Skeleton-based Synthesis Flow for Computation-In-Memory Architectures

Yu, Jintao; Nane, Razvan; Ashraf, Imran; Taouil, Mottaqiallah; Hamdioui, Said; Corporaal, Henk; Bertels, Koen

DOI
10.1109/TETC.2017.2760927

Publication date
2020

Document Version
Accepted author manuscript

Published in
IEEE Transactions on Emerging Topics in Computing

Skeleton-based Synthesis Flow for Computation-In-Memory Architectures

Jintao Yu, *Student Member, IEEE,* Răzvan Nane, *Member, IEEE,* Imran Ashraf, *Member, IEEE,* Mottaqiallah Taouil, *Member, IEEE,* Said Hamdioui, *Member, IEEE,* Henk Corporaal, *Member, IEEE,* and Koen Bertels, *Member, IEEE*

***Abstract*—Memristor-based Computation-in-Memory (CIM) is one of the emerging architectures for next-generation Big Data problems. Its design requires a radically new synthesis flow because the memristor is a passive device that uses resistance to encode its logic value. This article proposes a synthesis flow for mapping parallel applications on memristor-based CIM architecture. It employs solution templates that contain scheduling, placement, and routing information to map multiple algorithms with similar data flow graphs to memristor crossbar; this template is named *skeleton*. Complex algorithms that do not fit any skeleton can be solved by nested skeletons. Therefore, this approach can be applied to a wide range of applications with a limited number of skeletons. It further improves the design when spatial and temporal patterns exist in input data. To accelerate simulation of generated SystemC models, we integrate MPI in skeletons. The synthesis flow and its additional features are verified with multiple applications, and the results are compared against a multicore platform. These experiments demonstrate the feasibility and the potential of this approach.**

***Index Terms*—Memristor, algorithmic skeleton, SystemC.**

I. Introduction

Big Data Analytics is becoming increasingly difficult to solve using CMOS-based Von Neumann computer architecture [1]. The reasons include, but are not limited to, the access bottleneck between the processor and memory, energy inefficiency [2], and the limited scalability of CMOS technology [3]. Memristor [4], [5]-based Computation-in-Memory (CIM) architectures [6]–[9] address the aforementioned problems by enabling in-memory processing using emerging non-volatile technologies. Manually designed case studies revealed their enormous potential by outperforming the state-of-the-art with orders of magnitude [10]–[12]. Exploring the potential of such architectures and appropriately evaluating their performance and scalability for larger applications require automatic flows and methods that efficiently map high-level algorithmic description to low-level memristor crossbar configuration.

Existing Computer-Aided Design (CAD) flows for CMOS-based VLSI (Very-Large-Scale Integration) are not applicable to memristor-based CIM because of different signal propagation styles. In CMOS circuits, logic values are represented by the voltage. The voltage change of a source automatically propagates to the sink along a dedicated wire within one clock cycle [13]. However, in a memristor crossbar, logic values can only propagate to other positions with the help of controllers, because they are encoded by memristors' resistance. The controller transfers the data in one or multiple steps, each of which is conducted along a vertical or horizontal nanowire shared by many memristors. Therefore, the number of steps equals to the number of turnings in the path between the source and the sink [14]. In addition to computation, memristor-based CIM needs extra clock cycles for communication, and the communication latency is determined by the routing result. In conventional VLSI CAD flows, placement and routing are performed based on the High-Level Synthesis (HLS) scheduling results [15]. However, it is not applicable for memristor-based CIM since the routing result is required to schedule communications. As a consequence, a new methodology is needed to eliminate the cyclic dependency among scheduling, placement, and routing.

In this work, we propose a synthesis flow that simultaneously performs scheduling, placement, and routing. This is inspired by the *skeleton* concept used in parallel computing domain [16]–[20]. A skeleton is a scheduling template for a specific class of algorithms that share a similar Data Flow Graph (DFG) in the sense of data dependency. A scheduling template handles parallelism, synchronization, and communication among threads, regardless of their functionality. It can be optimised according to the characteristic of DFG structures, thus achieving better performance than generic scheduling algorithms. FPGA developers extended this concept into a *hardware skeleton* with placement information [21], [22]. Routing is not included in hardware skeletons since it is generated by FPGA back-end tools. Nevertheless, the routing information is essential for mapping algorithms to memristor crossbar. Hence, we further extend the hardware skeleton concept with routing information and refer it as [1]*CIM skeleton* [23]. This skeleton can be configured with different predesigned circuits for implementing corresponding algorithms. Furthermore, complex algorithms can be implemented via skeleton nesting. This article is built on our preliminary work, where the main focus was laid on the general idea of applying skeletons to CIM architecture design. Compared to the preliminary work, we have made the following new contributions:

J. Yu, R. Nane, I. Ashraf, M.Taouil, S. Hamdioui, and K. Bertels are with the Department of Quantum Engineering, Delft University of Technology, 2628 CD Delft, The Netherlands (e-mail: j.yu-1, r.nane, i.ashraf, m.taouil, s.hamdioui, k.l.m.bertels@tudelft.nl).

H. Corporaal is with Department of Electrical Engineering, Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands (e-mail: h.corporaal@tue.nl).

Manuscript received December 19, 2016.

[1]*Skeletons* refer to CIM skeletons in the rest of the paper.

- We developed memristor-based computational units including a 32-bit adder and a 16-bit multiplier. The adder outperforms state-of-the-art memristor adder designs in terms of delay.
- We specified a methodology that allows us to integrate multiple computational units in the crossbar while maintaining parallel computing.
- We considered data input and output processes and identified possible patterns in the input/output data.
- We provided new test cases. All the skeletons are covered by the updated test cases.

The rest of the paper is organized as follows. First, Section II presents our memristor-based designs and systems. Subsequently, Section III presents the skeleton-based synthesis flow. Section IV explains implementation details of its key parts. Experimental results of three case studies are shown in Section V. Finally, Section VI concludes the paper and discusses future research directions.

II. Hardware Platform

Section II-A presents the primitive circuits that we designed for CIM, including a 32-bit adder and a 16-bit multiplier. Subsequently, Section II-B presents the hardware organization at the system level.

A. Primitive Circuits in CIM

In memristor-based CIM architectures, a *primitive circuit* or a *circuit* in short, is a memristor circuit that performs a computational operation in the crossbar, such as addition and multiplication. These circuits can be implemented in various manners. Some design styles may use shared controllers where the applied voltages are data-independent. Examples are material implication logic [24], Boolean logic [25], majority logic [26], and MAGIC (Memristor-Aided loGIC) [27]. Other designs cannot have a shared controller, such as CRS [28], where the control signals are data-dependent. In this work, we use MAGIC due to its simplicity. In principle, any of the above logic schemes that support a shared controller can be used.

CIM regards memristors as digital devices that have two stable states. In MAGIC, logic '1' is represented by low resistance (ON) and '0' by high resistance (OFF) [29]. The operations that switch a memristor to ON/OFF states are respectively called SET/RESET. They can be achieved by applying positive or negative voltages that are larger than the threshold voltages of the memristor [27].

In MAGIC, memristors are placed on a 2-dimensional grid, where each memristor is connected to a horizontal and a vertical nanowire [27] (see Fig. 1). Appropriate voltages are applied to nanowires by the CMOS controlled voltage drivers. The voltage drivers consist of a set of voltage sources and switches as shown in Fig. 2. The voltage sources have different voltage levels, and switches determine which supply voltages are selected. MAGIC uses up to nine voltage levels, including the three levels shown in the figure. Here, V_0 is the execution voltage, which works together with ground (GND) to execute a logic operation. V_{VS} is the isolation voltage and is applied

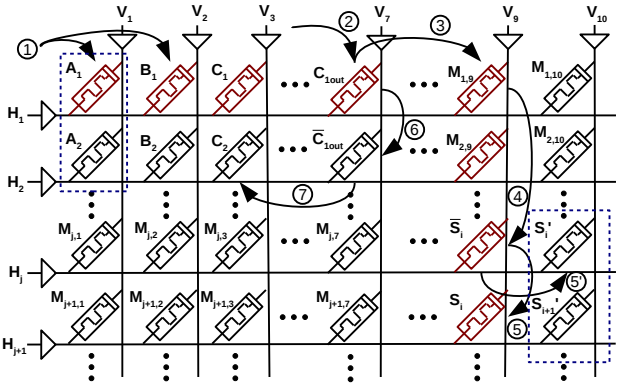


Fig. 1: 32-bit adder implemented with MAGIC logic.

