Efficient In-Memory Processing Using Spintronics

Zamshed Chowdhury, Jonathan D. Harms, S. Karen Khatamifard, Masoud Zabihi, Yang Lv, Andrew P. Lyle, Sachin S. Sapatnekar, Ulya R. Karpuzcu, Jian-Ping Wang
University of Minnesota, Twin Cities

Abstract—As the overhead of data retrieval becomes forbidding, bringing processor logic to the memory where the data reside becomes more energy-efficient. While traditional CMOS structures are unsuited to the tight integration of logic and memory, emerging spintronic technologies show remarkable versatility. This paper introduces a novel spintronics-based processing-in-memory (PIM) framework called *computational RAM (CRAM)* to solve data-intensive computing problems.

Index Terms—CRAM, MTJ, processing-in-memory, spintronics, STT-MRAM.

1 Introduction

Today's processors are inadequately equipped to address the computational demand of big data analytics as data set sizes grow exponentially with time. Hardware paradigms have moved towards greater specialization to handle this challenge, and specialized units for memory-centric computing are vital to any future solution. Technology scaling has further enhanced the need for memory-centric computing as it has improved logic efficiency more than data communication. As a result, communication energy dominates computation energy and even the cleverest latency-hiding techniques cannot conceal the overhead of communication.

An effective way to overcome this bottleneck and maintain data locality is to embed compute capability into the main memory: *Processing in-memory (PIM)* can effectively address the communication bottleneck through distributed processing of data at the source, obviating the need for intensive energy-hungry communication. PIM features a rich design space, which spans full-fledged processors and co-processors residing in memory [1]. However, until recently, such promising studies could not render practical prototype designs due to the incompatibility of the state-of-the-art logic and memory technologies. The emergence of 3D-stacked architectures solved this problem partially by enabling *processing near-memory, PNM* [2], [3], [4], but genuine *processing in-memory* has remained elusive.

This paper introduces a *high-density reconfigurable spintronics-based* platform providing *true processing-in-memory* semantics, as opposed to most CMOS-based solutions which only deliver processing-near-memory, leaving the highly demanded potential in energy-efficiency untapped. The resulting *Computational RAM*, CRAM, platform can configure logic blocks of different functionality within a RAM array with rows which can be accessed in parallel. Thus, a CRAM-based solution not only meets true PIM semantics, but also facilitates reconfigurability which enables tailoring computational and memory resources to the demands of different algorithms or working sets.

A CRAM array can serve as both, a stand-alone full-fledged processor or a domain-specific co-processor attached to a host processor. From an application standpoint, algorithms are evolving, and such evolution may render very different requirements for hardware acceleration than imposed by the previous generation algorithms the hardware is tailored to. The reconfigurability of CRAM helps it sidestep the inflexibility of any non-reconfigurable design. The key characteristic of emerging application domains is data-intensity.

This work is supported by DARPA Non-Volatile Logic program, NSF SPX grant no. 1725420, and by C-SPIN, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA. Chowdhury and Harms equally contributed to this work.

Our preliminary analysis indicates that CRAM can outperform CMOS solutions in energy efficiency with competitive throughput: CRAM can deliver higher throughput at iso-energy, or lower energy consumption at iso-throughput. The idea of the CRAM has been proposed in a brief description in [5], but as we show, substantial refinement is necessary to take this idea to a practical implementation across the system stack.

2 Computational Random Access Memory

Although several methods for building spintronic logic have been proposed in the past, none can be easily integrated with memory because their structural difference from memory cells would cause a break in memory regularity. In contrast, CRAM uses a small modification to the magnetic tunnel junction (MTJ) based memory cell [6], [7] to enable logic operations. A major advantage over conventional PNM is that multiple operations can proceed in parallel within the memory array, which is particularly visible for computations such as dot products with many independent operations.

