical states of the memristors before the execution of the logical operation associated with

R. Ben-Hur, R. Ronen, A. Eliahu, Natan Peled, and S. Kvatinsky are with the Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion Israel Institute of Technology, Haifa 32000, Israel. A. Haj-Ali is with the Faculty of Electrical Engineering and Computer Science, University of California, Berkeley, California, USA. D. Bhattacharjee is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore (E-mail: rotembenhur@campus.technion.ac.il, shahar@ee.technion.ac.il).

This research is partially supported by the ERC under the European Union's Horizon 2020 Research and Innovation Programme (grant agreement no. 757259), and by the Israel Science Foundation grant no. 1514/17.

the gate. Likewise, the output of the gate is the state of the memristor after the execution of the logical operation associated with the gate.

Several stateful logic families compatible with memristive memories have been proposed in this context [14], [15]. A leading candidate among these is Memristor-Aided loGIC (MAGIC) [16]. MAGIC has been shown to outperform competing methods [17], while enabling a full implementation within a standard memristive memory array. The basic MAGIC gate executes a logical NOR operation. Since NOR is a complete logic function, a MAGIC NOR gate is sufficient for the execution of any logical operation within the memory: the desired logic function is divided into a sequence of MAGIC NOR operations. In what follows, we refer to a single execution of a function with specific inputs as a *computation instance* (or as an *instance*). MAGIC NOR gates can be applied anywhere within the memory array, using the same cells and structure, where the stored data functions as input for the logical operation.

The integration of data storage capabilities with MAGIC-based processing within memristive memories has led to the development of the $memristive\ Memory\ Processing\ Unit$ (mMPU) [18]. This novel architecture replaces conventional DRAM memory with a memory that is also capable of performing general-purpose computing. The mMPU consists of standard memristive memory arrays, with only minor modifications to the CMOS periphery and control circuits to allow support for computations as well as conventional data read and write operations. The mMPU is therefore completely compatible with standard von Neumann architectures, as it can operate either as a hybrid memory-processing unit or as a standard memory. Nevertheless, the advantages of a memristive crossbar array, *e.g.*, density and non-volatility, are maintained in the mMPU architecture.

To perform a computation within the mMPU, a compute command is received by the mMPU controller [19]. The controller interprets the command and converts it into a sequence of MAGIC operations. It then sends the corresponding control signals to the memristive memory arrays to perform the actual logic operations. To realize the full potential of the mMPU, the desired computation must be converted into an efficient sequence of MAGIC operations. Such a sequence should involve a small number of computational steps, utilize only a limited area within the array, and consume low energy. Several such sequences were proposed for some popular arithmetic operations and shown to be relatively efficient. The studied functions include fixed-point addition and multiplication [17], [20], [21], and convolution [22]. However, all of these works relied on manual crafting and optimization of the sequence of operations, designed for a specific logical function. Obviously, this is neither a general, nor optimal, design methodology. Furthermore, manual designs are naturally time consuming and error-prone, hence prolonging the time-to-market of any future product.

Recent work has focused on automatic conversion of arbitrary logical functions to a sequence of executions within the memory. In [23], [24], tools were developed to generate execution sequences for arbitrary logical functions, while minimizing the computational latency in a memristive memory setup. Latency is minimized for a single computation instance by exploiting parallelism features of in-memory stateful logic operations so to **execute several NOR operations within an instance in a single cycle**. However, because multiple rows and columns must be utilized, this optimized latency comes at the cost of disabling significant areas within the array, since many cells are unused for the computation. Additionally, numerous instances of the same logical function within a given memory array can only be executed serially in this method.

In this paper, we take a different approach: we improve the performance of the mMPU by maximizing the *throughput* rather than minimizing the *latency*. We work under the *Single Instruction, Multiple Data* (SIMD) [2] concept, **exploiting parallelism among different computation instances rather than optimizing a single instance of a given logical function**. In this model, the mMPU will perform a series of identical computations (differing only in the input data) on many computation instances. We present a novel synthesis and mapping tool called SIMPLER[1] (Synthesis and In-memory MaPping of Logic Execution in a single Row). SIMPLER unleashes the full potential of parallel computations offered by in-memory executions within a memristive array, in the SIMD setup.

As a synthesis tool, SIMPLER outperforms previous work by performing *in parallel* multiple instances of computations associated with a given logical function. Thus, although the latency of a single computation instance may be slightly higher, the overall throughput of the array increases dramatically thanks to the ability to compute each instance in a different row in parallel. Such a configuration allows SIMD operations to be supported efficiently in an mMPU setup for the first time. The magnitude of this paradigm change is illustrated by a simple, realistic, example: a system consisting of a 512-row memory array can execute 512 computation instances in parallel. Even at $4\times$ longer latency, the system provides an astonishing $128\times$ higher throughput.

SIMPLER represents a fundamental shift in the design of in-memory computing systems. New challenges that arise in this novel, throughput-oriented, approach are all solved using SIMPLER. In this context, SIMPLER makes the following contributions:

1) **Automates** the process of generating in-memory MAGIC NOR execution sequences to **improve the throughput** of the computation and allow **efficient SIMD** executions.
2) **Compresses** relatively complex computations into a single finite size memory row by efficient reuse of cells when necessary.
3) **Reduced tool complexity.** SIMPLER uses efficient compiler like register allocation technology to generate an optimized mapping for huge computations within seconds.

Compared to previous work that optimizes latency rather than throughput for a single function ([24], [25]), SIMPLER achieves an average throughput improvement of over $435\times$ for a memory array with 512 rows. When the previous tools are parallelized in a similar manner to SIMPLER, SIMPLER

[1]The SIMPLER tool may be found at:
https://github.com/RotemBenHur/SIMPLER-MAGIC.git

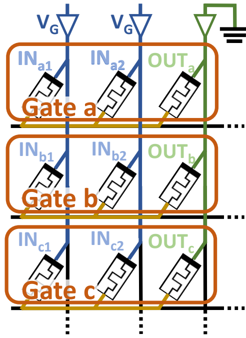


Fig. 1. Parallel execution of three aligned MAGIC NOR gates.

offers at least 5× higher throughput and 23× smaller area usage.

II. Preliminaries and Motivation

All in-memory computations in this work rely on the basic MAGIC NOR operation. We first explain MAGIC and how it motivates our choice to optimize throughput in Section II-A. Then we describe a motivational example to explain and support this choice in Section II-B. Next, we survey the relevant related work in Section II-C. Finally, we formally define the problem this paper solves in Section II-D.

A. Preliminaries and Definitions

The MAGIC NOR gate is performed by applying voltage(s) to the input(s) and output memristors. The state of the output memory cell changes in accordance with the logical states of the memristors. The advantages of MAGIC over other stateful logic techniques include the separation between the input(s) and output memristors, the need for only a single execution voltage (called V_g), and the lack of additional periphery elements [17]. The N-input NOR gate operation requires two steps (clock cycles):
(1) A logical '1' is written to the output memristor by applying a voltage, denoted V_{w1}, across it.
(2) V_g is applied to all N inputs, and the ground is connected to the output.

A single-input NOR is a NOT gate; hence, both N-input NOR and NOT gates may be executed by MAGIC. Figure 1 illustrates the in-memory execution of three MAGIC NOR gates, each using three cells: two for the inputs and one for the output. The inputs and output of a single gate need not be located in adjacent cells; the only requirement is that they be located on the same row ($MAGIC\ row$ operation), or column ($MAGIC\ column$ operation). To perform both MAGIC row and MAGIC column operations, a transpose memory [17] is required.

Since a NOR operation spans the complete set of Boolean operations, a MAGIC NOR gate is sufficient to execute any desired logic function. Hence, MAGIC NOR may be used as the basic computing element for all kinds of processing within the memory by dividing the desired function into a sequence of MAGIC NOR operations ($execution\ sequence$). The execution of a sequence of two two-input MAGIC NOR gates is shown in Figure 2. These basic NOR operations are performed serially using the memory cells, where the output of the first MAGIC gate acts as one of the inputs

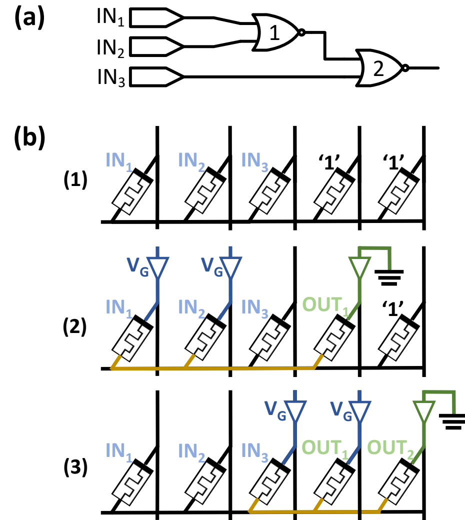


Fig. 2. Serial execution of two MAGIC NOR gates in a single row.
(a) The two-gate netlist.
(b) Execution of the netlist in three steps (clock cycles):
(1) Writing a logical '1' to the output memristors (initialization).
(2) Gate 1 execution - NOR(IN_1,IN_2).
(3) Gate 2 execution - NOR(IN_3,OUT_1).

of the second gate during the second stage (cycle) of the computation. Serially executing all the gates in the sequence may be time consuming. Aligning the inputs and outputs of different MAGIC NOR gates allows them to be executed in parallel, as illustrated in Figure 1. Wisely exploiting this property may either improve the latency of a logic function or the throughput of a SIMD (*Single Instruction, Multiple Data*) operation.

The latency may be improved by parallelizing several gates of the NOR execution sequence belonging to the same single instance, using multiple rows for the execution of a single instance [23]. The throughput may be improved by parallelizing the execution of many instances of the same logic function: each instance is placed in a different row.

In our context, throughput is defined as:

$$Throughput = \frac{\#instances}{Latency}, \tag{1}$$

where $\#instances$ is the number of instances of the function performed in parallel (=#rows), and $Latency$ is the number of clock cycles required for the computation. The throughput increases linearly with the number of instances.

When the row is wide enough to hold all the inputs and gates required to execute the desired function (number of columns $\geq$ number of inputs + number of gates), the computation fits easily into that row, and the number of execution steps (clock cycles) is equal to the number of gates. However, if the row is not wide enough (number of columns $<$ number of inputs + number of gates), the computation must be split into different rows or cells must be reused.