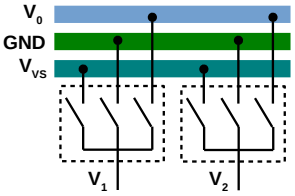
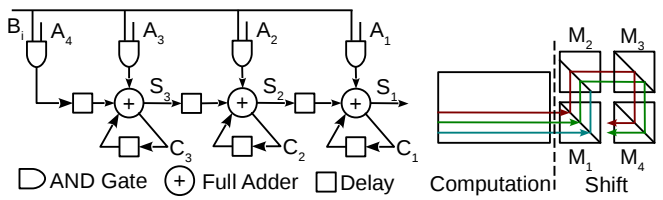


Fig. 2: Voltage controller implementation in CIM.

to the columns where memristor states are intended to not change. Examples of other voltages are SET, RESET and READ voltages. The number of voltage levels is much more than RRAMs, which typically require four voltage levels [30].

MAGIC supports only one logic operation in the crossbar, an n-input NOR operation [27]. The input and output values of the NOR operation are stored in memristors that share the same row or column. Note that when $n = 1$, the NOR operation functions as a NOT operation. As NOR is functionally complete, we can use it to build various circuits. Talati *et al.* presented a 1-bit full adder [27] using the red memristors shown in Fig. 1. First, the inputs are copied to the adder (step ①). Then, several NOR operations are conducted in the first row and the carry bit is obtained at the seventh column step ②). Subsequently, step ③ and step ④ are used as intermediate computation steps. Finally, the sum bit S is obtained from its complement $\overline{S}$ (step ⑤). We designed an n-bit adder based on this adder. The n-bit adder is also shown in Fig. 1. Two main changes have been made. The first one is that we inserted two steps (⑥ and ⑦) between step ② and ③ to move the carry output bit to the carry input bit of the next 1-bit adder. The second change is to conduct step ⑤ horizontally (as shown by ⑤') instead of vertically, to allow parallel operation. We have listed the detailed control steps in the supplementary material of this article. Talati *et al.* also presented an n-bit adder in which multiple 1-bit adders are linked together, with an overall latency as $10n + 3$ [27]. We have improved it to $8n + 8$ by leveraging the data parallelism as indicated by the dashed box in Fig. 1. Our design is also faster than the MAGIC-based design provided in [31].

We have also implemented a 16-bit multiplier, inspired by the Carry-Save Add-Shift (CSAS) algorithm [32]. Fig. 3a shows a 4-bit CSAS multiplier, which contains four AND gates and three 1-bit full adders. For more details regarding this



(a) 4-bit CSAS multiplier (b) CIM multiplier

Fig. 3: CIM multiplier design.

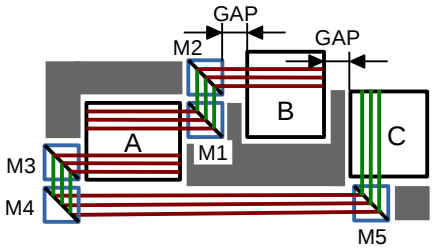
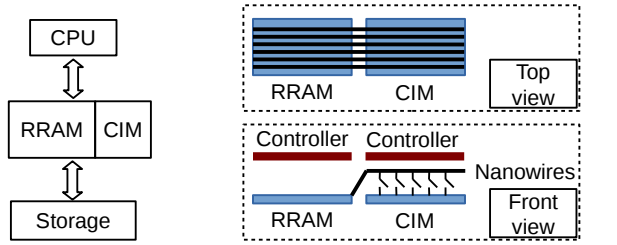


Fig. 4: The linkage of three primitive circuits.

TABLE I: Attributes of Primitive Circuits Used in the Article

Circuits	Latency (CC)	Width	Height	Energy (pJ)
Adder	264	10	96	265.19
Multiplier	499	81	64	3084.53
3-input XOR	32	12	32	169.28



(a) Overall structure (b) Communication between RRAM and CIM

Fig. 5: CIM/CPU heterogeneous computing.

algorithm, we refer the reader to Gnanasekaran's article [32]. CSAS algorithm suits a memristor implementation due to the parallel AND and add operations. However, the shift operation (required to shift the sum S_i) might be a bottleneck. When no dedicated hardware is provided for this shift operation, its time complexity equals $O(N)$, with N representing the amount of bits to shift [33]. We try to accelerate the shift operation in CIM using additional hardware as shown by Fig. 3b. The left part (donated by *Computation*) contains all AND gates, 1-bit full adders, and intermediate results. The right part (donated by *Shift*) contains four *mirrors* (M_1 to M_4); a mirror is a small square crossbar that only contains memristors on diagonal positions and are used to link horizontal and vertical nanowires [14]. Mirror M_2 is a special mirror in which its memristors are located on a line parallel to the diagonal. Therefore, this mirror shifts a signal by one position as shown by the red and green lines. More details, including control steps, can be found in the supplementary material.

The latency, area, and energy consumption of the primitive circuits are listed in Table I. The latency, expressed in clock cycles, is directly obtained from the number of cycles the controller needs (see also the supplementary material). The area, expressed in number of required memristors, is determined from the number of rows (Height) and columns (Width). The energy is calculated from the number of operations per cycle and the data width. We assume one memristor to be written for each bit during each operation. Note that the energy consumption is ideally input-depended. The cost to write a memristor (SET/RESET) lies in the range 0.1 fJ [34] to 230 fJ [35]. We assume the worst case of 230 fJ. The static power consumption is ignored here and will be part of the future work.

B. Circuits Organization in CIM

When we assemble the primitive circuits into a large system, we have two main targets. First, these circuits must operate in parallel to achieve better performance, and second, they should not conflict with each other during operation. Therefore, we avoid placing input/output ports on the same rows or columns and link them by using two or more mirrors. Fig. 4 shows for example three primitive circuits A, B, and C. A exchanges data with B and C. The horizontal and vertical lines are colored red and green respectively. Nanowires that are irrelevant to the communication among A, B, and C are omitted. In CIM, we assume that all data words are 32-bit wide, and that all the bits are transferred in parallel. When A transfers a word to B, it is first copied to $M1$, then to $M2$, and finally to B. The nanowires between $M1$ and B are removed, and barriers are inserted in the gap to eliminate or reduce parasitic capacitance. The barriers are shown as gray blocks. This allows A and B to operate independently since they do not share nanowires or memristors. Communication is conducted when A and B are not operating. The negative affect of this solution is that it slightly decreases the density of the crossbar and, more importantly, increases the manufacturing complexity.

The latency of the communication described above equals to the number of mirrors plus one. Therefore, it needs one more cycle for A to transfer data to C than to B in Fig. 4. This feature has a significant influence on the design automation, which will be analyzed in Section III-A.

Fig. 5 shows one of the CIM's working scenarios [9], [23], where Resistive Random Access Memory (RRAM) serves as the main memory. It exchanges data with CPU and storage in the same way as conventional technologies. Since the memristor crossbar is not continuous, we need to add additional nanowires to transfer main inputs and outputs for internal circuits. CIM works as an accelerator and is placed besides RRAM. RRAM and CIM both have a memristor layer, and a CMOS layer. The controller for both RRAM and CIM is implemented in CMOS. CIM is connected to the RRAM via nanowires in a dedicated layer as shown in the part of Fig. 5b. Each nanowire creates a connection between RRAM and one or more primitive circuits as shown in the front view. A nanowire transfers at most a single data bit during each clock cycle.

The controller's area puts another constraint on the system

design. We have synthesized the CMOS controller for the 32-bit adder with Cadence's RTL Compiler and NanGate's $15\,\mathrm{nm}$ library [36]; the reported area is $30\,\mu\mathrm{m}^2$. However, International Technology Roadmap for Semiconductors 2.0 (ITRS 2.0) predicts that the density of memristor crossbar will reach $2.38 \times 10^{11}\,\mathrm{bit/cm}^2$ in 2020 [37]. With this density, a crossbar of 96 rows and 10 columns (i.e. the size of the adder) is only $0.23\,\mu\mathrm{m}^2$. This means that the CMOS controller is 130x larger than the memristor crossbar. If we assign a dedicated a controller to every primitive circuit, then CMOS controller's area dominates the chip area and we cannot exploit the high density of the crossbar. Therefore, the controller must be shared between the logic circuits. Furthermore, this is possible due to the nature of memristor logic as discussed in Section II-A. Sharing the same controller requires the circuits to operate synchronously. In Section V, we will show that the controller can handle 10^4 to 10^5 circuits simultaneously. As a result, the crossbar area becomes dominant. Due to the small area of the controller, it will be ignored in the rest of this paper.

RTL Compiler reports the power of the 32-bit adder to be $13.59\,\mu\mathrm{W}$ with a frequency of $1\,\mathrm{GHz}$. It leads to an energy consumption of $3.588\,\mathrm{pJ}$ for a time period of $264\,\mathrm{ns}$ as the latency of the adder is 264 cycles. It is less than 2% compared to the energy consumption in the memristor crossbar. When the controller is shared with many primitive circuits, the percentage will be even smaller. Therefore, we will also omit the energy consumption of CMOS layer in the rest of the article.