We next outline the hardware structure of CRAM and its reconfiguration to build logic functions. The CRAM concept is unique in its ability to reconfigure memory as logic with minimal changes to a standard spintronic memory array. This is in contrast with competing approaches (e.g., [8], [9]) that embed dedicated processing elements in a memory array or near its periphery, or utilize expensive table look-up operations to implement logic functions. In principle, a memory array could perform logic in much the same way as an FPGA [10], but the additional overhead required to access look-up tables could compromise the regularity and/or density of the memory array.

CRAM in memory mode: CRAM uses the 2T(ransistor) 1M(TJ) cell architecture shown in Fig. 1a. This is similar to the standard 1T1M magneto-resistive RAM (MRAM) cell in terms of memory function, except that the CRAM cell features a second transistor that allows for logic operations to be performed within the array. When the wordline for memory (WLM) is logic high, the cell operates as a standard MTJ memory cell, i.e., it behaves identically to the 1T1M MRAM cell. The MTJ is accessible for read/write operations over bitline (BL) and bitline bar (BLB).

CRAM in logic mode: For the logic mode, we show the operation of a two-input CRAM gate in Fig. 1b. For a two-input logic operation, the first two cells are loaded with the input data, either by retrieving the result of a previous CRAM operation or by writing the data to RAM using the memory mode. The bitline for logic (BLL) is enabled for these cells, which connects all of them to the shared logic line (LL), and a voltage pulse is applied to BL2 while BL[0:1] are grounded. The three MTJs are now reconfigured into the circuit in Fig. 1c [6],

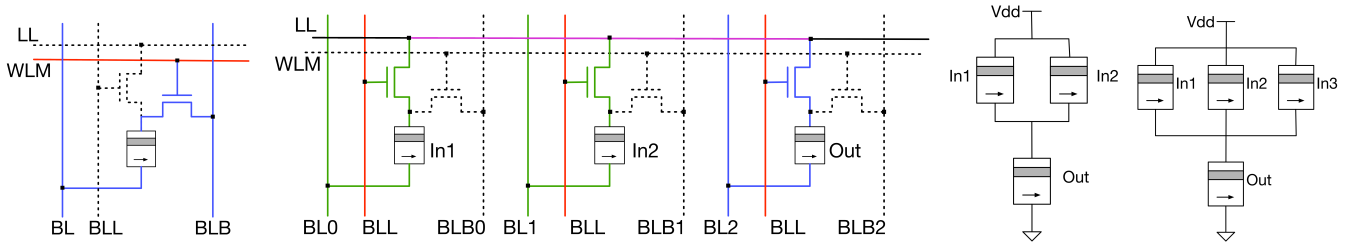


Fig. 1: (a) CRAM cell in (default) memory mode. (b) 2-input CRAM gate in logic mode. An implementation of (c) 2-input NAND, (d) inverting 3-input MAJ, using MTJs.

communicating through LL. The initial states of the three MTJs determine the current through the output MTJ. If this current is large enough, the state of the output MTJ switches. Thus, computation entails switching the state of one (output) MTJ by passing currents through the input MTJs.

CRAM supports *true in-memory* computing by reconfiguring cells as logic gates *within* the memory array. Further, because all cells in the array are identical, the order of applied voltage pulses (as opposed to the physical layout) determines inputs and outputs. This allows great flexibility in reconfiguring the CRAM array as memory or as various types of logic gates.

In logic mode, CRAM facilitates two types of reconfigurability: (i) each MTJ can serve as an input or as an output in a logic gate, as a function of evolving computational demands of the workload over time; (ii) for a fixed input-output assignment, the logic configuration is reprogrammable and can realize different logic functions by selecting the proper voltage pulses. For a fixed input-output assignment, the two input configuration can implement NOT, AND, NAND, NOR, and OR functions; and a three-input configuration, all of these as well as the majority function, MAJ. Reconfiguration (considering a fixed input-output assignment) entails, first, setting the output to a known initial state, and then, applying a fixed voltage pulse (specific to the logic function to be implemented). For each configuration, the final output state depends on the input MTJ states.