A cell can be reused when it stores data no longer needed for the rest of the computation, $i.e.$, all its consumers have already been computed. A cell must be initialized before reuse. Because MAGIC requires that a logical '1' be initially written to the output memristor, a cell is re-initialized by writing a '1' to it. We assume that all desired cells within a row can be re-initialized in a single clock cycle, by applying

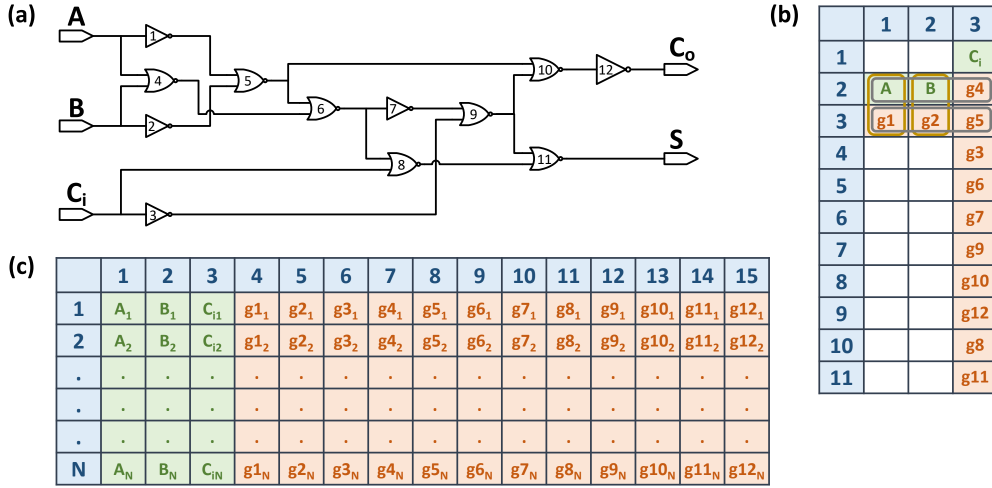


Fig. 3. (a) Single-bit full adder (1-bit FA) netlist. (b) A single 1-bit FA execution using an 11×3 array. Each green cell initially stores the inputs. Each orange cell gi stores the result of gate i. Gates $g1$ and $g2$ (yellow rectangles) are executed in parallel during the first clock cycle, and gates $g4$ and $g5$ (gray rectangles) are executed in parallel during the second clock cycle. All other gates are executed serially. The total execution time is therefore 10 clock cycles, but parallelizing instances is difficult. (c) N 1-bit FA operations executed in parallel. Each FA operation is allocated to a single row. Each green cell initially stores the inputs. Since they are all aligned, all N FA operations are executed concurrently. Each orange cell gi_j stores the result of gate i of the jth FA operation. All results of gi_j of a specific i for all j (each orange column) are executed in a single clock cycle. Since there are enough columns for computation, no initialization cycles are necessary, and the execution time is 12 cycles, in accordance with the number of gates.

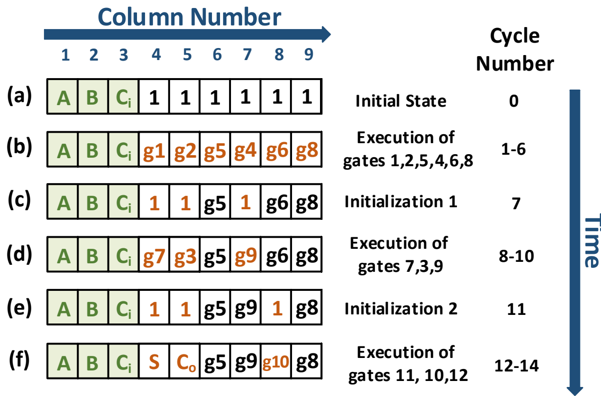


Fig. 4. Execution of the *1-bit full adder* netlist using a single row of the memory with 9 columns, following the execution order detailed in Figure 6. The states of the row during different stages of the computation are: (a) The initial state of the row. (b) Serial calculation of the first 6 gates. (c) Initialization of the 3 gate results not required as inputs for future computations. (d) Serial calculation of the next 3 gates. (e) Initialization of 3 gate results not required as inputs for future computations. (f) Serial calculation of the last 3 gates.

V_{w1} to their columns and connecting the row to ground. Re-initializations that take more than a single-clock cycle is addressed in Section IV. The overall number of total cell initializations does not change because of cell reuse. Every MAGIC write has to be preceded by a cell initialization, so the total number of writes remains the same regardless of cell reuse; only their timing is different. Depending on the desired computation and the number of cells available for it, cell reuse may or may not suffice. The execution order of the gates of a given computation determines the number of cells that can be reused at a given time; thus, a mapping will be found only if the execution order of the gates is efficient. For large logic functions, all of the cells might still be required for re-initialization, because there are no unneeded cells to free. Consequently, the tool fails to find a possible mapping.

Since the row has to be large enough for complex executions, the size of the memory array limits the complexity of the computations that can be executed in-memory. In conventional DRAM memories, the smallest unit is called a MAT. Usually, a DRAM MAT consists of an array of 512×512 memory cells. In memristive memories, a MAT may contain up to 512×512 cells in transistor-less memristive arrays and up to 2048×8192 in 1T1R arrays [8]. Using a single MAT to execute a given function yields better latency and energy consumption. In contrast, distributing a single computation instance over more than a single MAT requires data transfer between the MATs, significantly reducing the benefits of in-memory computation [26]. However, the same computation can be parallelized over different data in different MATs. In this paper, we consider a MAT size of 512×512. This size allows us to maintain the conventional DRAM structure while supporting in-memory execution of relatively large functions (large number of MAT columns) and improving the parallelism (large number of MAT rows). To show the potential of executing larger functions, we report some results with a 1024×1024 MAT size.

B. Motivational Example

To further clarify the definitions and explanations from the previous subsection, we describe an example of a single-bit full adder, consisting of 12 gates[2], as described in Figure 3a. To improve the latency of a single instance of the full adder, it is executed using an 11×3 memory array. Two pairs of NOR operations are parallelized, and the execution takes 10 clock cycles, as shown in Figure 3b.

The throughput of the full adder can be improved by executing N instances in parallel using an $N \times 15$ array. Each instance is executed using a single row, and the number of execution steps (clock cycles) is 12, in accordance with the number of NOR and NOT gates (independent of the value of N), as demonstrated in Figure 3c. In this case,

[2]The best known full adder consists of nine two-input NOR Gates. We use a 12-gate full adder to better explain various aspects of the SIMPLER tool.

$row\ size = 15\,columns = 12\,gates + 3\,inputs$; hence, no re-initialization cycles are required. A row with fewer than 15 columns is too small to execute all the gates. Cells must be re-initialized in this case. If, for example, the instance is executed using a 9-cell row, two re-initialization cycles will suffice when the execution order is chosen wisely, as described in Figure 4. On the other hand, when the row size is less than 8, re-initialization will not help, and no mapping exists, regardless of the execution order.

C. Related Work

Techniques for mapping an execution of complex functions into a limited-size MAT were previously proposed. The two leading approaches are (1) to automatically improve the latency of a single computation instance by executing different gates of the same instance in parallel, and (2) to manually improve the throughput by executing different instances of the same logic function in parallel.

In our previous work, we developed SIMPLE MAGIC (Synthesis and In-memory MaPping of Logic Execution for Memristor-Aided loGIC) [23], a tool for improving the latency of a single instance by using both MAGIC row and column operations to exploit the parallelism within that instance. Without reusing cells, SIMPLE maps the gates of the same instance of a function to the memory array so that as many gates as possible of the same computing instance are aligned in either rows or columns; therefore, many gates can be performed in parallel. To achieve the best mapping, SIMPLE solves an optimization problem that minimizes the latency, area or/and energy. SIMPLE improves the latency by 48% on average as compared to a single row execution (without reusing cells).

Motivated by the computational burden of solving optimization problems, Yadav *et al.* [24] suggested heuristics for finding a mapping. Instead of generating the mapping with the maximum number of aligned gates (thus, with maximum parallelism and minimum latency), the unaligned gate outputs required as inputs for other gates were moved by adding copy cycles. Their method resulted in a latency increase of 4.9% and an average area increase of $4.7\times$ as compared to serial execution using a single row.

To alleviate the copy cycles overhead, the authors of SAID [25] proposed an improved heuristics that uses a Look-Up Table based synthesis, based on a Sum-of-Products (SoP) representation, to increase the number of gates executed in parallel. SAID reduces the latency by 28% compared to serial execution using a single row, but increases the area by $8.6\times$.

In [17], the authors proposed that each logic function be mapped into a single row. Parallelism within a single computation instance is not allowed, meaning that the gates are executed serially. Therefore, the latency of the execution (in terms of the number of clock cycles) equals the number of NOR and NOT gates in the execution sequence of that logic function, which is not minimized. Additionally, cell reuse is not supported. Unfeasibly wide rows are thus required to execute large logic functions. In [22], manually crafted algorithms that reuse cells are proposed to solve this problem.

Cell reuse allows larger functions to be feasibly executed inside the size-limited memory arrays. However, manual implementation of complex algorithms (with and without cell reuse) is very tedious and error-prone. The variety of functions they can fit in a restricted memory row size is therefore quite limited. In contrast, SIMPLER automatically generates the execution sequence that allows maximum cell reuse when necessary. For example, SIMPLER reduces the minimum required number of cells for the execution of an 8-bit multiplier from 77 (in [22]) to 65, and reduces the execution latency using a row size of 77 cells from 918 cycles to 699.

Efficient in-memory execution of SIMD operations yields significantly higher throughput than low latency execution of a single computation instance. This motivated our choice to automatically map the function execution into a single row.

D. Problem Definition and Complexity

The problem we address in this paper is similar to the problem discussed in [27]. The problem is generating an execution sequence ES that uses the minimum number of memory cells (smallest row). It is formally defined as follows:

$Problem\ Statement\ 1$: Given a data dependence graph G that represents a logical function, derive an execution sequence ES for G that is optimal in the sense that the number of memory cells required for executing the graph is minimal.

For a general DAG, the problem is known to be NP-Complete [28]. We therefore propose heuristics to find an optimized mapping in a linear complexity.

III. SIMPLER Mapping Algorithm

The SIMPLER flow is divided into two main stages, as shown in Figure 5:

1) ABC Synthesis Tool: Similarly to [23], the first stage is the ABC synthesis tool [29] with a modified cell library. ABC receives an arbitrary logic function in a .pla or .blif format, and produces a NOR and NOT netlist in a verilog format, while minimizing the number of gates.

2) SIMPLER Mapping Tool: The second stage is the SIMPLER mapping tool, an in-house Python script that does the following:

(i) Receives the minimized NOR and NOT netlist and the number of cells within the memory row dedicated to the computation.