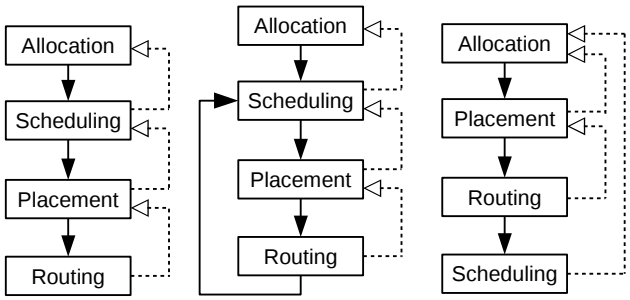
III. Skeleton-based Synthesis Flow

Section III-A motivates the reason why a radically new design flow is required. Subsequently, we introduce the skeleton-based synthesis flow in Section III-B, III-C, and III-D.

A. Requirement for a New Flow

The communication characteristic of memristor crossbar makes scheduling depend on routing results because communication latency is decided by routing. Fig. 6a shows the HLS flow used for CMOS circuit design. It consists of sequential processes, mainly are resource allocation, scheduling, placement, and routing. Only when a process fails meeting the performance or resource constraints, it goes back to the previous process. It is worth noting that scheduling is conducted before routing; hence, it is not applicable to CIM.

We can try to adapt the regular HLS flow to CIM, but these variants all lead to unsatisfactory situations. Since routing information is not available at scheduling phase, we can assume all communication has a maximum latency, like six or eight cycles. Based on this assumption, the operators can be scheduled. Then, after the routing phase, the communication latency is updated. Scheduling is conducted again to get a more accurate design. This adapted flow is shown in Fig. 6b. Placement and routing are time-consuming processes, so these iterations are extremely time consuming. If we make a trade-off between the quality of the solution and the execution time, then the performance of the generated design will be only suboptimal.



(a) HLS flow for CMOS (b) Flow with communication latency assumption (c) Flow with adjusted sequence

Fig. 6: Regular HLS flow and its variants for CIM.

Fig. 6c shows an alternative variant, i.e. conducting placement and routing before scheduling. In this scenario, the scheduler has accurate information on communication latency. The drawback of this flow is that we need to go through all the processes before knowing whether the latency constraint is met. Similarly to Fig. 6b, the long execution time of placement and routing will impair either the productivity or the quality of the design.

The fundamental problem of these adaptions is that they cannot eliminate the cyclic dependence among scheduling, placement, and routing. In a regular HLS flow, placement and routing should be conducted based on the scheduling result. However, in CIM architecture, the communication mechanism makes scheduling depend on the routing result. Therefore, a radically new approach is required. Different from these adapted flows, we solve the scheduling, placement, and routing altogether using CIM skeletons. The optimal solution is guaranteed without the need of iteration. This methodology is introduced in the next section.

B. Hardware/Software Partitioning

Figure 7 shows the overview of the complete CIM synthesis flow, which consists of four components. At the application level (Box 1), the user partitions the original program into software and hardware, taking the hint given by the profiling tool. The hardware part needs to be rewritten to fit predefined skeletons. A skeleton contains scheduling, placement, and routing algorithms for a specific type of DFG structure (Box 2). The compilation at the kernel level (Box 3) is to instantiate skeletons with *Primitive Circuits*, which are predefined function units like adders and multipliers. The design of the circuit level (Box 4) has been presented in Section II-A In the flowing subsections, we will elaborate each of the rest boxes.

Before the compilation at the kernel level, we need to identify the favorable algorithms for hardware implementation. The best candidates should meet the following criteria:

- They form a large percentage of the execution time. According to Amdahl's law [38], accelerating such kernels can generate a recognizable overall speedup.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2760927, IEEE Transactions on Emerging Topics in Computing

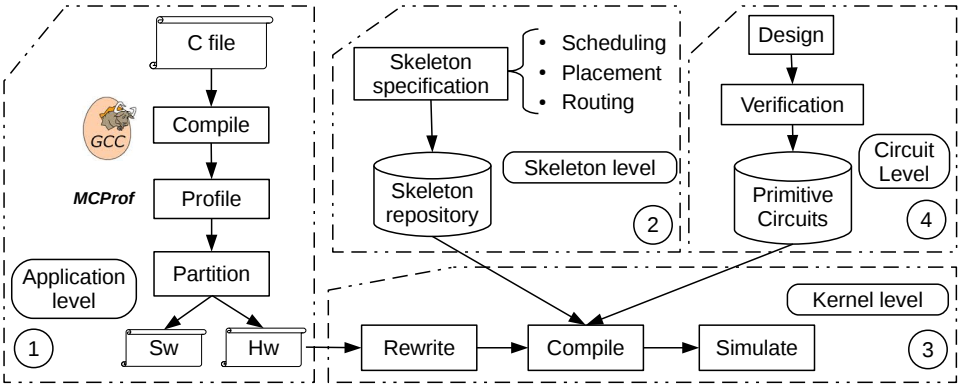


Fig. 7: Synthesis flow for memristor-based CIM architecture.

```
1  #define InputSize 8192
2  #define TAPS 64
3  int pre[TAPS], coe[TAPS], out;
4  void FIR(int in){
5    int temp = 0;
6    for(int j=TAPS-1; j>0; j-=1)
7      pre[j] = pre[j-1];
8    pre[0] = in;
9    for(int j=0; j<TAPS; j++)
10     temp+=pre[TAPS-j-1]*coe[j];
11     out = temp;
12 }
13 void main(){
14   int i, total = 0;
15   for(i=1; i<=InputSize; i++){
16     FIR(i); total += out;
17   }
18   printf("total:%d\n", total);
19 }
```

Fig. 8: FIR filter source codes.

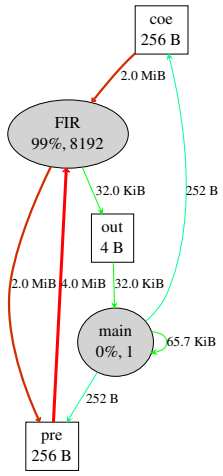


Fig. 9: MCProf profiling result.

- They are coarse-grained, which means they do not change a large quantity of data with other parts of the application.
- They have inherent massive parallelism so that they have the potential to be accelerated.
- Their structures are easy to be implemented by hardware.

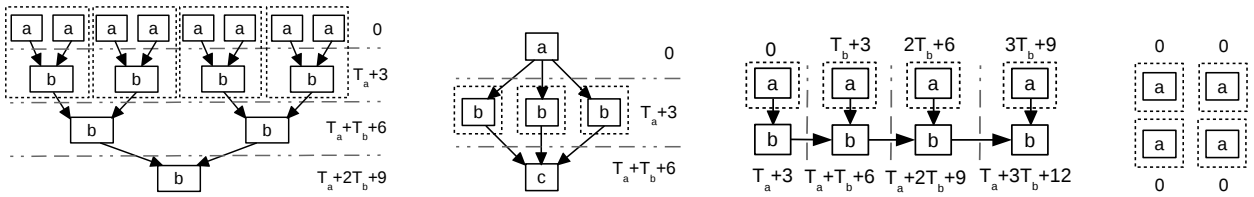
In order to highlight the computing and memory intensity parts of an application and to obtain the communication among these parts, we utilize *MCProf* [39], [40]. MCProf is a runtime memory and communication profiler based on Intel *Pin* dynamic binary instrumentation framework [41]. MCProf takes the binary of an application as input to generate profiling results in various formats. Based on the information generated by MCProf, developers can partition the application into software and hardware parts, as shown in Box 1 in Fig. 7. Later, the hardware part enters the kernel-level design flow which will be explained in Section III-C.

To illustrate the utilization of MCProf to extract the required information from an application, let us consider the C program of a Finite Impulse Response (FIR) filter modified based on LegUp's [42] testbench as shown in Fig. 8. The initialization of the coefficient array *coe* is omitted for concision. MCProf generates the output shown in Fig. 9. The functions are represented by ovals, which contains the name of the function, its percentage execution contribution, and the total number of calls; e.g. *FIR* function consumes 99% of the overall execution time and is called 8192 times. The rectangles represent objects, such as the *pre* and *coe* arrays in this case. The arrows represent the communication with the data amounts marked near the lines. Dense communication is indicated by red color (bold lines), and the rest is green. Clearly, the *FIR* function consumes most of the execution time, and most of the communication is between it and arrays *pre* and *coe*. If we implement the *FIR* function in main memory using the CIM concept, then the data transfer between the processor and the memory will be several orders of magnitudes smaller than the original version. In this example, the profiling is performed at the function level. By using markers, it is also possible to obtain profiling information at lower granularity levels, such as the loop level.