For an MTJ, the free layer direction with respect to the fixed layer, i.e., antiparallel (AP) and parallel (P), defines logic 1 and 0, respectively. For a two-input gate, logic state 00 corresponds to both input MTJs being P; 01 or 10, to one of the MTJs being AP; and 11, to both being AP. To evaluate a NAND, the output MTJ is set to P. Next, CRAM is configured as shown in Fig. 1b to derive the structure in Fig. 1c. Since the P configuration has a lower resistance than AP, the parallel resistance of the two input MTJs varies with the input logic state, implying that the current flow through the output MTJ depends on the inputs. The current is the smallest for the 11 case where both inputs are in the AP state. By optimizing the applied voltage pulse, we can ensure that this current is inadequate to switch the output MTJ (but the currents for the 00, 01, and 10 cases are sufficient for switching the output), such that the output remains in the preset state of 0. Thus, the NAND function is implemented. Other gates can be implemented similarly. Fig. 1d shows the structure for a three-input majority gate (MAJ3) that sets the output to the majority state of the three inputs. If the resistance of two or more inputs is low enough (i.e., if two or more inputs are in P state), the current becomes sufficient to switch the output.

A basic set of computational functions can be implemented by combining a set of universal NAND or MAJ gates. The voltage pulse applied on the BL lines (through appropriate voltage multiplexers) varies with the logic function. Reconfiguring cells to perform a new operation requires the application of voltage pulses to BL and BLL, and turning off WLM. This is easy to accomplish since it has a similar complexity as addressing elements of the memory array. We omit further circuit details as they are not the focus of this paper. We next overview proof-of-concept implementations of a basic set of computational functions; others can be realized similarly.

3 Implementing Computational Functions

A full adder takes inputs A, B, and C and provides two output bits, the sum S = A $\oplus$ B $\oplus$ C, and the output carry C_{out} = MAJ(A,B,C). This function (like any other) can clearly be implemented using a set of universal NAND gates, but is built more compactly using a pair of MAJ evaluations [11]. A full adder computation proceeds in three (sequentially dependent) steps, as shown in Fig. 2:
Step 1: Compute the output carry C_{out} = MAJ(A,B,C).
Step 2: Compute D = E = INV(C_{out}).
Step 3: Compute the sum S = MAJ(A,B,C,D,E).

This implementation uses two MAJ gates in two stages and an inversion stage to obtain INV(C_{out}). In contrast, a NAND gate implementation can be shown to use 9 logic stages. Since each stage of logic requires the evaluation of a configuration similar to Fig. 1b, the computation time depends on the number of stages, implying that the MAJ-based full adder is preferable. Multi-bit adders can be derived from full adders using standard techniques, ranging from ripple-carry adders to various faster schemes that speed up the carry chain.

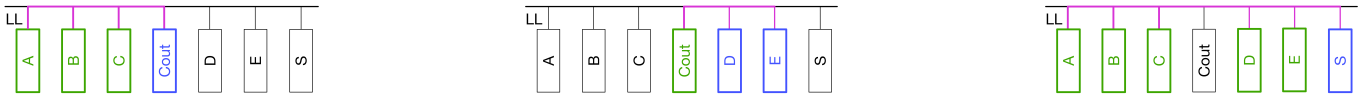
The problem of performing computations in the CRAM array has both a temporal aspect, to avoid data conflicts, and a spatial aspect, where data layout is optimized to maximize parallelism and to ensure that the latency of transmitting data within a configured "gate" is modest. To illustrate this, we consider the scheduling of operations of a ripple-carry adder (similar considerations hold for other types of adders with faster carry chains), consisting of MAJ-based full adders. For an n-bit adder, if all operands are stored within the same row, they must share the same LL. This induces a sequential dependence, requiring $3n$ steps to perform n-bit addition.