(ii) Maps all netlist gates to that single row by determining the locations (column/cell number) of each input and output of the gate and the timing (clock cycle number) in which it is executed.

(iii) Adds re-initialization cycles when cells must be reused.

In the next subsections, the SIMPLER mapping algorithm is described. Section III-A discusses the assumptions and considerations for optimizing the in-memory execution. In Section III-B, the mapping of the netlist gates into the memory row is described. The complexity of the algorithm is then discussed in Section III-C.

A. Principles for Efficient Mapping into the Memory

The mapping of a computation to the memory cells is produced by first determining the execution order of the gates

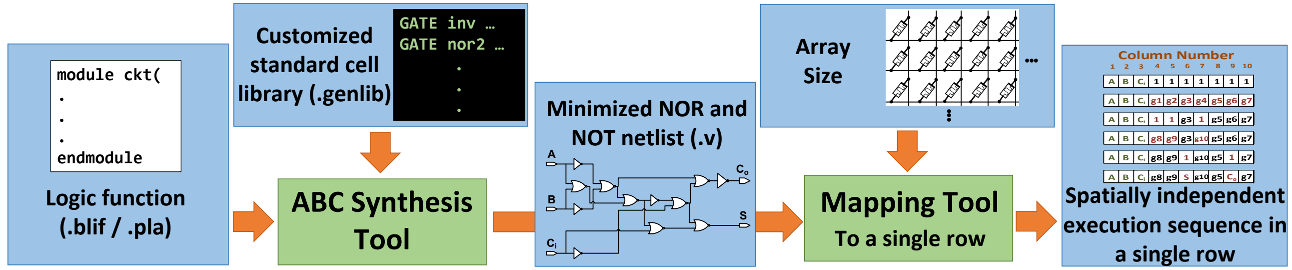


Fig. 5. SIMPLER general flow. The ABC synthesis tool receives an arbitrary logic function and a modified library, and produces a minimized NOR and NOT netlist. An in-house Python-based mapping tool receives the netlist along with the possible memory array size and produces the execution sequence.

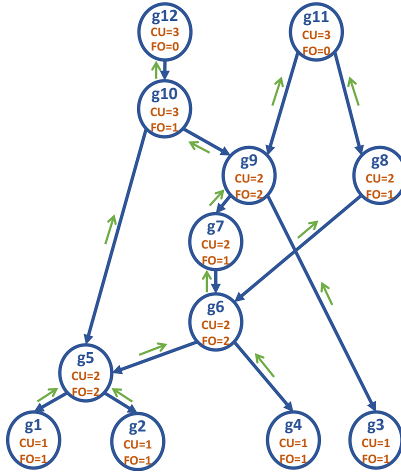


Fig. 6. A reduction from one-bit full adder netlist (Figure 3a) to a directed acyclic graph (DAG), where each vertex represents a gate and each blue edge represents a wire. The green arrows represent the direction the signals propagate in the original netlist. The FO and the CU are, respectively, the Fan Out and the Cell Usage values of each vertex. The order of the execution sequence is determined according to the CU values.

and then deciding to which free cell each gate is allocated. The order influences the number of cells required for the entire execution, as some gates may be freed earlier than others; thus more cells may be reused during prior stages of the computation. To minimize the required number of cells, we use a register allocation technique that minimizes the number of registers necessary for a computation [30]. This technique requires first performing a *reduction from the NOR and NOT netlist to a DAG (Directed Acyclic Graph)*:

$$vertex_i \leftarrow gate_i \tag{2}$$

$$edge_i \leftarrow wire_i \tag{3}$$

for all i gates in the netlist. Therefore, the DAG's roots are the gates whose outputs are connected only to the function outputs, and its leaves are the gates whose inputs are connected only to the function inputs. Additionally, each vertex receives a Fan Out (FO) value according to the number of vertices its output is connected to. For example, a reduction of the one-bit full adder netlist from Figure 3a to a DAG is shown in Figure 6.

To maximize the number of cells that may be reused at a given time, the following two principles should be considered while determining the execution order:

- *Use Depth-First Search (DFS)* rather than Breadth-First Search (BFS) [31], $i.e.$, start from the root node(s) and traverse along each branch as far as possible. A node V is inserted into the execution sequence on the way back up if either: (1) V is a leaf, or (2) all children of V are in the sequence. For example, if the execution order of the DAG in Figure 6 is determined according to BFS: $g1 \rightarrow g2 \rightarrow g4 \rightarrow g3 \rightarrow g5$, then after five cell allocations, only two may be reused $(g1, g2)$. However, with DFS, the order is $g1 \rightarrow g2 \rightarrow g5 \rightarrow g4 \rightarrow g6$, and then after five cell allocations, four cells may be reused $(g1, g2, g5, g4)$.
- *The quality of the execution order depends on the order of traversal among the node's children.* Our heuristic for determining the order is to first execute sub-graphs that require more cells for their execution. For example, in Figure 6, $g6$ has two sub-graphs: $g5$ and $g4$, which require, respectively, execution of three cells and a single cell. If sub-graph $g5$ is executed first, then at least three cells are required for the execution of $g6$ $(g5, g4, g6$, in the cycle that $g6 = NOR(g4, g5)$ is executed, since the cells of $g1, g2$ are reused). On the other hand, if sub-graph $g4$ is executed first, then at least four cells are required for the execution of $g6$ $(g4, g1, g2, g5$, in the cycle that $g5 = NOR(g1, g2)$ is executed).

To determine the minimum number of cells required for the execution of each sub-graph, our algorithm follows the generalized Strahler algorithm proposed in [30], which uses the Strahler number [32] to determine the minimal number of registers needed to compute an arithmetic expression tree. In the SIMPLER algorithm, this number is called Cell Usage (CU), and it is calculated for each vertex. The CU represents an estimation for the number of memory cells known to be sufficient for executing the sub-tree of each vertex. As previously stated, the execution of a MAGIC logic gate also requires a memory cell for its output. Therefore, $CU + 1$ of a vertex is the estimated sufficient number of memory cells for the entire execution of that vertex (after execution of all its descendants and reusing cells when possible). For example, as discussed in the previous paragraph, to execute $g6$ from Figure 6, three cells are enough when choosing the right execution order $(CU(g6) = 2)$.

The Strahler algorithm [30] is intended for trees only. However, in our case the netlist is reduced to a DAG (Directed Acyclic Graph), since some gates may be connected to several parents or ancestors, $i.e.$, their FO is larger than 1. As a result, the $CU + 1$ value does not predict the minimum number of cells accurately. For example, $CU(g9) = 2$, but when the execution order is $g1 \rightarrow g2 \rightarrow g5 \rightarrow g4 \rightarrow g6 \rightarrow g7 \rightarrow g3 \rightarrow g9 \rightarrow g8$, four cells are required when $g9$ is executed. The reason is that $FO(g6) = 2$; thus, while executing $g9$,

$g6$ is still needed for the execution of $g8$. Therefore, $g6$ may not be freed after $g7$ is executed. Consequently, during the execution of $g9$, four cells are occupied by the following gates: $g6, g7, g3, g9$. As a result, determining the execution order according to the Strahler number alone, while perfectly correct, does not necessarily produce the optimal execution. More optimal cell ordering should be evaluated in future work.

B. Mapping Execution Sequence of Logic Functions into a Single Row

The mapping algorithm, presented in Algorithm 1, receives the netlist as a directed graph $G = (V, E)$ representation (V and E are, respectively, the sets of nodes and edges), along with the number of cells dedicated for the computation (row size), and produces a mapping of the nodes to that limited-size row. In the initial state of the memory, when a computation is started, all inputs are stored in a single row, in adjacent cells. (This is not in fact mandatory for in-memory execution of MAGIC, but the algorithm works this way for simplicity and without loss of generality since the locations of the inputs are irrelevant because the mapping is arbitrary.)

The mapping is done by traversing all vertices of the DAG twice: (a) once to determine the execution order by calculating a CU value for each vertex, and (b) again, using the order imposed by the CU values, to allocate the vertices into the available memory cells. Both traversals are done by starting from all the DAG roots (the gates that produce an output of the netlist, $e.g.$, vertices $g12$ and $g11$ in Figure 6) to the leaves (gates connected only to the input(s) of the netlist, $e.g.$, vertices $g1 - g4$ in Figure 6).

The stages of the mapping are:

1) Stage 1 - Compute the Cell Usage value for each vertex: In this stage the CU value is computed for each vertex, by function *ComputeCU*, as detailed in Algorithm 2. The CU is computed by traversing all of the vertices, starting from each one of the roots, and continuing to the leaves. A CU value is assigned to a vertex only if all its children were already given a CU value. The CU of a given vertex V is determined according to the following rules:

- If V is a leaf ($i.e.$, all of the inputs of V are also inputs of the function): $CU(V) = 1$.
- Else: Sort all N children of V by descending order of their CU values. Then:
  $CU(V) = max\{CU(V_{child,i})) + i - 1\}, \forall i = (1\ to\ N)$.
  Explanation: when executing child i, all $i - 1$ children must already be allocated, and $CU(V_{child,i})$ is the estimation of the number of cells necessary for executing child i's sub-tree. Therefore, for the execution of child i, the number of cells is $CU(V_{child,i}) + i - 1$. Therefore, the number of cells sufficient for executing V's sub-tree ($CU(V)$) is $max\{CU(V_{child,i}))+i-1\}, \forall i = (1\ to\ N)$.

An example of the CU values of a single-bit full adder is given in Figure 6.

2) Stage 2 - Allocate the gates to the memory cells: In this stage, the gates are allocated to the memory cells, and each allocation is assigned an execution clock cycle number t. This number is assigned by functions *AllocateRow* and *AllocateCell*, which are detailed in Algorithms 3 and 4, respectively. The graph is traversed again, starting from the roots, and continuing to the leaves. The child with the larger CU value is traversed first. For example, when traversing the graph in Figure 6, $g5$ is traversed before $g4$ from $g6$, since $CU(g5) = 2 > CU(g4) = 1$. When a gate whose children were already allocated to cells ($i.e.$, executed) is reached, it can also be executed (since all its inputs are ready). For example, in Figure 6, the execution order when starting with the root $g11$ is either:

(a) $g1 \rightarrow g2 \rightarrow g5 \rightarrow g4 \rightarrow g6 \rightarrow \mathbf{g8} \rightarrow g7 \rightarrow g3 \rightarrow g9 \rightarrow g11 \rightarrow g10 \rightarrow g12$

or:

(b) $g1 \rightarrow g2 \rightarrow g5 \rightarrow g4 \rightarrow g6 \rightarrow g7 \rightarrow g3 \rightarrow g9 \rightarrow \mathbf{g8} \rightarrow g11 \rightarrow g10 \rightarrow g12$

since $CU(g8) = CU(g9)$; thus no priority between them is defined. In future work, priorities between vertices with equal CU values and priorities between different roots should be explored. The gate that is ready for execution is allocated to a free cell, and t is incremented by 1. If no free cells are left, the cells that store the outputs of the gates that are no longer needed are all re-initialized in a single cycle; thus t is also incremented by 1. Consequently, there is more space available for mapping the outputs of the next gates; thus, the gate is allocated, with another increment of t.