C. CIM Skeletons

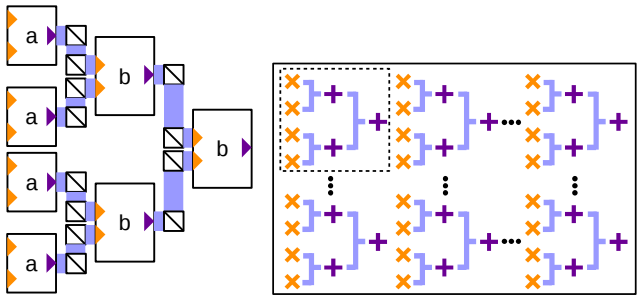
In this skeleton-based synthesis flow, targeting problems are mapped to the crossbar by instantiating predefined solution templates with primitive circuits. Each skeleton can map a specific class of problems that share a similar DFG structure. In this paper, we generally follow the classification defined by Campbell [43] and define four structures as shown in Fig. 10. We chose this classification because it contains a relatively small number of classes while covering a broad range of problems. Each box in Fig. 10 represents an operation or a DFG consisting of multiple operations, and boxes with the same labels represent the same operation(s). The arrows between them indicate data dependency. The four structures are:

- **Recursively partitioned**. Problems are partitioned into a small size, and they are solved separately. After that, the solutions are collected in a recursive style.
- **Farm**. The same function is applied potentially in parallel to a list of independent jobs. The results are combined by a controlling process.
- **Systolic**. It consists of processes that have data flowing between them, and that may operate concurrently in a pipelined fashion.



(a) Recursively Partitioned (b) Farm (c) Systolic (d) Crowd

Fig. 10: DFGs, scheduling, and parallel simulation support of fundamental skeletons.



(a) Binary-tree layout pattern (b) 4×4 matrix multiply

Fig. 11: Binary tree layout pattern and its usage in matrix multiplication.

- **Crowd**. Similar to the Systolic skeleton except for that there is no data flow between the concurrently operating processes.

Note that we are not using Campbell's *Task queue* skeleton, which is a generalized version of *Farm* as it is not suitable for hardware implementation. It is also worth noting that the sizes of these structures are unfixed. For example, a *Recursively Partitioned* skeleton also suits problems with more layers as long as the solutions are collected recursively. Other classification, e.g. the one used in STAPL framework [17], can also be adopted in the synthesis flow.

For each problem class, we specify the scheduling, placement, and routing algorithms, and store them in a repository as shown in Box 2 of Fig. 7. In the placement aspect, primitive circuits and the hardware design they constituted are represented by their bounding rectangles. These rectangles are not allowed to overlap each other. We take *Recursively Partitioned* skeleton as an example of the solution templates. The placement algorithm specified in this skeleton places boxes *a* and *b* following a binary-tree patternas shown in Fig. 11a. All the intermediate data are transferred via two mirrors, which are minimum number required (see Section II-B). Since the communication cost is known as three cycles, the problem can be scheduled as the expressions shown in Fig. 10a. The expressions are the starting moments of corresponding operations, in which T_x represents the latency of box x, e.g., T_a means a's latency. The dash-dot lines divide the DFG into several regions. Boxes in each of them execute in parallel. For other skeletons shown in Fig. 10, the scheduling results are also marked in a similar way.

The skeleton can break the cyclic dependence of scheduling, placement, and routing that we discussed in Section III-A. The reason is that these algorithms are defined altogether instead of separately. Limiting the problems' DFG structures facilities the development of these algorithms. For instance, the binary-tree placement algorithm improves the performance for the *Recursively Partitioned* skeleton, but it cannot be applied to other problems.

D. Skeleton Instantiation

A skeleton generates a hardware design after instantiated with primitive circuits or other hardware designs. In the latter case, we can solve complex problems that do not fit any fundamental skeleton. One advantage of the skeleton-based flow is that the users do not need to take care of implementation details. Instead, they just need to analysis the DFG and apply the right skeleton.

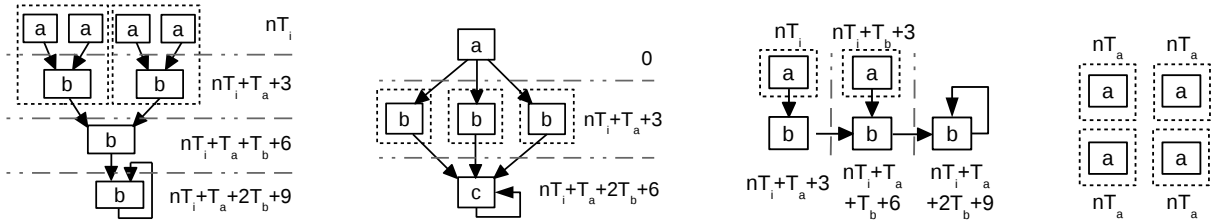
Suppose we intent to map the matrix multiply algorithm on CIM:

$$AB = \begin{pmatrix} \vec{a_1}^\intercal & \vec{a_2}^\intercal & \cdots & \vec{a_n}^\intercal \end{pmatrix}^\intercal \begin{pmatrix} \vec{b_1} & \vec{b_2} & \cdots & \vec{b_n} \end{pmatrix} = \begin{pmatrix} \vec{a_1} \cdot \vec{b_1} & \vec{a_1} \cdot \vec{b_2} & \cdots & \vec{a_1} \cdot \vec{b_n} \\ \vec{a_2} \cdot \vec{b_1} & \vec{a_2} \cdot \vec{b_2} & \cdots & \vec{a_2} \cdot \vec{b_n} \\ \vdots & \vdots & \ddots & \vdots \\ \vec{a_n} \cdot \vec{b_1} & \vec{a_n} \cdot \vec{b_2} & \cdots & \vec{a_n} \cdot \vec{b_n} \end{pmatrix}, \quad (1)$$

where $\vec{a_i}$ is a row vector of matrix A, and $\vec{b_i}$ is a column vector of B. It is a complex algorithm that does not fit any skeleton. However, we can see that it contains repetitive patterns. Each element of the result matrix is an inner product of two vectors. Thus, we can divide it into two levels. The top level is a *Crowd* skeleton because there are no data flows between these elements. The lower level is the vector inner product function. This function suits a *Recursively partitioned* skeleton, with "a" and "b" boxes in Figure 10 replaced as multipliers and adders.

To implement the matrix multiply, we need to build the system bottom-up. First, we instantiate a *Recursively partitioned* skeleton with the multiplier and the adder. After that, we instantiate the *Crowd* skeleton with the inner product just generated. We assume both matrices are 4×4, so the vector size of the inner product is also 4. Figure 11b represents the generated system. The symbols "$\times$" and "$+$" stand for multipliers and adders while dashes between them are communication paths. Each subsystem, as shown in the dashed box, has a detailed layout following the binary-tree pattern. If an application cannot fit any existing skeleton, it is necessary to develop a new one. In this case, the skeleton repository should be extended.

In a similar way, we can implement the *FIR* function shown in Fig. 8. The *for* loop at line 9 and 10 suits the



(a) Recursively Partitioned (b) Farm (c) Systolic (d) Crowd

Fig. 12: DFGs, scheduling, and parallel simulation support of fundamental skeletons with hardware reuse.

Systolic skeleton, where "a" and "b" boxes are instantiated with multipliers and adders. To further accelerate the program, we instantiate the *Crowd* skeleton with the generated *FIR* kernel to enable the high-level parallelism represented by the *for* loop in the main function (line 15 and 16).

IV. Constraints and Optimizations

Next, we introduce more implementation details of CIM synthesis flow. They improve the flow's ability to support large designs. First, we present the methods used for meeting area constraints in Section IV-A. Then in Section IV-B, we analyze the patterns existing in data transfer and use them to decrease the bandwidth and optimize the design. Thereafter, the tool's feature of supporting parallel SystemC simulation to deal with large designs is covered in Section IV-C.

A. Area Constraint and Hardware Reuse

Our skeleton-based flow supports user-defined area constraint, which represents the chip size or the area reserved for a hardware design. When the design area exceeds the constraint, we need to allocate less hardware and reuse it. We first adjust the DFGs to preserve the functionality. Then, the scheduling, placement, and routing algorithms are modified accordingly.

The modified DFGs are shown in Fig. 12. Boxes in these DFGs execute n times instead of just once in Fig. 10. Loop-backs are introduced to accumulate the result generated in different iterations. Comparing Fig. 10a and Fig. 12a as examples, we can find the box b at the lowest level both accepts two inputs. In the former DFG, these two inputs come from two sub-DFGs at higher levels simultaneously. In the latter one, they are from the same sub-DFG sequentially. The result would be the same as long as b is correctly initialized. For instance, if b is an adder, its initial output should be set to zero.

The mapping and routing algorithms for these skeletons are similar to the *flattened* designs, i.e. the skeletons without hardware reuse. A *demultiplexer*, or a *demux* in short, is introduced into each box that has a loop-back routing. The demux can route the output signal to loop-back during computing, and send it to the output port of the whole design at the final stage. The scheduling results are also indicated in Fig. 12; T_i in these expressions means the largest latency among all the boxes. It is usually called *initiation interval*, which is the number of cycles that must elapse between two sets of inputs.