TABLE 1: Scheduling operations for a ripple-carry adder.

Time	1	2	3	4	5	6
Bit 0	$C_{out,1}$	–	D_0, E_0	S_0	–	–
Bit 1		$C_{out,2}$	–	D_1, E_1	S_1	–
Bit 2			$C_{out,3}$	–	D_2, E_2	S_2

If the operands lie in different (e.g., adjacent) rows whose LLs can (dis)connect by switching (off) on a communicating transistor, *switch*, computations in separate rows can proceed in parallel, or be pipelined by turning on the corresponding *switch*. Orchestrating computations in CRAM hence requires careful temporal and spatial scheduling of operations (the latter, by data layout optimization). One of the major advantages of CRAM is that similar independent operations can proceed in parallel along different LLs. This is different from most of the PNM, where the need to perform computations at the periphery inherently introduces a serial bottleneck.

Table 1 illustrates the sequence of pipelined computations for the first few bits of an n-bit adder, where each bit is placed in a separate row, and the LLs are isolated using a *switch*. We assume unit delays for each step to enable lock-step computation, completing an n-bit addition in $n + 3$ steps. The



(a) Step 1: C_{out} = MAJ(A,B,C) (b) Step 2: D = E = INV(C_{out}) (c) Step 3: S = MAJ(A,B,C,D,E)

Fig. 2: Executing a full adder operation on the CRAM array. The inputs are shown in green and the output in blue.

computation of Bit 0 starts at time 0 when its *Step 1* is initiated. *Step 1* of Bit 1 can start at time 1, as all of Bit 1's operands (specifically, $C_{out,1}$) are ready. However, *Step 2* of Bit 0 also requires $C_{out,1}$ to compute D_0 and E_0. Note that the CRAM computation in Fig. 1b only allows an MTJ to participate in *one* gate at any given time (otherwise it would effectively short-circuit the involved gates by connecting their LL lines). The best schedule in this case begins the computation of (*Step 1* of) Bit 1 at time 1, pauses the computation of Bit 0 for one time slot, and then resumes the computation of (*Step 2* of) Bit 0 at time 2. Similarly, a waiting period is inserted after $C_{out,1}$ and $C_{out,2}$ are computed, as illustrated in the table.

A multiplier can be implemented efficiently using arrays of full adders arranged as Wallace/Dadda trees using 3-to-2 compression schemes [12]. An alternative structure for a multiplier, which trades space efficiency for a longer computation time, is a multiply-accumulate structure in which partial products are successively generated and added to a set of memory cells that function as an accumulator for the sum of such products.

The above operations lay the groundwork for more complex computations. For example, the dot product of two integer vectors X = $[x_1, \cdots, x_m]$ and Y = $[y_1, \cdots, y_m]$, where all x_is and y_is are n-bit integers, is given by $\sum_{i=1}^{m} x_i \cdot y_i$. Viewing this as a sum of multiply operations, and a multiplier as a computation that adds a set of partial products, it is easy to see that the operation can be performed similar to multiplication: in effect, a dot product involves the addition of all the partial products generated in multiplying the integers x_is and y_is.

4 Evaluation

We continue with an end-to-end quantitative characterization of CRAM's energy efficiency potential. We choose an emerging application domain, *bioinformatics*, which can benefit from CRAM. Bioinformatics constitutes a major class of memory-intensive big-data applications, where hardware platforms that collocate logic and memory to avoid the latency and energy overhead of expensive data transfers with the potential to enhance energy efficiency significantly. These applications process long text strings representing biological sequence data and require fast pattern matching in very large databases residing in memory. The computations are dominated by integer arithmetic and comparison operations – excellent acceleration targets for CRAM as indicated in Sections 2 and 3. Table 2 lists the benchmark applications from the BioBench suite [13] deployed for this study.

TABLE 2: Bioinformatics benchmarks deployed.