To determine that the output of a gate is not needed as input for future gates, the Fan Out (FO) value is computed for each gate. When a gate is allocated, the FO values of all gates connected to its inputs are reduced by one. Therefore, during the initialization cycle, the cells of the gates with $FO = 0$ may be freed and reused, since they are not needed as inputs for any future gate. For example, the mapping of a single-bit full adder into 9 cells is given in Figure 4. The total number of cycles is 14, 2 of which are dedicated for initialization. In the first initialization cycle, gates $g1$, $g2$ and $g4$ are reused. However, $g5$ and $g6$ cannot be freed yet, since $FO(g5) = FO(g6) = 2$ and during the first initialization cycle only one of their parents nodes was executed ($g6$ is parent of $g5$ and $g8$ is parent of $g6$).

When all the netlist gates are mapped, the mapping tool prints the produced mapping and the required number of initialization cycles. When there is no possible mapping, the tool reports it.

C. SIMPLER Complexity

We evaluated SIMPLER complexity and found it to be $O(|V|)$, where $|V|$ is the number of vertices in the graph (# of gates). The analysis is as follows (where $|E|$ is the number of edges (wires), and N is the row size used for allocation):

1) The maximum number of edges entering a vertex depends on the type of gate used: 2 and 4 for NOR gates with, respectively, 2 and 4 inputs. Hence $|E| < 4 \cdot |V|$, or $O(|E|) \leq O(|V|)$.
2) We use DFS in both stages. Classical DFS has complexity of $O(|V| + |E|)$; hence, in our case it is $O(|V|)$.
3) In stage 1, for each vertex we sort the incoming edges according to the CU of the vertices they came from. Since the number of incoming edges is limited to 2 or 4, this is basically a (small) constant cost.

Algorithm 1 SIMPLER MAGIC

Inputs:
(1) Directed graph G = (V,E) (G, $V = \{V_1, \dots, V_{|V|}\}$ and E represent the NOR & NOT netlist, gates and wires, respectively).
(2) N = number of cells in the row (including cells for storing the function inputs and outputs).
Output: A T-tuple, where T is the number of clock cycles of the entire execution. Each element of the tuple details the inputs & output of the executed gate along with the cell numbers they are allocated to.
Initially: Without loss of generality, all $\mathcal{I}$ netlist inputs are stored in adjacent cells of a single row (columns (0 to $\mathcal{I} - 1$)). Therefore the number of cells dedicated for the computation is $N - \mathcal{I}$.

All variables are global (thus available in all functions):
$\forall$i $\in \{1, \dots, | V |\}$: C(V$_i$) $\leftarrow$ $set\ of\ children\ of$ V$_i$
$\forall$i $\in \{1, \dots, | V |\}$: P(V$_i$) $\leftarrow$ $set\ of\ parents\ of$ V$_i$
ROOTs $\leftarrow$ $set\ of\ all\ roots\ of\ G$
t = 0 \\ $number\ of\ clock\ cycles$
$\forall$i $\in \{1, \dots, | V |\}$: CU(V$_i$) $= 0$
\\ $CU(V_i)\ is\ the\ Cell\ Usage\ value\ of\ V_i$
$\forall$i $\in \{1, \dots, | V |\}$: $FO(V_i) =| P(V_i) |$
\\ FO(V$_i$) $is\ the\ Fan\ Out\ (= number\ of\ parents)\ of\ V_i$
$\forall$i $\in \{1, \dots, | V |\}$: map(V$_i$) $= 0$
\\ $map(V_i)\ is\ the\ number\ of\ the\ cell/column\ V_i\ is\ mapped\ to$
$\forall$i $\in \{0, \dots, \mathcal{I} - 1\}$: UsedList.insert(cell(i))
$\forall$i $\in \{\mathcal{I}, \dots, N - 1\}$: AvailableList.insert(cell(i))
InitList $\leftarrow \Phi$
\\ $AvailableList,\ UsedList\ and\ InitList\ are\ linked\ lists\ with\ double\ pointers$
\\ $Each\ cell\ number\ i\ (i \in \{0, \dots, N-1\})\ of\ the\ row\ is\ represented$
\\ $by\ element\ cell(i),\ which\ is\ located\ in\ the\ corresponding\ linked\ list$
\\ $according\ to\ its\ state$
\\ $The\ cell\ states\ are:$
\\ $1.\ available\ -\ a\ cell\ waiting\ to\ be\ used\ for\ the\ computation$
\\ $2.\ used\ -\ an\ already\ used\ cell\ for\ the\ computation$
\\ $3.\ init\ -\ a\ cell\ that\ may\ be\ reused\ but\ initialization\ is\ required$
  for all $r \in ROOTs$ **do**
    computeCU(r)
  end for
  for all $r \in ROOTs$ **do**
    if AllocateRow(r) == FALSE **then**
      return FALSE \\ $Cannot\ find\ mapping$
    end if
  end for
  return TRUE
  \\ $A\ mapping\ of\ the\ entire\ netlist\ is\ found$

4) In stage 2, allocating a cell for a vertex involves searching for a free cell and changing its state as needed. The complexity of this search is $O(1)$, achieved by using linked-lists to link all cells with the same state.

All in all, SIMPLER complexity is linear with the number of vertices in the graph. This low complexity results in a fast execution time. As an example, SIMPLER maps a graph with over 12K vertices, independent of the memory row size, in less than 0.6 seconds on a client notebook (HP EliteBook 840, Intel Core i7, 16GB RAM, 512GB SATA SSD).

IV. Experimental Results and Evaluation

We evaluate the SIMPLER synthesis and mapping tool by calculating the latency, throughput, area, and area efficiency of each benchmark execution using the Python-based tool we developed. Additionally, we compare SIMPLER to other tools by assessing its ease of use and the time it takes to generate the mapping. The netlists we use consist of N-input NOR gates, where either: (1) $N \in \{1, 2\}$, $i.e.$, NOT gates and two-input NOR gates ($NOR2$), or (2) $N \in \{1, 2, 3, 4\}$ ($NOR4$). The evaluation consists of three parts:

1) Comparison to other mapping tools: SIMPLE MAGIC [23], Yadav *et al.* [24] (referred as *YADAV* for the rest of the paper) and SAID [25]. For each comparison, we use the benchmark suites used by the original authors. For both SIMPLE and YADAV we use NOR2 netlists, and for SAID we use NOR4, similarly to each work. Tables I, II and III list the results of the comparison with, respectively, SIMPLE MAGIC, YADAV and SAID.
2) Evaluation of SIMPLER on the EPFL combinational benchmark suite [33]. EPFL is a quite large, modern benchmark suite designed to challenge modern logic optimization tools. The trade-off between the area and the performance (latency and throughput) is examined. The SIMPLER tool is very efficient and produces the mapping for each of the EPFL benchmarks within seconds. The SIMPLER EPFL results are given for both NOR2 and NOR4. These results will be useful for comparison with future work, and are detailed in Table IV.
3) Comparison to Optimal SIMPLER (*OptiSIMPLER*[3]): OptiSIMPLER tries to determine the minimum area (in terms of number of cells) required for the execution of

Algorithm 2 Function computeCU

\\ $computeCU(V_i)$ - Receives a vertex V_i and computes its Cell Usage (CU) (recursive function, thus when receives a root computes the CU value for all its descendants):
\\ $CU(V_i)\ is\ the\ Cell\ Usage\ value\ of\ V_i$
\\ $C(V_i)\ is\ set\ of\ children\ of\ V_i$
\\ $C(V_i)^{(j)}\ is\ child\ j\ of\ V_i$

int function *computeCU(V_i)*
  if $CU(V_i) > 0$ **then**
    \\ $CU(V_i)\ was\ generated\ already$
    return $CU(V_i)$
  else if $C(V_i) == \phi$ **then**
    \\ $V_i\ has\ no\ children\ \rightarrow\ V_i\ is\ connected\ to$
    \\ $function\ inputs\ only$
    return $CU(V_i) = 1$
  else
    for $j = 1, \dots, |C(V_i)|$ **do**
      $CU(C(V_i)^{(j)}) \leftarrow computeCU(C(V_i)^{(j)})$
    end for
    $C'(V_i) \leftarrow$ sorted($C(V_i)$)
    \\ $'sorted'\ sorts\ C(V_i)\ by\ CU\ values\ in$
    \\ $descending\ order\ indexed\ k$
    return $max_j\{C'(V_i)^{(k)} + k - 1\}$
  end if
end function *computeCU*

Algorithm 3 Function AllocateRow

\\ $AllocateRow(V_i)$ - Receives a vertex V_i and returns $TRUE$ only if an allocation is found for it and for all its descendants (recursive function):
\\ $CU(V_i)\ is\ the\ Cell\ Usage\ value\ of\ V_i$
\\ $C(V_i)\ is\ set\ of\ children\ of\ V_i$
\\ $map(V_i)\ is\ the\ number\ of\ cell\ (column)\ the\ V_i$ is
\\ $mapped\ to$
\\ $FO(V_i)\ is\ the\ Fan\ Out\ (= number\ of\ parents)\ of\ V_i$
bool function *AllocateRow(V_i)*
  for all $V_j \in$sorted(C(V$_i$)) **do**
    \\ $'sorted'\ sorts\ C(V_i)\ by\ CU\ values\ in$
    \\ $descending\ order$
    if $map(V_j) == 0$ **then**
      if $AllocateRow(V_j) == FALSE$ **then**
        return FALSE
      end if
    end if
  end for
  if $map(V_i) == 0$ **then**
    \\ $V_i\ is\ not\ mapped\ yet$
    $map(V_i) \leftarrow AllocateCell(V_i)$
    if $map(V_i) == 0$ **then**
      \\ $V_i\ could\ not\ be\ mapped$
      return FALSE
    end if
  end if
  return TRUE
end function *AllocateRow*

[3]The OptiSIMPLER tool may be found at:
https://github.com/debjyoti0891/arche.git