Fig. 13 shows the procedure of constructing designs with hardware reuse. First, we build a flattened design without

```
   Data: Area constraint A_c, problem size S_p
   Result: A valid hardware design
1  Calculate flattened design's area A_f;
2  if A_f < A_c then return The flattened design ;
3  N_u ← S_p, N_l ← 1 ;      // Upper & lower bounds
4  while N_l + 1 < N_u do      // Search space > 1
5  |  N_c ← (N_u + N_l)/2;      // Set reuse times
6  |  Build a design for problem size Ceil(S_p/N_c);
7  |  Calculate current design's area A_r;
8  |  if A_r < A_c then      // Shrink search space
9  |  |  N_u ← N_c;
10 |  else
11 |  |  N_l ← N_c;
12 |  end
13 end
14 return The design for problem size Ceil(S_p/N_u);
```

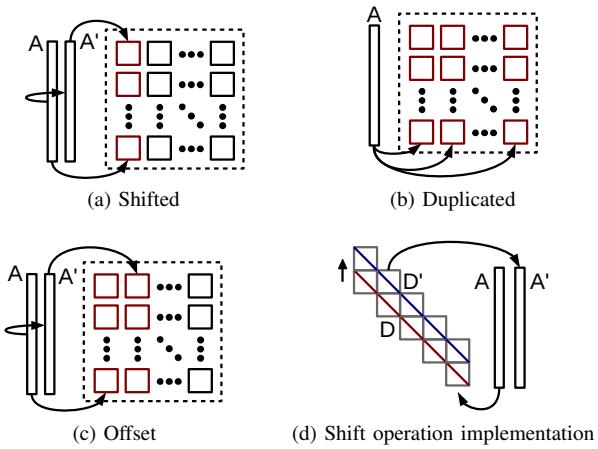
Fig. 13: Build designs under area constraint.

hardware reuse (line 1). If the area meets the constraint, this design will be returned immediately (line 2). Otherwise, hardware reuse is required. In this case, we use binary search to decide how many times the hardware needs to be reused (line 3 to line 13). The initial search space is between one and the problem size S_p (line 3), and the exit condition is that the search space has shrunk to one (line 4). When the hardware is reused for N_c times, each time the hardware only needs to process S_p/N_c inputs. We build a new hardware design for this problem size (line 6) and calculate its area (line 7). Then we update the upper or lower bounds depending on whether the area meets the constraint (line 8 to line 12). The final design is for problem size S_p/N_u, which has lowest latency and meets the area constraint.

B. Data Transfer and Bandwidth Constraint

After building the hardware for computation following previous sections, we need to consider their input/output data transfer, which also has an important impact on the overall performance. This section focuses on the communication between RRAM and CIM (See Fig. 5). In this paper, we assume the data has been stored in the RRAM. The communication between RRAM and other components, such as CPU and storage, is beyond the scope.

Before presenting our data transfer solution, let us examine the patterns in the input/output data. In the *FIR* function shown



(a) Shifted (b) Duplicated

(c) Offset (d) Shift operation implementation

Fig. 14: Data transfer for spatial and temporal patterns and shift operation.

in Fig. 8, line 9 and 10 specify the computation while line 6 to 8 describe the input data transfer. The computation part has two input arrays named *pre* and *coe*. In each execution, *pre* is shifted from the previous iteration (line 6 to 8). Similar patterns are common in other programs. By leveraging these patterns, the data can be transferred more efficiently. These patterns can be either temporal or spatial. The temporal patterns we recognized include:

- **Constant**. The data does not change in all/some iterations, like the *coe* array in *FIR* function. (The initialization of *coe* array is omitted as shown in Fig. 8.)
- **Shifted**. The data should be shifted before it is applied to different iterations. This is the case of *pre* array in *FIR* function.
- **New**. No aforementioned temporal relations among the data.

The spatial patterns are:

- **Duplicated**. The same data is used in different parts of the design. E.g., $\vec{a_1}$ is the input array for all the inner product in the first row of the matrix multiply as shown in Formula 1.
- **Offset**. The original data and its shifted versions are applied to different parts of the design. In Section III-D, we introduced that the *FIR* function can be implemented with the *Crowd* and the *Systolic* skeletons. It means duplicated hardware work in parallel. In this case, the *pre* array has an *Offset* pattern.
- **Irrelevant**. No aforementioned spatial relations among the data.

Next, we will show the way to deal with the above patterns in case of CIM design. Fig. 14 shows the data transfer procedures for different patterns. The dotted box represents CIM, and the boxes inside it are logic units. The rectangle on the left symbolizes the input data arrays stored in RRAM. We will not show the solution for the *Constant* pattern since the data does not change. Other solutions are listed below.

- **Shifted**. As shown in Fig. 14a, first the original data A is transferred to CIM. Then, it is shifted in RRAM. After one iteration of execution, the new data A' is transferred to CIM for the next iteration.
- **New** and **Irrelevant**. Data is transferred from RRAM to CIM column by column.
- **Duplicated**. Data is simultaneously transferred to multiple columns as shown by Fig. 14b, following the broadcast method proposed by Xie [14]. It is faster and more energy efficient compared with column-by-column data transfer.
- **Offset**. Similar to the solution for the *Shifted* pattern except that the shifted data A' is now transferred to other parts of the design within the same iteration as A, as illustrated by Fig. 14c.

The solutions for *Offset* and *Shifted* patterns both require the shift operation. This is conducted by using two groups of mirrors following the steps shown in Fig. 14d. First, the data A is copied to mirrors D. Then, all the bits are shifted to mirrors D' in parallel. Finally, the data is copied back as A', which is the shifted version of A.

Data: Input matrices dimensions: m, n, and k
Result: Hardware design of matrix multiply $A_{m\times n} \times B_{n\times k}$

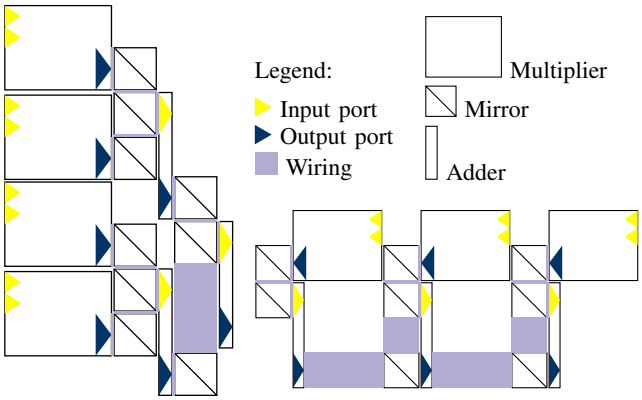
```
1 SetAreaCon(1e5, 1e5);
2 Multiplier mul(a, b);
3 Adder add;
4 Recur_ske inner⟨mul, add, n⟩(NONE, NONE);
5 Crowd_ske row⟨inner, m, HORZ⟩(DUPL, NONE);
6 Crowd_ske mm⟨row, k, VERT⟩(NONE, DUPL);
7 return mm.GenSystem();
```

Fig. 15: Pseudo codes of specifying matrix multiply in CIM compiling flow.

We use matrix multiply as an example to show the usage of data patterns. Fig. 15 specifies the matrix multiply with three skeletons. First, we set the area constraint (line 1), which represents a crossbar with $10^5 \times 10^5$ memristors. Then we define primitive circuits including the multiplier (line 2) and the adder (line 3). After that, three skeletons are instantiated: one *Recursively Partitioned* skeleton and two *Crowd* skeletons (line 4 to line 6). This instantiation is based on primitive circuits (such as *mul* and *add*), matrix parameters (such as m and n), as well as other skeletons; e.g., the instantiation of *row* makes use of *inner* (line 5), which is a *Recursively Partitioned* skeleton. Note that *Crowd* skeleton make use of two constants, HORZ and VERT, to specify the direction of duplicating circuits. HORZ in line 5 indicates *inner* is duplicated and placed in a horizontal direction (i.e., forming a row of *inner*). On the other hand, VERT in line 6 indicates that the former row of *inner* is duplicated and placed vertically, resulting in a matrix of *inner*. The parameters in parenthesises indicate the data patterns. Matrix multiply has two *duplicated* (DUPL) patterns for rows and columns.

The communication bandwidth between RRAM and CIM is also a significant constraint on the hardware design. The product of the bandwidth and initiation interval indicates the maximum data amount that can be transferred between two

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2760927, IEEE Transactions on Emerging Topics in Computing



(a) Inner product (b) FIR filter

Fig. 16: Generated graphic output of study cases.

pipelining stages. If the computation kernel expects more data, it will stall. We can limit the size of hardware design to avoid such stall, so that the same performance can be achieved with a smaller area. The *Duplicated* pattern reliefs the bandwidth constraint because it requires fewer data transfers from RRAM to CIM. If no bandwidth constraint is specified by the user, the theoretical maximum bandwidth is used. The theoretical bandwidth is N bits per Clock Cycle (CC), where N denotes the number of nanowires across the interface.