Benchmark	Description
blastn (bl), fasta_dna (fdna)	DNA Sequence Searching
blastp (bl-p), fast_prot (fprot)	Protein Sequence Searching
clustalw (cl)	Multiple Sequence Alignment
hmmer (hm)	Sequence Profile Searching
mummer (mm)	Genome-Level Alignment
protpars (pr)	Phylogenetic Analysis
tigr	Sequence Assembly
plsa	Parallel Linear Space Alignment
geneNet (gn)	Sequence Search
semphy (sm)	Phylogenetic Tree Construction
snp	Single Nucleotide Polymorphism
svm	Recursive Feature Elimination

Our comparisons are made against a state-of-the-art competing solution. Due to recent advances in 3D stacking, practical implementations that catalyze *processing near-memory* have been rendered practical. These solutions include the Hybrid Memory Cube (HMC) [2], Hybrid Bandwidth Memory (HBM) [3], and Active Memory Cube (AMC) [4], and previous studies report by up-to 15× improvement in energy efficiency due to 3D stacking, when compared to conventional, classic execution [14]. While these approaches represent the best PNM solutions, they still require data to be brought to the periphery of the memory for processing. To quantify the advantage of PIM using the CRAM approach, for representative bioinformatics applications from Table 2, we compare the energy efficiency in terms of energy-delay product (EDP) [15] of CRAM to a hypothetical, HMC-based conventional (CMOS) PNM-based accelerator. To this end, we obtained instruction (i.e., operation) mixes of the benchmark applications using Pin [16], and derived the overall energy efficiency using per operation estimates (we deployed the peak energy efficiency and throughput reported for HMC [2] in estimating energy and delay per operation, not to favor CRAM). CRAM memory interface is modeled using [17]. For the specific technology used for individual memory cells, we consider two potential candidates for implementing CRAM in a 10nm technology:

- A *near-term* STT-MRAM technology, with an MTJ tunneling magnetoresistance ratio (TMR) of 100%, a parallel resistance, R_P, of the MTJ of 2KΩ, and a transistor resistance of 4KΩ. The critical switching current, J_{c0}, is taken to be 13.5MA/cm^2, and the MTJ switching time is set to 0.5ns.
- A *projected long-term* STT-MRAM technology, with an MTJ TMR of 600%, as experimentally demonstrated in [18] (this is possibly conservative, since TMRs of 1000% have been projected within the next decade [19], but is adequate to show the promise of CRAM). The value of J_{c0} is reduced to 1MA/cm^2, which is realistically achievable. The MTJ switching time is 0.5ns, R_{AP} is 5KΩ, and the transistor resistance remains at 4KΩ.

We implement a carry lookahead-adder based on the MAJ-based full adder from [11], and a multiplier based on a Wallace tree structure [12]. The energy and delay per operation estimates for CRAM come from these implementations. Fig. 3 shows the results of this comparison when all computation is performed *near-memory* using HMC vs. *in-memory* using CRAM. We analyze both 2D and 3D-stacked implementations of the CRAM co-processor, and report the energy efficiency under current and projected CRAM technologies. Overall, we observe that

- In the near-term technology, the 2D CRAM implementation can improve the energy efficiency by about 1.8×; the 3D-stacked correspondent, by up to 11×.
- In the projected technology, the CRAM array can unlock more than 2.2 (2.8)× energy efficiency in 2D (3D), above and beyond the near-term number. For both technologies, the energy per operation of the CRAM memory cell is dominated by the overhead of the memory periphery. Therefore, the 3D-stacked CRAM implementation can significantly reduce this cost by reducing array interconnect overheads.

CRAM also incurs a latency overhead per basic operation, due to the sequencing of low-level computations. This preliminary study is incomplete and is an indicator of the potential benefits of CRAM. Further exploration of temporal and spatial scheduling techniques can mask this overhead by exploiting fine grain parallelism in computational steps at the level of individual operations.