Algorithm 4 Function AllocateCell

```
\\ AllocateCell(V_i) - Receives a vertex V_i and returns the cell (column) number
it is allocated to, and 0 if it fails:

int function AllocateCell(V_i)
    FreeCell = AvailableList.GetFirst()
    \\ List.GetFirst() returns the first element of List
    \\ or Φ when List is empty
    if FreeCell == Φ then
        \\ No available cell; therefore, all init cells
        \\ are initialized simultaneously :
        if InitList.IsNotEmpty() == True then
            \\ List.IsNotEmpty() returns True if List is not empty
            \\ i.e., at least one cell ready for initialization exists
            AvailableList.concatenate(InitList)
            \\ List1.concatenate(List2) concatenates List2 to List1
            InitList ← Φ
            FreeCell = AvailableList.GetFirst()
            t+ = 1 \\ Initialization cycle − increment # of cycles
        else
            return 0 \\ No cells to initialize
        end if
    end if
    UsedList.Insert(FreeCell)
    \\ List.Insert(Element) inserts Element to List
    AvailableList.DeleteFirst()
    \\ List.DeleteFirst() deletes the first element from List
    \\ i.e., first available cell is now Used
    t+ = 1 \\ Allocate to cell − increment # of cycles
    for all V_k ∈ C(V_i) do
        \\ Update the FO value of the allocated gate :
        FO(V_k)− = 1
        if FO(V_K) == 0 then
            InitList.Insert(cell(map(V_k)))
            UsedList.delete(cell(map(V_k)))
            \\ List.delete(Element) deletes Element from List
            \\ i.e. moves cell(map(V_k)) from UsedList to InitList
        end if
    end for
    return FreeCell
end function AllocateCell
```

a function using a single row of the memory by reusing cells, and then to find the minimal latency possible for this area. We developed OptiSIMPLER as part of our work on SIMPLER to assess SIMPLER's benefits. OptiSIMPLER works by solving an optimization problem, using the Z3 SMT solver [34]. This tool evaluates the quality of SIMPLER's heuristics by comparing the minimum number of cells proposed by SIMPLER to the optimal solution generated by OptiSIMPLER. Since solving optimization problems is computationally cumbersome, its run-time has to be restricted. We allowed OptiSIMPLER to run no more than 2 days per benchmark. With this limit, OptiSIMPLER succeeds in mapping only small benchmarks (less than 100 gates). Therefore, it succeeds in running only the LGsynth91 benchmark suite [35]. OptiSIMPLER and SIMPLER are compared in Table V, where both map NOR4 netlists.

A. Description of the Compared Previous Works

We compare SIMPLER to three previously developed mapping tools: SIMPLE MAGIC [23], YADAV [24] and SAID [25].

1) SIMPLE MAGIC [23] - solves an optimization problem that minimizes the latency of the execution of a function in-memory. As opposed to SIMPLER, SIMPLE uses several memory rows for the execution of a single computation instance, with the goal of reducing the latency of a single computation instance. For the same reasons as OptiSIMPLER, SIMPLE can map very small benchmarks only; thus, we evaluated SIMPLE using the LGsynth91 benchmark suite [35].
2) YADAV [24] - uses heuristics to map the execution of larger functions into the memory. It can process larger benchmarks than SIMPLE. The authors use the ISCAS85 benchmark suite [36] for their evaluation. To execute the larger benchmarks within an array with limited-size rows, several rows are used for a single computation instance and data is copied among the different rows occasionally. Each copy of a single bit takes two clock cycles (two NOT operations), thus increasing the execution latency.
3) SAID [25] - similar motivation and general approach as YADAV, but employing a different mapping technique and using the IWLS'93 benchmark set.

B. Description of the Criteria for Efficient Mapping

Latency is the number of cycles to complete a computation. A MAT (memory array) may contain several computation instances. The latency of executing all computation instances in the given MAT (#CycAllInst) is higher than the execution latency of a single computation instance (#CycSinInst), unless all computation instances can be executed in parallel. In SIMPLER, where all instances are simultaneously executed, the overall latency of all instances is equal to the latency of a single instance, which is equal to the number of executed gates + the number of initialization cycles (independent of the number of instances). However, in SIMPLE, YADAV and SAID (which aimed to improve the latency of a single instance), when parallelized in a similar manner to SIMPLER, the latency of all computation instances increases with the number of instances. When the computation instances are aligned by rows ($row\ alignment$), MAGIC column operations of different instances are executed in parallel, and MAGIC row operations are serially executed, and vice versa when the instances are aligned by columns ($col\ alignment$), as shown in Figure 7. Hence, $\#CycAllInst = \#Cr \times \#Ir + \#Cc \times \#Ic$, where $\#Cr$ and $\#Cc$ are, respectively, the number of cycles in which MAGIC row and MAGIC column operations are executed. $\#Ir$ and $\#Ic$ are, respectively, the number of instances that can be executed within the given MAT in the $row\ alignment$ and $col\ alignment$ configuration ($i.e.$, respectively, two and three instances in Figure 7). The tools developed by YADAV and SAID are not public; thus, we could not evaluate the exact $\#Cr$ and $\#Cc$. Therefore, for both we optimistically assume a latency equal to that of executing a single instance, which is a very loose lower-bound on the latency of executing all the instances. This bound can only be reached when each instance is executed using a single row. For the rest of the section, latency of SIMPLE refers to $\#CycAllInst$, and of YADAV and SAID refers to $\#CycSinInst$.

Throughput is the maximum number of instances that can be executed within the MAT within a given time unit, as stated in Equation 1. Maximal throughput improvements can only be achieved for well-parallelizable code. To assess the practical benefit for a specific kernel/application, a comprehensive system analysis is needed. Additionally, physical restrictions [26] may also limit the number of rows that can be executed in parallel, thus reducing the throughput gain proportionally.

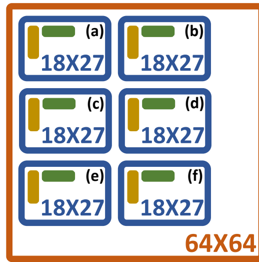


Fig. 7. A 64×64 memory array. Each blue rectangle is a computation instance computed within the array. Each instance is executed using 18×27 cells. The yellow lines represent a MAGIC column and the green lines a MAGIC row operation. When the instances are aligned by rows ($row\ alignment$), two instances can be aligned, since $\lfloor 64/27 \rfloor = 2$, $e.g.$ (a) and (b). All aligned MAGIC column operations of the different row-aligned instances may be executed in parallel, whereas the MAGIC row operations of the different instances are serially executed. When the instances are aligned by columns ($col\ alignment$), three instances can be aligned, since $\lfloor 64/18 \rfloor = 3$, $e.g.$, (a), (c) and (e). All aligned MAGIC row operations of the different column-aligned instances may be executed in parallel, whereas the MAGIC column operations of the different instances are serially executed.

In this paper, we evaluate the throughput as if the maximum parallelism is possible, $i.e.$, the number of instances equals the number of rows in the MAT ($i.e.$, 512). For SIMPLE, YADAV and SAID, we evaluated the throughput in two ways: first, the throughput for a single instance ($SinTP$), as these methods were originally intended to be executed that way. Additionally, we computed the throughput for as many instances as possible that are executed concurrently, within the given MAT ($ParTP$). $ParTP$ is only the hypothetical potential throughput of the existing methods should they employ parallel computation of different instances. Overall, $ParTP$ favors the existing methods by giving them the extra benefit of running multiple computation instances in parallel (which may be unrealizable). Also, as previously mentioned, the latency of SAID and YADAV is a loose lower bound; hence, we use a quite optimistic upper-bound for the throughput potential of these two methods.

Area is determined by $the\ number\ of\ columns \times the\ number\ of\ rows$ allocated for the execution of a single computation instance, including the area necessary for storing the function inputs, $e.g.$, $18 \times 27 = 486$ in Figure 7. For SIMPLER, the area is equal to the number of active memristors, since a single row execution enables the use of all available memristor cells for the computation. Therefore, when cell reuse is not required, the area is equal to the number of gates + the number of inputs. Otherwise, when cell reuse is required, the area is equal to the number of columns allocated for the computation (which is smaller than the allocated area with no reuse). On the other hand, for all three tools, the area is much larger than the number of active memristors. This is because the processing is distributed over several rows; hence, they do not use all the memristors in the array for the executions.
The *area efficiency* is calculated as $\frac{1}{Latency \times Area}$.

C. SIMPLER Results

Tables I, II and III (first three columns) list the number of inputs (#In), outputs (#Out), and the number of all gates (both two-input NOR and NOT) after the ABC optimization (ABC, #Gates), which is equivalent to the number of pure execution steps (latency) in SIMPLER (that is, without re-initialization cycles). The next seven columns of Table I and six columns of Tables II and III, show the results for SIMPLE, YADAV and SAID, using a memory array size of 512×512 (the last three benchmarks in Table II use a 1024×1024 array, according to the area required for the execution of YADAV). First, the latency necessary for execution is given. For SIMPLE, the number of cycles for executing a single instance (#CycSinInst) and the number of cycles for executing all instances in the MAT (#CycAllInst) are given in two different columns. As discussed in Section IV-B, the given latency for YADAV and SAID is a loose lower bound equal to the number of cycles for executing a single instance (#CycSinInst). The next columns show the maximum number of computation instances that fit the given array size (#inst), the throughput (TP), the number of active memristors used for the execution (#Mem), the area necessary for the computation of a single instance (Area), and the area efficiency (AreaEff). The next four columns list the results of the proposed SIMPLER synthesis tool, when executing within a row with the minimum number of cells required by SIMPLER for the execution of each benchmark (referred as #MinCells) and an additional $max\{5\%\ of\ \#MinCells, 10\}$ cells. #Inst is not stated for each benchmark, since SIMPLER maps the computation to a single row; thus, the number of instances that can be executed concurrently equals the number of rows in the array (1024 for the last three benchmarks in Table II, and 512 for all other benchmarks in Tables I, II and III). Area and #Mem are listed together in one column, as they are the same for SIMPLER since only a single row is used. Finally, the area efficiency (AreaEff) is listed.