C. Parallel Simulation Support

Our compiler integrates parallel SystemC simulation support into skeletons' specification for acceleration and enabling large simulation scale. At the current development phase, the compiler generates SystemC files for behavior verification. However, the standard SystemC implementation [44] does not support parallelism, which limits the performance and scale of the simulation. Therefore, we replace some channels with Message Passing Interface (MPI)-based communication. Subsequently, we can distribute the simulation to multiple machines.

Fig. 10 and Fig. 12 illustrate our parallel simulation support for each skeleton. The parts surrounded with dotted boxes are simulated in parallel by different threads (possibly on different machines). The number and sizes of these boxes are decided by the number of available threads, which is set by the user.

V. Experimental Results

We conducted three sets of experiments to validate the design flow. Section V-A uses inner product and matrix multiply as case studies to show the source codes and graphic outputs. After that, we analyze the scalability of the flow in Section V-B while considering FIR filter. Parallel SystemC simulation results will be presented in Section V-C. Finally, we discuss the strength and limitations of the proposed synthesis flow in Section V-D.

A. Case Studies

We use the inner product of vector size four (see Fig. 16a) and FIR filter with tap size three (see Fig. 16b) as two

```
1  void encrypt (unsigned* v, unsigned* k) {
2    unsigned v0=v[0], v1=v[1], sum=0, i;
3    unsigned delta=0x9e3779b9;//key schedule const
4    for (i=0; i < 32; i++) {
5      sum += delta;
6      v0 += ((v1<<4)+k[0])^(v1+sum)^((v1>>5)+k[1]);
7      v1 += ((v0<<4)+k[2])^(v0+sum)^((v0>>5)+k[3]);
8    }
9    v[0]=v0; v[1]=v1;
10 }
```

Fig. 17: Tiny Encryption Algorithm source codes.

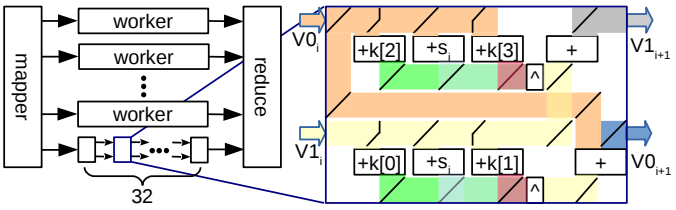


Fig. 18: Tiny Encryption Algorithm's hardware implementation with the *Farm* and *Systolic* skeletons.

examples to show the generated graphic layout of the skeleton-based synthesis flow. The large and small rectangles represent multipliers and adders, respectively. Within them, the light yellow and dark blue triangles denote the input and output ports, and the light blue fields represent the area dedicated for wiring. The figures clearly show the usage of the binary tree and systolic patterns in these figures.

Next, we use a more complex case study, i.e. Tiny Encryption Algorithm (TEA), to show how the skeleton-based design methods can be used to implement real-life applications. TEA is a simple block cipher that uses a 128-bit key to encrypt 64-bit data blocks [45]. Fig. 17 shows its C implementation. The function accesses the plaintext and the key with pointers (line 1), and the ciphertext is also returned via a pointer (line 9). A 32-bit constant ($0x9e3379b9$, line 3) is used to prevent simple attacks based on the symmetry of the rounds. The encryption process consists mainly of a loop of 32 iterations (line 4 to 8). Each iteration contains shift, addition, and XOR operations (line 6 and 7).

We manually designed a hardware unit as shown in the right part of Fig. 18 to implement one iteration of TEA. This unit has eight adders and two 3-input XOR operators, represented by rectangles with "+" and "^" symbols, respectively. As shown in the source code (line 6 and 7 in Fig. 17), most adders have one constant input. These constants are also provided in Fig. 18. "S_i" means variable sum's value in the ith iteration, which is available during compilation. The mirrors are represented by black slashes. They link the horizontal and vertical nanowires, which are illustrated by colored stripes. Different colors are used to indicate different data. Polylines ("⟋" and "⟍") represent special mirrors whose memristors are located in a line parallel to the diagonal. These special mirrors are used to implement shift operations. Next, this unit is duplicated using the *Systolic* skeleton as shown in the bottom left part of Fig. 18. The resulting circuit (*worker*) implements all the 32 iterations, thus representing an entire

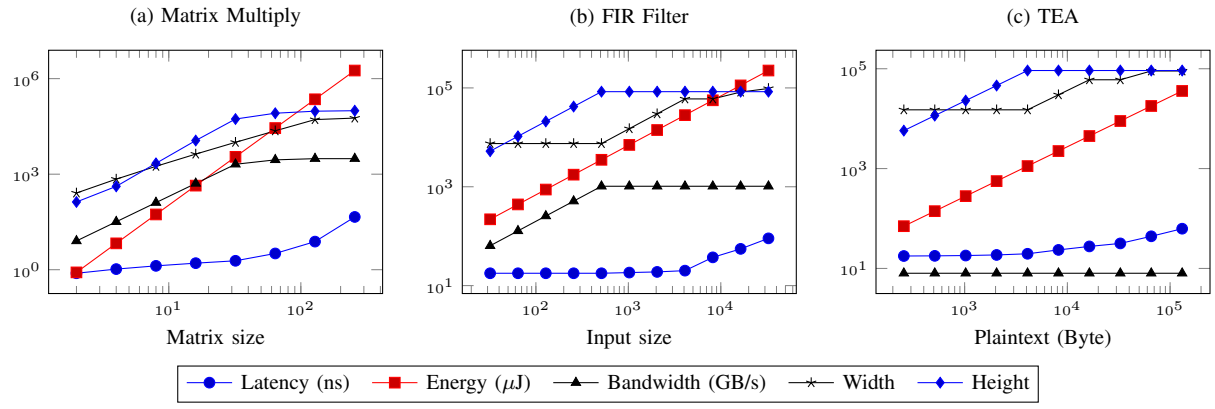


Fig. 19: Latency, energy, bandwidth, and area of scaling applications in CIM.

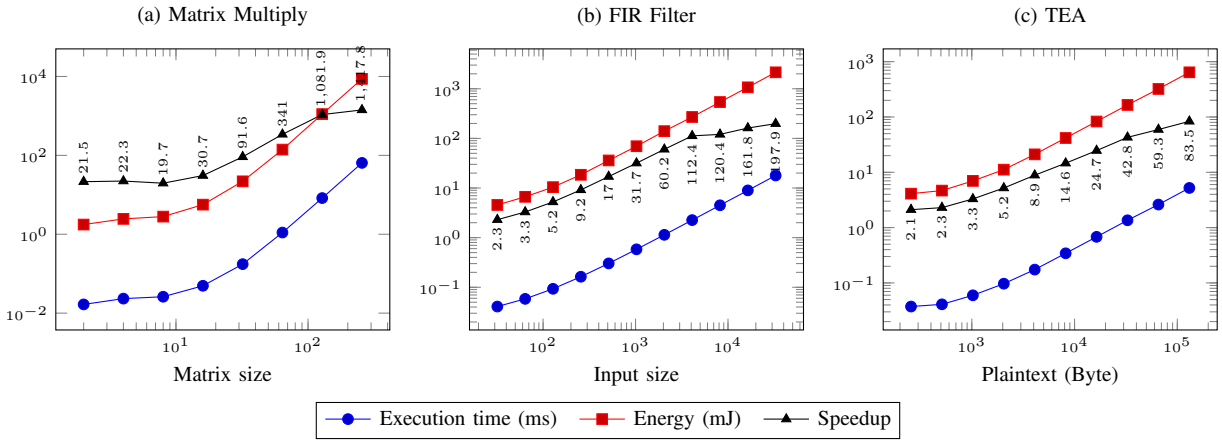


Fig. 20: Latency and energy of scaling applications on the multicore platform.

TEA function. This circuit is further duplicated by the *Farm* skeleton to speedup encrypting different parts of the plain text in parallel. The *Farm* skeleton uses two helper circuits (*mapper* and *reduce*) to split the plaintext and merge the ciphertext as shown in the left part of Fig. 18.

B. System Scaling

In this section, we compare CIM's performance against a multicore system to show the quality of our compiler's generation. The targeted multicore system is Intel Xeon X7460 processor that consists of six cores on a die of $503\,\mathrm{mm}^2$, running at $2.66\,\mathrm{GHz}$ each [46]. We assume the CIM chip to be only 10% of the area of Xeon X7460, and only 10% of the CIM chip is used for computation (the rest is used as RRAM, see Fig. 5). Then, the computation part contains about 10^{10} memristors according to the density predicted by ITRS [37]. Therefore, we add an area constraint $10^5 \times 10^5$ to the synthesis flow.