5 Conclusion & Discussion

This paper introduces Computational RAM, CRAM, a high-density reconfigurable spintronics-based platform facilitat-

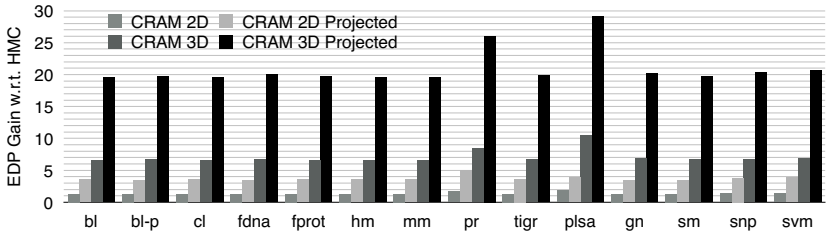


Fig. 3: EDP gain of CRAM w.r.t. HMC [2] for a representative set of bioinformatics applications.

ing logic and memory. CRAM features true processing-in-memory semantics, and by that differs from most CMOS-based processing-near-memory solutions.

CRAM is unique in combining multi-grain (possibly dynamic) reconfigurability with true processing *in* memory semantics. The resistive element based *Associative Processor* [20] and the DRAM technology based *DRAF* [21] on the other hand, both represent look-up-table based solutions which can support reconfigurable fabrics like FPGA. A similar concept to CRAM based on resistive RAMs is introduced in [22], but not a practical implementation. The SRAM-based *Compute Cache* [23] can carry out different vector operations such as copy, search, comparison in the cache, but CRAM can perform a wider range of computations in the memory array. Furthermore, maintaining data coherence among cores which constitute PNM logic is an issue [24], [25] which is not the case for CRAM due to the absence of dedicated cores (with full-fledged memory hierarchies) to implement logic operations.

This paper proposes a method for performing true in-memory computation using STT-MRAMs. The CRAM design is based on configuring segments of the memory array as resistive dividers, since the state of an STT-MRAM cell is expressed as one of two resistance values. This idea does not directly extend to DRAMs, where the state is stored as a charge, and the problems with performing similar computations involve charge sharing, loss of state, etc. In fact, to the best of our knowledge, there is no true in-memory approach available for DRAM structures as prior methods (e.g., [26]) perform computations at the edge of memory. This implies that the bitlines required to transport data to the edge of memory constitute a serial bottleneck. On the other hand, in CRAM, it is possible to simultaneously perform multiple operations in parallel in multiple rows. A comparable design based on memristor-based technologies, MAGIC [27], also uses resistive division. However, this work does not go into a system-level evaluation of applications running on this platform, as we do in this paper. At the same time, such arrays suffer from significant endurance issues as compared to STT-MRAMs.

The CRAM architecture can perform computations locally within the array for any intermediate operation, where major data movement tasks are generally only required for input data and the eventual output data. In other words, over an n-step operation where n is sufficiently large, the cost of performing these write operations can be amortized. In contrast, in a conventional PNM architecture, data must be taken out to the periphery of the memory array on every operation, and then the result must be brought back to an array location, incurring this type of overhead in each of the intermediate steps.

Furthermore, the conventional PNM structure suffers from an inherent serial bottleneck in that while one data set is being taken to the periphery, no other operation may be performed within the array because the bit lines are shared over all rows in the array. The CRAM array, on the other hand, can perform operations in parallel *in every row* of the array because the logic lines for each row are separate.

Based on the encouraging preliminary analysis from Section 4, our future work is directed to enable CRAM's energy efficiency potential by bridging the gap between the current and projected gains in energy efficiency.