The comparisons between all three works (SIMPLE, YADAV and SAID) and SIMPLER are given, respectively, in the last five columns (Comparison) of Tables I, II and III. All averages are relative numbers computed as geometric mean. All compare the number of cycles, throughput, and area efficiency of SIMPLE, YADAV and SAID to SIMPLER, calculated as $\frac{SIMPLER\ results}{SIMPLE/YADAV/SAID\ results}$. Additionally, the area compression (AreaCom) is calculated as $\frac{SIMPLE/YADAV/SAID\ Area}{SIMPLER\ Area}$. SIMPLER is area efficient since it can reuse cells; therefore, it uses on average $24\times$ less area as compared to YADAV, $22\times$ less than SAID and $6.5\times$ less than SIMPLE. SIMPLER achieves $9.8\times$ better (lower) average latency than SIMPLE, when executing all instances. Additionally, SIMPLER achieves 2.6% better average latency than YADAV. On the other hand, SAID achieves 42% better average latency than SIMPLER (note that the latency of YADAV and SAID is for a single instance only). For SIMPLE, the latency for executing all instances is $14\times$ greater than for the execution of a single instance only. Assuming the latency overhead for executing all instances by YADAV and SAID is similar, SIMPLER achieves $14\times$ better average latency. Thus, the bound on latency is very loose. Overall, SIMPLER exhibits higher throughput than previous work. Compared to the original versions of YADAV, SAID and SIMPLE, all of which operate on a single computation instance at a time, SIMPLER, respectively, achieves higher throughput (SinTP) of $526\times$, $360\times$ and $332\times$. When previous work is parallelized,

SIMPLER achieves $4.9\times$ better average throughput (ParTP) than SIMPLE, and at least $9.3\times$ and $2.7\times$ better throughput than, respectively, YADAV and SAID (again, with latency of a single instance), when using the loose upper bound on the throughput of SAID and YADAV. Additionally, SIMPLER achieves at least $25\times$ and $16\times$ better area efficiency than, respectively, YADAV and SAID, and $63\times$ better area efficiency than SIMPLE.

Table IV lists the results for executing the EPFL combinational benchmark suite within the memristive memory using the SIMPLER algorithm. The first three columns are similar to Tables I, II and III. The next eight columns of Table IV list the results using NOR2 netlists. First, the number of gates is given (ABC, #Gates). Then, the latency (#Cyc) and area (Area) of three cases are listed: when executing using (1) an unlimited number of cells (UnlimitCells), *i.e.*, an array with enough columns to execute with no initialization cycles, (2) the minimum number of cells required by SIMPLER (MinCells), and (3) the minimum number of cells required by SIMPLER with an additional $max\{5\%\ of\ \#MinCells, 10\}$ (MinCells+5%/10). These additional cells decrease computation latency with a relatively low area cost. Re-initializing all desired cells within a row might take more than a single clock cycle, depending on the size of the row and the voltage applied to it [37]. The last column of the NOR2 results shows the latency (LimitInitCells, #Cyc) when no more than 10 cells can be initialized simultaneously (during the same clock cycle), while the area is equal to MinCells+5%/10. The last eight columns list similar results when using NOR4 netlists.

The number of cycles and the area for the UnlimitCells case are compared to the following four cases: (1) MinCells, (2) MinCells+5%/10 (3) LimitInitCells and (4) lowest-bound Area (#In+#Out). The results for NOR2 are given in Figure 8a, where all averages are relative numbers computed as geometric mean. When using the minimum required number of cells (MinCells), the area decreases by $5.8\times$ on average as compared to UnlimitCells, at the cost of a 6.2% average increase in the number of cycles. Adding a small percentage of cells (MinCells+$5\%/10$) reduces the overhead of initialization cycles from 6.2% to a Geomean of 2.3%, at the cost of a small increase in area (7.5%). In the LimitInitCells case, the latency increases by 10% on average as compared to executing with no limitation on the number of initialized cells with UnlimitCells. In general, when the number of initialized cells is increased to A, the relative latency increases by less than ($1+\frac{1}{A}$). For example, when the number of initialized cells is increased to 25, the latency increases to less than 4%. Under the assumption that 25 cells or more can be initialized simultaneously, the latency overhead is small; therefore, for all other results we ignore the limitation on the number of cells that can be initialized during the same clock cycle. The lowest-bound area is the minimal theoretical possible area required for the execution, as if cells store only the inputs and outputs of the function, assuming that no cells are needed for execution of the intermediate gates. This bound indicates how close the obtained mapping is to the theoretical lowest limit. For example, although benchmark dec reduces the area to only 76%, the lowest-bound area shows it may not be reduced below 72%, since dec inputs and outputs occupy most of the cells.

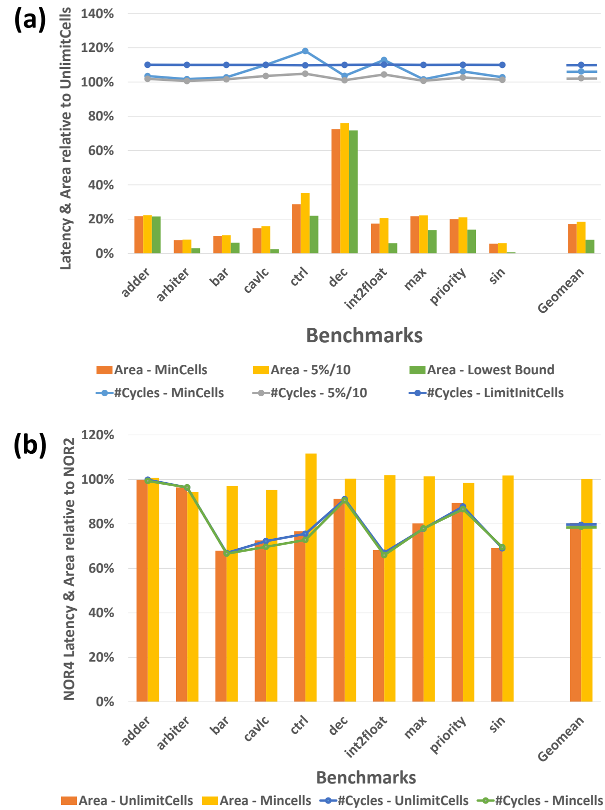


Fig. 8. The results of EPFL benchmarks for (a) NOR2 and (b) NOR4 as compared to NOR2.

In Figure 8b, the NOR2 and NOR4 configurations are compared as $\frac{NOR4results}{NOR2results}$, for UnlimitCells and MinCells. In the UnlimitCells case, the area and latency decrease, respectively, by an average of 19.6% and 20.5% when using NOR4 netlists. With MinCells, the latency decreases by an average of 21.3% compared to NOR2, similar to the relative decrease in latency; however, the area is almost similar (increases by a Geomean of 0.2%) for NOR2 and NOR4. The reason is that the fan-in of NOR4 gates (*i.e.*, the number of children) is higher than for NOR2, and the cells that store the children cannot be freed for reuse until the gate is executed. Therefore, when using NOR4 gates, fewer cells may be freed and reused at a given time. As a result, the area when using MinCells with NOR4 (*i.e.*, the number of reuse cycles is large) is similar to MinCells with NOR2.

Table V compares the minimum number of cells required for different executions using SIMPLER and OptiSIMPLER. The first four columns are similar to Tables I, II and III. The next two columns detail the minimal number of gates for which the SAT solver found a mapping (SAT) and the minimal number of gates for which the SAT solver found that no mapping is possible (UNSAT). In all benchmarks except mux, the SAT solver found the mapping within the 2-day time limit using the minimum number of cells ($SAT_Area - UNSAT_Area =$ 1). In mux, the SAT solver could not decide on the optimal mapping under this limited run-time, (SAT_Area = 30 and $UNSAT_Area$ = 27; thus, a mapping with 29 or 28 cells might also exist). As can be seen in the table, SIMPLER

succeeded in generating a mapping using only 29 cells in mux, meaning that it produced a better mapping than limited-run-time-OptiSIMPLER. In the next column, the optimal number of cycles (#Cyc) for the chosen mapping (SAT case) is listed. The next two columns list the minimum number of gates for which SIMPLER found a mapping with (MinCells) and the number of cycles (#Cyc) required for each benchmark in SIMPLER. The last two columns list the difference between the area and number of cycles of the minimized mapping found by OptiSIMPLER (SAT case) and SIMPLER.

Although SIMPLER is not optimal, it successfully generates mappings with only 0.8 additional cells on average, compared to the optimal mapping produced by OptiSIMPLER. However, SIMPLER reduces the number of cycles by an arithmetic average of 1.1, compared to OptiSIMPLER. Since the results are given for small benchmarks only, a more accurate evaluation may require further work.

In addition to comparing the numerical results, we also compare the ease of use among the different tools:

- *Flexibility*: For SIMPLER, the mapping is done into a single dimension. SIMPLE, YADAV and SAID, on the other hand, generate a mapping using a two-dimensional array. Therefore, the process for the mapping generation is much shorter and simpler for SIMPLER.
- *Overhead on memory periphery and control:* In SIMPLER, either MAGIC row or MAGIC column operations are used. In contrast, SIMPLE, YADAV and SAID use both operations. Therefore, they use a transpose memory [17], which has a larger and more complex periphery. Additionally, SIMPLE, YADAV and SAID require scattered execution of gates among rows and columns, thus complicating the memory controller.
- *Function input and output locations*: For SIMPLER, the function inputs and outputs can be located more naturally, $i.e.$, adjacent to each other, with no overhead. On the other hand, the inputs and outputs in SIMPLE and SAID are located to allow optimal latency. In YADAV, if the inputs are in adjacent cells, copy cycles are necessary to align them for the execution of the gates they feed, and the outputs are moved to their final locations after they are ready.

V. Conclusion

This paper presents an automatic logic synthesis flow called SIMPLER (*Synthesis and In-memory MaPping of Logic Execution in a single Row*) for optimizing the throughput of in-memory SIMD computations. SIMPLER automatically generates a sequence of MAGIC NOR gates and then maps the execution of a single instance of a desired logic function to a single size-limited row, reusing cells as needed. Mapping a computation into a single row allows numerous instances to be executed **in parallel**, according to the number of rows dedicated to the computation, thus **dramatically improving the throughput**. The SIMPLER algorithm uses heuristics to reduce the complexity of mapping the computation in-memory; thus, SIMPLER can quickly generate an optimized mapping for huge benchmarks. The optimized mappings that SIMPLER generates are the basis for designing an efficient memristive memory processing unit (mMPU) controller. Hence, SIMPLER is a stepping stone towards a powerful mMPU.

Our experimental results show that SIMPLER yields an average throughput improvement of $435\times$ compared to Yadav *et al.* and SAID (which optimize the latency, rather than throughput). When these previous tools are parallelized in a similar manner to SIMPLER, SIMPLER achieves a throughput improvement of at least $5\times$, with at least $23\times$ better area and at least $20\times$ better area efficiency, at the cost of up to 17% average latency degradation.