We varied the problem sizes to evaluate the scaling capabilities with area constraint and generated three cases: matrix multiply, FIR filter, and TEA. We assume the matrices to be square $n \times n$. In the FIR application, the taps number is fixed as 64, and input size changes; see Fig. 8. The problem size of TEA can be valued by the plaintext size. We assume CIM's clock frequency is $1\,\mathrm{GHz}$, considering the memristor switching time is in the range of a hundred picoseconds [47]. The performance and the cost of generated designs are shown in Fig. 19. In all three cases, the latency increases faster when the area limit is reached. This indicates that the hardware is reused to meet the area constraint. Whether the hardware is reused or not, the energy consumption increases almost at the same rate as it is determined by the total number of operations. For matrix multiply and FIR, there is a positive correlation between the bandwidth and the crossbar height, since the data in different rows can be transferred in parallel (see Section II-B). On the other hand, TEA's bandwidth remains constant, because it uses a *mapper* circuit to split sequential inputs to the *worker* threads (see Fig. 18). In all three cases, the width and the height do not increase when they approach 10^5, due to the area constraint we set.

To show the quality of the synthesized designs, we evaluated the execution time and energy consumption of these applications on a multicore platform and compared the execution time against CIM. This evaluation is conducted with Sniper [48], and the energy consumption is reported by McPAT [49]. We employed a simulator instead of using real hardware because it benefits the reproducibility. The targeted hardware platform is an Intel Xeon X7460 processor, which consists of six cores, each running at $2.66\,\mathrm{GHz}$. These cores have $64\,\mathrm{kB}$ L1 cache each and share a $16\,\mathrm{MB}$ L3 cache. Every two cores share an

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2017.2760927, IEEE Transactions on Emerging Topics in Computing

TABLE II: Comparison Between Design Methodologies

	Application		Design quality		Development	
Design methodologies	Type	Size	Latency	Area	Efforts	Execution time
Manual	Regular	Large	Low	Large	Much	-
Skeleton-based (this work)	Half-regular	Large	Low	Large	Medium	Short
Fully automated	Irregular	Medium	Slightly high	Small	Little	Long

TABLE III: Baselines of Parallel Simulation

Applications	Size	Base (s)
Matrix multiply	64	1686
FIR filter	512	1036
TEA	4096	1859

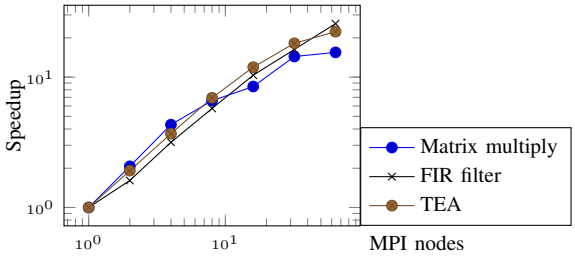


Fig. 21: Parallel simulation speedup.

L2 cache of 3 MB. The experimental results, including the speedup of CIM over the multicore platform, are shown in Fig. 20. The values of the speedup are marked beside the line. The maximum speedup for matrix multiply, FIR filter, and TEA is 1418x, 197.9x, and 83.5x, respectively. The energy consumption of multicore is about one order of magnitude larger than CIM for all the three cases.

C. Parallel Simulation

We enabled the parallel simulation support to examine its effect. These experiments are performed on a high-performance computer with 20 Intel Xeon E5-2670 HT cores, running at 2.50 GHz each. First, we simulated the baselines which are based on sequential simulations. Table III shows the sizes of simulated applications and the corresponding simulation time. After that, we fixed the system size and changed the number of MPI nodes and generated eight configurations. For each of these configurations, we performed the simulation ten times and calculated the average execution time after removing the maximum and minimum values. Figure 21 shows the speedup of each configuration over the sequential simulation as the baseline. The output data of all the parallel simulations are verified and found to match those of the sequential one. When MPI nodes are less than 16, the speedups are almost the same as the thread number. When the nodes number increases beyond 16, the speedup tends to saturation. It is understandable since the cores in hardware are limited. This result shows a good scalability.

D. Discussion

We compared the skeleton-based design flow, the manual design flow, and a potential fully automated flow in Table II to identify their characteristics. The fully automated flow is similar to existing VLSI design flows that can map any application to the hardware without using predefined solutions. Such a flow is currently not available due to design constraints of memristor-based CIM architectures that have been discussed in Section II-B and III-A. In addition, existing research on manual designs, such as [10], [50], have different assumptions on primitive circuits, hardware platforms, and applications as compared to this work. Therefore, the comparison is qualitative instead of quantitative. We first compare the supported applications of these three design methodologies. Manual designs can only handle regular applications such as parallel addition [50] and matrix multiply [10] due to design complexity. Skeleton-based flow requires the application to be regular at the top level while it has no restriction for the computational kernel at low level, as demonstrated in the TEA case study. The fully automated flow is the most flexible one with regard to the application type. However, the application size supported by the automated flow is not as large as the skeleton-based flow because the former has to explore the compute design space. Secondly, with respect to the quality of the generated designs, automated design flow cannot achieve optimal communication cost as discussed in Section III-A. However, since communication latency (typically 2-6 cycles) is relatively small compared to computation latency (tens to hundreds of cycles), the difference in performance between optimal design and suboptimal one would be minor. From an area point of view, the manual and skeleton-based flows have large white space in the designs, and hence require a larger design area than the automated flow. Finally, comparing their design efforts, the automated flow would be the easiest one to use. For the skeleton-based flow, the user is required to identify the patterns in the application; hence, it needs more effort. A skeleton-based synthesis tool executes quickly because it does not require design space exploration.

The implementation of primitive circuits is a key factor for memristor-based computation, including CIM. The latency of the multiplier (499 CC) and the adder (264 CC) that we used in the experiments is much greater than those operators implemented with CMOS technology. We can still achieve a notable speedup because of the massive parallelism CIM provides. However, even greater performance improvement can be obtained if these primitive circuits are implemented in a better way. On the other hand, only a few arithmetic operators have been implemented in memristor crossbars. It limits the number of algorithms that we can map to CIM. Since memristor-based computation is emerging, future researches will produce more and better circuits designs, and they will benefit CIM.

Endurance is a limit to memristor crossbar [6], [51]. Hence, it also restricts the potential of memristor-based computation and CIM. In this flow, we write memristors in a very high

frequency. Currently, the largest number of allowed write/erase operations on a memristor is only around 10^{12} [52], [53], but this number is believed to be able to reach 10^{15} [54] in the future. On the other hand, CIM is an accelerator that does not work as frequently as CPUs. Therefore, it also has a lower requirement for endurance.

VI. Conclusion and Future Work

Memristor-based CIM architecture requires a radically new development flow due to the characteristics of the memristor crossbar. We built a desirable synthesis flow for CIM based on an extension of algorithmic skeletons. In this flow, scheduling, placement, and routing algorithms are specified according to problems' DFG structures. We also investigated data patterns existing in stream applications and combined them with skeletons. To accelerate SystemC simulation, we integrated it with MPI. This work is verified using three applications, and their latency is compared against a multicore system. Primary results show the feasibility and potential of the proposed approach.

In future work, we will further investigate the communication between the RRAM and other components, such as the storage and the CPU. We are also developing a new Domain-Specific Language (DSL) to better describe CIM skeletons, especially with data patterns.

References

[1] M. Saecker and V. Markl, *Big Data Analytics on Modern Hardware Architectures: A Technology Survey*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 125–149. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36318-4_6

[2] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra *et al.*, "Top ten exascale research challenges," DOE ASCAC, Tech. Rep., 2014.

[3] T. Skotnicki, J. A. Hutchby, T.-J. King, H. S. P. Wong, and F. Boeuf, "The end of cmos scaling: toward the introduction of new materials and structural changes to improve mosfet performance," *IEEE Circuits and Devices Magazine*, vol. 21, no. 1, pp. 16–26, Jan 2005.

[4] L. O. Chua, "Memristor-the missing circuit element," *Circuit Theory, IEEE Transactions on*, vol. 18, no. 5, pp. 507–519, 1971.

[5] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *nature*, vol. 453, no. 7191, pp. 80–83, 2008.

[6] A. Morad, L. Yavits, S. Kvatinsky, and R. Ginosar, "Resistive gp-simd processing-in-memory," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 57:1–57:22, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2845084

[7] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 173:1–173:6. [Online]. Available: http://doi.acm.org/10.1145/2897937.2898064

[8] P. E. Gaillardon, L. Amar, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. D. Micheli, "The programmable logic-in-memory (plim) computer," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 427–432. [Online]. Available: http://ieeexplore.ieee.org/document/7459349/

[9] S. Hamdioui, L. Xie, H. A. D. Nguyen, M. Taouil, K. Bertels, H. Corporaal, H. Jiao, F. Catthoor, D. Wouters, L. Eike, and J. van Lunteren, "Memristor based computation-in-memory architecture for data-intensive applications," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 1718–1725. [Online]. Available: http://dl.acm.org/citation.cfm?id=2755753.2757210

[10] A. Haron, J. Yu, R. Nane, M. Taouil, S. Hamdioui, and K. Bertels, "Parallel matrix multiplication on memristor-based computation-in-memory architecture," in *2016 International Conference on High Performance Computing Simulation (HPCS)*. IEEE, July 2016, pp. 759–766.