REFERENCES

[1] G. H. Loh *et al.*, "A Processing in Memory Taxonomy and a Case For Studying Fixed-function PIM," in *Workshop on Near-Data Processing in conjunction with MICRO*, 2013.
[2] [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.18.3-memory-FPGA/HC23.18.320-HybridCube-Pawlowski-Micron.pdf
[3] [Online]. Available: http://www.amd.com/en-us/innovations/software-technologies/hbm
[4] R. Nair *et al.*, "Active Memory Cube: A Processing-in-Memory Architecture for Exascale Systems," *IBM Journal of R.&D.*, vol. 59, no. 2/3, 2015.
[5] J.-P. Wang and J. Harms, "General Structure for Computational Random Access Memory (CRAM)," 2015, US Patent 9224447 B2.
[6] A. Lyle *et al.*, "Direct Communication between Magnetic Tunnel Junctions for Nonvolatile Logic Fanout Architecture," *Applied Physics Letters*, vol. 97, no. 152504, 2010.
[7] J. Wang *et al.*, "Programmable Spintronics Logic Device Based on a Magnetic Tunnel Junction Element," *Applied Physics Letters*, vol. 97, no. 10D509, 2005.
[8] S. A. Wolf *et al.*, "The Promise of Nanomagnetics and Spintronics for Future Logic and Universal Memory"," *Proc. of IEEE*, vol. 98, no. 12, 2010.
[9] T. Hanyu *et al.*, "Spintronics-based Nonvolatile Logic-in-Memory Architecture Towards an Ultra-Low-Power and Highly Reliable VLSI Computing Paradigm," in *DATE*, 2015.
[10] O. Goncalves *et al.*, "Non-volatile FPGAs Based on Spintronic Devices," in *DAC*, 2013.
[11] C. Augustine *et al.*, "Low-power Functionality Enhanced Computation Architecture Using Spin-based Devices," in *International Symposium on Nanoscale Architectures*, 2011.
[12] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, 2nd ed. New York, NY: Oxford University Press, 2010.
[13] A. Jaleel *et al.*, "Last Level Cache (LLC) Performance of Data Mining Workloads on a CMP: A Case Study of Parallel Bioinformatics Workloads." *HPCA*, 2006.
[14] C. Weis *et al.*, "Design Space Exploration for 3D-stacked DRAMs." *DATE*, 2011.
[15] R. Gonzalez and M. Horowitz, "Energy Dissipation in General Purpose Microprocessors," *The Energy Journal*, vol. 31, no. 9, 1996.
[16] C.-K. Luk *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.
[17] X. Dong *et al.*, "Circuit and Microarchitecture Evaluation of 3D Stacking Magnetic RAM (MRAM) as a Universal Memory Replacement," in *DAC*, 2008.
[18] S. Ikeda *et al.*, "Tunnel Magnetoresistance of 604% at 300K by Suppression of Ta Diffusion in CoFeBMgOCoFeB Pseudo-spin-valves Annealed at High Temperature," *Applied Physics Letters*, vol. 93, no. 082508, 2008.
[19] A. Hirohata *et al.*, "Roadmap for Emerging Materials for Spintronic Device Applications," *IEEE Transactions on Magnetics*, vol. 51, no. 10, 2015.
[20] L. Yavits *et al.*, "Resistive Associative Processor," *CAL*, vol. 14, no. 2, 2015.
[21] M. Gao *et al.*, "DRAF: A Low-power DRAM-based Reconfigurable Acceleration Fabric," *ISCA*, 2016.
[22] D. Strukov, "Hybrid CMOS/Nanodevice Circuits with Tightly Integrated Memory and Logic Functionality," in *Nanotech*, 2011.
[23] S. Aga *et al.*, "Compute Caches," *HPCA*, 2017.
[24] A. Boroumand *et al.*, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *CAL*, vol. 16, no. 1, Jan 2017.
[25] M. Gao *et al.*, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *PACT*, 2015.
[26] S. Jeloka *et al.*, "A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory," *IEEE JSSCC*, vol. 51, no. 4, April 2016.
[27] S. Kvatinsky *et al.*, "Magic: Memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, Nov 2014.