References

[1] A. Pedram, S. Richardson, S. Galal, S. Kvatinsky, and M. A. Horowitz, “Dark Memory and Accelerator-Rich System Optimization in the Dark Silicon Era,” *IEEE Design Test*, vol. 34, no. 2, pp. 39–50, Apr. 2017.

[2] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, 5th ed. Elsevier, 2014.

[3] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, “A case for intelligent RAM,” *IEEE Micro*, vol. 17, no. 2, pp. 34–44, March 1997.

[4] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, “Ambit: In-memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 ’17, 2017, pp. 273–287.

[5] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, “Compute Caches,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[6] L. Chua, “Memristor-the missing circuit element,” *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, Sep. 1971.

[7] S. Kvatinsky, E. G. Friedman, A. Kolodny, and U. C. Weiser, “The Desired Memristor for Circuit Designers,” *IEEE Circuits and Systems Magazine*, vol. 13, no. 2, pp. 17–22, Secondquarter 2013.

[8] C. Xu, X. Dong, N. P. Jouppi, and Y. Xie, “Design implications of memristor-based rram cross-point structures,” in *2011 Design, Automation Test in Europe*, March 2011, pp. 1–6.

[9] E. Linn, R. Rosezin, S. Tappertzhofen, U. Böttger, and R. Waser, “Beyond von Neumann—Logic Operations in Passive Crossbar Arrays Alongside Memory Operations,” *Nanotechnology*, vol. 23, no. 30, July 2012.

[10] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, “The Programmable Logic-in-memory (PLiM) Computer,” in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, ser. DATE ’16, 2016, pp. 427–432.

[11] Y. Levy, J. Bruck, Y. Cassuto, E. G. Friedman, A. Kolodny, E. Yaakobi, and S. Kvatinsky, “Logic operations in memory using a memristive akers array,” *Microelectronics Journal*, vol. 45, no. 11, pp. 1429 – 1437, 2014.

[12] L. Xie, H. A. D. Nguyen, M. Taouil, S. Hamdioui, and K. Bertels, “Fast boolean logic mapped on memristor crossbar,” in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, 2015.

[13] C. Xu *et al.*, “Overcoming the Challenges of Crossbar Resistive Memory Architectures,” *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 476–488, Feb. 2015.

[14] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, “’Memristive’ Switches Enable ’Stateful’ Logic Operations via Material Implication,” *Nature*, vol. 464, no. 7290, pp. 873–876, Apr. 2010.

[15] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, “Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, pp. 2054–2066, Oct 2014.

[16] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, “MAGIC - Memristor-Aided Logic,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, nov 2014.

[17] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, “Logic Design Within Memristive Memories Using Memristor-Aided loGIC (MAGIC),” *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, July 2016.

[18] R. Ben-Hur and S. Kvatinsky, “Memory Processing Unit for In-memory Processing,” in *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, July 2016, pp. 171–172.

[19] R. Ben-Hur and S. Kvatinsky, “Memristive Memory Processing Unit (MPU) Controller for In-Memory Processing,” in *Proceedings of the International Conference on the Science of Electrical Engineering (ICSEE)*, Nov 2016.

[20] A. Haj-Ali, R. Ben-Hur, N. Wald, and S. Kvatinsky, “Efficient Algorithms for In-memory Fixed Point Multiplication Using MAGIC,” *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018.

TABLE I
COMPARISON BETWEEN SIMPLE MAGIC [23] WITH 512×512 MEMORY ARRAY SIZE AND SIMPLER, BOTH USING NOR2 NETLISTS.

Benchmarks	Original Netlist		ABC	SIMPLE MAGIC [23]							SIMPLER (this work)				Comparison - SIMPLER/SIMPLE				
	#In	#Out	#Gates	#CycSinInst	#CycAllInst	#Inst	TP $\left[\frac{\#Inst}{\#cyc}\right]$	#Mem	Area $[\#cells]$	AreaEff $\left[\frac{10^6}{\#cyc \cdot \#cells}\right]$	#Cyc	TP $\left[\frac{\#Inst}{\#cyc}\right]$	Area=#Mem $[\#cells]$	AreaEff $\left[\frac{10^6}{\#cyc \cdot \#cells}\right]$	#Cyc	AreaCom	SinTP	ParTP	AreaEff
5xp1	7	10	112	97	886	680	0.767	142	3x105	3.76	119	4.30	39	226	13.4%	8.08x	417x	5.61x	60.1x
clip	9	5	152	136	742	510	0.687	184	3x148	3.18	160	3.20	47	139	21.6%	9.45x	435x	4.66x	43.8
cm150a	21	1	62	51	570	1360	2.386	87	3x63	9.73	67	7.64	39	401	11.8%	4.85x	390x	3.20x	41.2x
cm162a	14	5	60	46	530	1360	2.566	92	3x62	10.64	64	8.00	35	468	12.1%	5.31x	368x	3.12x	44.0x
cm163a	16	5	61	45	522	1360	2.605	95	3x61	10.98	66	7.76	36	441	12.6%	5.08x	349x	2.98x	40.2x
misex1	8	7	78	45	1380	864	0.626	112	14x21	2.58	83	6.17	33	383	6.0%	8.91x	278x	9.85x	148.1x
parity	16	1	76	37	1078	1050	0.974	107	20x12	4.05	81	6.32	35	370	7.5%	6.86x	234x	6.49x	91.3x
x2	10	7	68	36	1404	1512	1.077	86	12x14	4.45	73	7.01	33	435	5.2%	5.09x	252x	6.51x	97.9
														Geomean:	10.3%	6.48x	332x	4.88x	63.2x

TABLE II
COMPARISON BETWEEN YADAV [24] WITH 512×512 MEMORY ARRAY SIZE (1024×1024 FOR LAST THREE BENCHMARKS) AND SIMPLER, BOTH USING NOR2 NETLISTS.

Benchmarks	Original Netlist		ABC	YADAV [24]						SIMPLER (this work)				Comparison - SIMPLER/YADAV				
	#In	#Out	#Gates	#CycSinInst	#Inst	TP (Upper Bound) $\left[\frac{\#Inst}{\#cyc} \cdot 10^3\right]$	#Mem	Area $[\#cells]$	AreaEff $\left[\frac{10^6}{\#cyc \cdot \#cells}\right]$	#Cyc	TP $\left[\frac{\#Inst}{\#cyc} \cdot 10^3\right]$	Area=#Mem $[\#cells]$	AreaEff $\left[\frac{10^6}{\#cyc \cdot \#cells}\right]$	#Cyc	AreaCom	SinTP	ParTP	AreaEff
c432	36	7	221	249	273	1096	290	69x13	4.69	237	2160	62	71.4	95.2%	14.5x	538x	2.0x	15.2x
c499	41	32	594	631	64	101	707	116x31	0.46	620	826	110	15.4	98.3%	32.7x	521x	8.1x	33.3x
c880	60	26	495	527	144	273	613	107x14	1.33	512	1000	142	14.4	97.2%	10.5x	527x	3.7x	10.9x
c1355	41	32	594	681	72	106	757	103x28	0.53	619	827	111	15.3	90.9%	26.0x	563x	7.8x	28.6x
c1908	33	25	569	594	75	126	648	93x33	0.58	588	871	122	14.6	99.0%	25.2x	517x	6.9x	25.4x
c2670	233	140	967	892	17	19	1183	340x29	0.12	891	575	383	3.1	99.9%	25.7x	513x	30.2x	25.8
c3540	50	22	1393	1668	36	22	1761	109x55	0.10	1434	357	192	3.8	86.0%	31.2x	596x	16.5x	36.3x
c5315	178	123	1974	1931	46	24	2251	547x22	0.05	2002	511	351	1.5	103.7%	34.3x	494x	21.5x	33.1x
c6288	32	32	2842	2916	160	55	3104	49x115	0.06	2938	349	149	2.4	100.8%	37.8x	508x	6.4x	37.5x
c7552	207	208	2241	2130	46	22	2486	542x22	0.04	2227	460	535	0.9	104.6%	22.3x	490x	21.3x	21.3x
													Geomean:	97.4%	24.4x	526x	9.3x	25.1x

TABLE III
COMPARISON BETWEEN SAID [25] WITH 512×512 MEMORY ARRAY SIZE AND SIMPLER, BOTH USING NOR4 NETLISTS.

Benchmarks	Original Netlist		ABC	SAID [25]						SIMPLER (this work)				Comparison - SIMPLER/SAID				
	#In	#Out	#Gates	#CycSinInst	#Inst	TP (Upper Bound) $\left[\frac{\#Inst}{\#cyc} \cdot 10^3\right]$	#Mem	Area $[\#cells]$	AreaEff $\left[\frac{10^6}{\#cyc \cdot \#cells}\right]$	#Cyc	TP $\left[\frac{\#Inst}{\#cyc} \cdot 10^3\right]$	Area=#Mem $[\#cells]$	AreaEff $\left[\frac{10^6}{\#cyc \cdot \#cells}\right]$	#Cyc	AreaCom	SinTP	ParTP	AreaEff
9sym	9	1	207	160	77	481	1026	70x46	2.04	218	2349	57	84.4	136.3%	56.5x	376x	4.9x	41.5x
apex5	117	88	773	777	32	41	2223	207x32	0.20	879	582	260	4.6	113.1%	25.5x	453x	14.1x	22.5x
clip	9	5	103	135	320	2370	451	50x16	9.71	114	4491	49	187.7	84.4%	16.3x	606x	1.9x	19.3x
duke2	22	29	409	300	28	93	1632	106x65	0.51	450	1138	135	17.3	150.0%	51.0x	341x	12.2x	34.0x
e64	65	65	389	134	252	1881	394	24x40	8.15	474	1080	143	15.5	353.7%	6.7x	145x	0.6x	1.9x
inc	7	9	92	55	560	10182	280	51x9	41.54	107	4785	42	233.3	194.5%	10.9x	263x	0.5x	5.6x
misex3c	14	14	501	518	27	52	2551	156x52	0.25	532	962	115	17.1	102.7%	70.5x	499x	18.5x	68.7x
rd73	7	3	99	150	252	1680	379	56x18	6.94	108	4741	44	220.7	72.0%	22.9x	711x	2.8x	31.8x
sao2	10	4	118	79	144	1823	559	54x32	7.68	128	4000	53	154.6	162.0%	32.6x	316x	2.2x	20.1x
vg2	25	8	101	55	572	10400	280	23x19	43.63	115	4452	61	149.5	209.1%	7.2x	245x	0.4x	3.4x
													Geomean:	142%	22.5x	360x	2.7x	15.8x

TABLE IV
EPFL BENCHMARKS EXECUTED USING SIMPLER WITH DIFFERENT MEMORY ARRAY SIZES (#MINCELLS, #MINCELLS + $max\{5\% \ of \ \#MinCells, 10\}$), AND #MINCELLS WITH LIMITATION ON NUMBER OF SIMULTANEOUSLY INITIALIZED CELLS (LIMITINITCELLS). COMPARISON BETWEEN RESULTS OF NOR2 AND NOR4 NETLISTS.