[11] 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, no. 10, pp. 2054–2066, Oct 2014.

[12] J. J. Yang, D. B. Strukov, and D. R. Stewart, "Memristive devices for computing," *Nature nanotechnology*, vol. 8, no. 1, pp. 13–24, 2013.

[13] C. L. Seitz, *Introduction to VLSI systems*. Reading, MA: Addison-Wesley, 1980, ch. System timing, pp. 218–262.

[14] L. Xie, H. A. D. Nguyen, M. Taouil, S. Hamdioui, and K. Bertels, "Interconnect networks for memristor crossbar," in *Nanoscale Architectures (NANOARCH), 2015 IEEE/ACM International Symposium on*. IEEE, July 2015, pp. 124–129.

[15] S. H. Gerez, *Algorithms for VLSI design automation*. New York: Wiley, 1999, vol. 8.

[16] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.

[17] M. Zandifar, M. Abdul Jabbar, A. Majidi, D. Keyes, N. M. Amato, and L. Rauchwerger, "Composing algorithmic skeletons to express high-performance scientific applications," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 415–424.

[18] C. Nugteren and H. Corporaal, "Bones: An automatic skeleton-based c-to-cuda compiler for gpus," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 35:1–35:25, Dec. 2014.

[19] Y. Wang and Z. Li, "Gridfor: A domain specific language for parallel grid-based applications," *International Journal of Parallel Programming*, vol. 44, no. 3, pp. 427–448, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10766-014-0348-z

[20] M. Goli and H. Gonzalez-Velez, "Heterogeneous algorithmic skeletons for fast flow with seamless coordination over hybrid architectures," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. IEEE, Feb 2013, pp. 148–156.

[21] K. Benkrid, D. Crookes, J. Smith, and A. Benkrid, "High level programming for fpga based image and video processing using hardware skeletons," in *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*. IEEE, March 2001, pp. 219–226.

[22] K. Benkrid and D. Crookes, "From application descriptions to hardware in seconds: a logic-based approach to bridging the gap," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 4, pp. 420–436, April 2004.

[23] J. Yu, R. Nane, A. Haron, S. Hamdioui, H. Corporaal, and K. Bertels, "Skeleton-based design and simulation flow for computation-in-memory architectures," in *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, July 2016, pp. 165–170.

[24] 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, 2010.

[25] J. Borghetti, Z. Li, J. Straznicky, X. Li, D. A. Ohlberg, W. Wu, D. R. Stewart, and R. S. Williams, "A hybrid nanomemristor/transistor logic circuit capable of self-programming," *Proceedings of the National Academy of Sciences*, vol. 106, no. 6, pp. 1699–1703, 2009.

[26] G. Rose, J. Rajendran, H. Manem, R. Karri, and R. Pino, "Leveraging memristive systems in the construction of digital logic circuits," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 2033–2049, June 2012.

[27] 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.

[28] 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, p. 305205, 2012.

[29] 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.

[30] P. J. Kuekes, D. R. Stewart, and R. S. Williams, "The crossbar latch: Logic value storage, restoration, and inversion in crossbar circuits," *Journal of Applied Physics*, vol. 97, no. 3, p. 034301, 2005. [Online]. Available: http://dx.doi.org/10.1063/1.1823026

[31] P. L. Thangkhiew, R. Gharpinde, P. V. Chowdhary, K. Datta, and I. Sengupta, "Area efficient implementation of ripple carry adder using memristor crossbar arrays," in *2016 11th International Design Test Symposium (IDT)*, Dec 2016, pp. 142–147.

[32] R. Gnanasekaran, "A fast serial-parallel binary multiplier," *IEEE Trans. Comput.*, vol. 34, no. 8, pp. 741–744, Aug. 1985. [Online]. Available: http://dx.doi.org/10.1109/TC.1985.1676620

[33] E. Lehtonen, J. H. Poikonen, and M. Laiho, *Memristive Stateful Logic*. Cham: Springer International Publishing, 2014, pp. 603–623. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-02630-5_27

[34] C.-L. Tsai, F. Xiong, E. Pop, and M. Shim, "Resistive random access memory enabled by carbon nanotube crossbar electrodes," *Acs Nano*, vol. 7, no. 6, pp. 5360–5366, 2013.

[35] S. Lee, J. Sohn, Z. Jiang, H.-Y. Chen, and H.-S. P. Wong, "Metal oxide-resistive memory using graphene-edge electrodes," *Nature communications*, vol. 6, no. 8407, pp. 1–7, 2015.

[36] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen, "Open cell library in 15nm freepdk technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, ser. ISPD '15. New York, NY, USA: ACM, 2015, pp. 171–178. [Online]. Available: http://doi.acm.org/10.1145/2717764.2717783

[37] I. R. Committee, "International technology roadmap for semiconductors 2.0," Tech. Rep., 2015. [Online]. Available: www.itrs2.net/

[38] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[39] I. Ashraf, V. Sima, and K. Bertels, "Intra-application data-communication characterization," in *Proc. 1st International Workshop on Communication Architectures at Extreme Scale*, Frankfurt, Germany, July 2015, pp. 1–11.

[40] I. Ashraf, "Communication driven mapping of applications on multicore platforms," Ph.D. dissertation, Delft University of Technology, Delft, Netherlands, April 2016.

[41] C. Luk and et al., "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI '05*. New York, NY, USA: ACM, 2005, pp. 190–200.

[42] A. Canis, J. Choi, B. Fort, B. Syrowik, R. L. Lian, Y. T. Chen, H. Hsiao, J. Goeders, S. Brown, and J. Anderson, *LegUp High-Level Synthesis*. Cham: Springer International Publishing, 2016, pp. 175–190. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-26408-0_10

[43] D. K. Campbell, "Clumps: a candidate model of efficient, general purpose parallel computation," Ph.D. dissertation, University of Exeter, 1994.

[44] A. S. Initiative, "Systemc," 2016. [Online]. Available: http://accellera.org/downloads/standards/systemc

[45] D. J. Wheeler and R. M. Needham, *TEA, a tiny encryption algorithm*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 363–366. [Online]. Available: http://dx.doi.org/10.1007/3-540-60590-8_29

[46] Intel, "Xeon processor x7460," 2008. [Online]. Available: http://ark.intel.com/products/36947/Intel-Xeon-Processor-X7460-16M-Cache-2_66-GHz-1066-MHz-FSB

[47] A. C. Torrezan, J. P. Strachan, G. Medeiros-Ribeiro, and R. S. Williams, "Sub-nanosecond switching of a tantalum oxide memristor," *Nanotechnology*, vol. 22, no. 48, p. 485203, 2011. [Online]. Available: http://stacks.iop.org/0957-4484/22/i=48/a=485203

[48] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629677

[49] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 469–480. [Online]. Available: http://doi.acm.org/10.1145/1669112.1669172

[50] H. A. D. Nguyen, L. Xie, M. Taouil, R. Nane, S. Hamdioui, and K. Bertels, "Computation-in-memory based parallel adder," in *Nanoscale Architectures (NANOARCH), 2015 IEEE/ACM International Symposium on*. IEEE, July 2015, pp. 57–62.

[51] P. Pouyan, E. Amat, and A. Rubio, "Memristive crossbar memory lifetime evaluation and reconfiguration strategies," *IEEE Transactions on Emerging Topics in Computing*, vol. PP, no. 99, pp. 1–1, 2016.

[52] F. Miao, J. Yang, J. Strachan, W. Yi, G. Ribeiro, and R. Williams, "Memristor with channel region in thermal equilibrium with containing region," Mar. 1 2016, uS Patent 9,276,204. [Online]. Available: https://www.google.com/patents/US9276204

[53] M.-J. Lee, C. B. Lee, D. Lee, S. R. Lee, M. Chang, J. H. Hur, Y.-B. Kim, C.-J. Kim, D. H. Seo, S. Seo *et al.*, "A fast, high-endurance and scalable non-volatile memory device made from asymmetric ta2o5-x/tao2- x bilayer structures," *Nature materials*, vol. 10, no. 8, pp. 625–630, 2011.

[54] K. Eshraghian, K. R. Cho, O. Kavehei, S. K. Kang, D. Abbott, and S. M. S. Kang, "Memristor mos content addressable memory (mcam): Hybrid architecture for future high performance search engines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 8, pp. 1407–1417, Aug 2011.