			NOR2								NOR4							
Benchmarks	Original Netlist		ABC	UnlimitCells		MinCells		MinCells+5%/10		LimitInitCells	ABC	UnlimitCells		MinCells		MinCells+5%/10		LimitInitCells
	#In	#Out	#Gates	#Cyc	Area	#Cyc	Area	#Cyc	Area	#Cyc	#Gates	#Cyc	Area	#Cyc	Area	#Cyc	Area	#Cyc
adder	256	129	1531	1531	1787	1585	388	1560	398	1685	1529	1529	1785	1574	391	1557	401	1682
arbiter	256	129	12798	12798	13054	13016	1016	12876	1054	14078	12330	12330	12586	12553	958	12416	994	13563
bar	135	128	4051	4051	4186	4162	429	4115	444	4457	2711	2711	2846	2772	416	2751	431	2986
cavlc	10	11	840	840	850	924	125	870	135	924	670	670	617	644	119	626	129	668
ctrl	7	26	143	143	150	169	43	150	53	157	108	108	115	123	48	112	58	118
dec	8	256	360	360	368	373	267	364	280	396	328	328	336	338	628	331	281	361
int2float	11	7	294	294	305	332	53	307	63	324	197	197	208	219	54	206	64	217
max	512	130	4200	4200	4712	4268	1020	4230	1046	4620	3268	3268	3780	3326	1034	3291	1061	3595
priority	128	8	852	852	980	905	196	885	206	938	748	748	876	784	193	766	203	823
sin	24	25	7915	7915	7939	8140	453	8019	475	8707	5463	5463	5487	5659	461	5539	483	6010

[21] M. Imani, S. Gupta, and T. Rosing, "Ultra-Efficient Processing In-Memory for Data Intensive Applications," in *Proceedings of the 54th Annual Design Automation Conference (DAC)*, 2017, pp. 1–6.

[22] A. Haj-Ali, R. Ben-Hur, N. Wald, R. Ronen, and S. Kvatinsky, "IMAGING-In-Memory AlGorithms for Image processiNG," *IEEE Transactions on Circuits and Systems I: Regular Papers (TCAS1)*, June 2018.

[23] R. Ben-Hur, N. Wald, N. Talati, and S. Kvatinsky, "Simple magic: Synthesis and in-memory Mapping of logic execution for memristor-aided logic," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 225–232.

TABLE V
COMPARISON BETWEEN THE MAPPING WITH THE MINIMAL REQUIRED NUMBER OF CELLS (OPTISIMPLER), AND SIMPLER, USING NOR4 NETLISTS.

Benchmarks	Original Netlist		ABC	OptiSIMPLER			SIMPLER (this work)		Diff (SIMPLER-OptiSIMPLER)	
	#In	#Out	#Gates	SAT - Area [#cells]	UNSAT - Area [#cells]	SAT - #Cyc	MinCells - Area [#cells]	#Cyc	Area [#cells]	#Cyc
b1	3	4	7	8	7	16	9	13	1	-3
cm138a	6	8	24	16	15	42	17	44	1	2
cm42a	4	10	14	15	14	30	16	31	1	1
cmb	16	4	49	25	24	74	27	67	2	-7
con1	7	2	22	12	11	37	13	32	1	-5
cordic	23	2	72	31	30	104	34	96	2	-8
decod	5	16	25	23	22	48	23	56	0	8
majority	5	1	9	10	9	13	10	13	0	0
mux	21	1	56	30	27	77	29	78	-1	1
xor5	5	1	18	10	9	28	10	28	0	0
								Arithmetic average:	0.8	-1.1

[24] D. N. Yadav, P. L. Thangkhiew, and K. Datta, "Look-ahead mapping of Boolean functions in memristive crossbar array," *Integration the VLSI Journal*, vol. 64, pp. 152–162, 2019.

[25] V. Tenace, R. G. Rizzo, D. Bhattacharjee, A. Chattopadhyay, and A. Calimera, "SAID: A Supergate-Aided Logic Synthesis Flow for Memristive Crossbars," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, March 2019, pp. 372–377.

[26] N. Talati, A. Haj Ali, R. Ben-Hur, N. Wald, R. Ronen, P.-E. Gaillardon, and S. Kvatinsky, "Practical Challenges in Delivering the Promises of Real Processing-in-Memory Machines," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Mar. 2018.

[27] R. Govindarajan, Hongbo Yang, J. N. Amaral, Chihong Zhang, and G. R. Gao, "Minimum register instruction sequencing to reduce register spills in out-of-order issue superscalar architectures," *IEEE Transactions on Computers*, vol. 52, no. 1, pp. 4–20, Jan 2003.

[28] R. Sethi, "Complete register allocation problems," *SIAM Journal on Computing*, vol. 4, no. 3, pp. 226–248, 1975.

[29] A. Mishchenko, "ABC: A System for Sequential Synthesis and Verification," *Berkeley Logic Synthesis and Verification Group*, 2012. [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc/

[30] D. Auber, "Using Strahler Numbers for Real Time Visual Exploration of Huge Graphs," in *Computer Vision and Graphics (ICCVG)*, 2002.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press and McGraw-Hill, 2001.

[32] A. N. Strahler, "Hypsomic Analysis of Erosional Topography," in *Bulletin Geological Society of America 63*, 1952, pp. 1117–1142.

[33] L. Amar, P.-E. Gaillardon, and G. De Micheli, "The EPFL Combinational Benchmark Suite," in *Proceedings of the 24th International Workshop on Logic Synthesis (IWLS)*, 2015.

[34] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

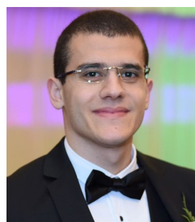
[35] S. Yang, "Logic synthesis and optimization benchmarks user guide: Version 3.0." in *MCNC*, 1991.

[36] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran," in *Proceedings of IEEE Int'l Symposium Circuits and Systems (ISCAS 85)*. IEEE Press, Piscataway, N.J., 1985, pp. 677–692.

[37] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 476–488.

Rotem Ben-Hur received her B.Sc in Electrical Engineering from the Technion - Israel Institute of Technology, in 2014. In 2012 she joined Elbit Systems as an FPGA designer. Since 2015, she is a graduate student working toward a PhD degree (direct path) at the Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion - Israel Institute of Technology. Her current research is focused on novel architectures for logic with emerging memory technologies.

Ronny Ronen is a Senior Researcher at the Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion Israel Institute of Technology. He received the B.Sc. and M.Sc. degrees in computer science in 1978 and 1979, respectively, both from the Technion Israel Institute of Technology. Ronny was in Intel from 1980 to 2017 in various technical and managerial positions. In his last role, Ronny led the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI). Until 2011, Ronny was a senior staff computer architect in the Intel Development Center in Haifa and before that the director of Microarchitecture Research in that center. In these roles, Ronny led/was involved in the initial definition and pathfinding of major leading edge Intel processors. Earlier, Ronny led the development of several system software products and tools including the Intel Pentium processor performance simulator and several compiler efforts. Ronny holds over 70 issued patents and has published over 20 papers. Ronny is an IEEE Fellow and was an Intel Senior Principal Engineer.



Ameer Haj-Ali is currently a Ph.D. student at the Department of Electrical Engineering and Computer Science at UC Berkeley. He completed his M.Sc. studies at the Andrew and Erna Viterbi Faculty of Electrical Engineering at the Technion - Israel Institute of Technology in 2018. He received the B.Sc. degree in computer engineering, summa cum laude, in 2017 from the Technion - Israel Institute of Technology. From 2015 to 2016 he was with Mellanox Technologies as a chip designer. His current research is focused on auto-tuning, reinforcement learning, ASIC design, and high performance computing.

Debjyoti Bhattacharjee received his B.Tech in Computer Science and Engineering from WBUT, India in 2013, M.Tech in Computer Science from Indian Statistical Institute, Kolkata in 2015 and PhD degree in Computer Science and Engineering from Nanyang Technological University, Singapore, in 2018. During his doctoral studies, he worked on design of architectures using emerging technologies for in-memory computing. He developed novel technology mapping algorithms, technology-aware synthesis techniques, and proposed novel methods for multi-valued logic realization. His current research is focused on hardware design automation tools and application-specific accelerator design, with emphasis on emerging technologies.

Adi Eliahu received her B.Sc. in Electrical Engineering from the Technion Israel Institute of Technology, in 2018. Since 2018, she is a graduate student working toward a M.Sc degree at the Andrew and Erna Viterbi Faculty of Electrical Engineering at the Technion. Her current research focuses on designing architectures for low-power systems using non-volatile memory emerging technologies.

Natan Peled received his B.Sc. in Computer Engineering from the Technion - Israel Institute of Technology, in 2019. In 2018 he joined Annapurna labs - Amazon, as a verification engineer. Since 2019, he is a graduate student working toward a M.Sc degree at the Andrew and Erna Viterbi Faculty of Electrical Engineering at the Technion. His current research focuses on novel architectures for logic with emerging memory technologies.

Shahar Kvatinsky is an Assistant Professor at the Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion Israel Institute of Technology. Shahar received the B.Sc. degree in Computer Engineering and Applied Physics and an MBA degree in 2009 and 2010, respectively, both from the Hebrew University of Jerusalem, and the Ph.D. degree in Electrical Engineering from the Technion Israel Institute of Technology in 2014. From 2006 to 2009, he worked as a circuit designer at Intel. From 2014 and 2015, he was a post-doctoral research fellow at Stanford University. Kvatinsky is an editor of Microelectronics Journal and has been the recipient of numerous awards: the 2019 Krill Prize for Excellence in Scientific Research, 2015 IEEE Guillemin-Cauer Best Paper Award, 2015 Best Paper of Computer Architecture Letters, Viterbi Fellowship, Jacobs Fellowship, ERC starting grant, the 2017 Pazy Memorial Award, the 2014 and 2017 Hershel Rich Technion Innovation Awards, 2013 Sanford Kaplan Prize for Creative Management in High Tech, 2010 Benin prize, and seven Technion excellence teaching awards. His current research is focused on circuits and architectures with emerging memory technologies and design of energy efficient architectures.