Endurance Management for Resistive Logic-In-Memory Computing Architectures

Saeideh Shirinzadeh*, Mathias Soeken†, Pierre-Emmanuel Gaillardon‡, Giovanni De Micheli†, Rolf Drechsler*§
*Department of Mathematics and Computer Science, University of Bremen, Germany
†Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
‡Electrical and Computer Engineering Department, University of Utah, Salt Lake City, UT, USA
§Cyber-Physical Systems, DFKI GmbH, Bremen, Germany

***Abstract*—Resistive Random Access Memory (RRAM) is a promising non-volatile memory technology which enables modern in-memory computing architectures. Although RRAMs are known to be superior to conventional memories in many aspects, they suffer from a low write endurance. In this paper, we focus on balancing memory write traffic as a solution to extend the lifetime of resistive crossbar architectures. As a case study, we monitor the write traffic in a *Programmable Logic-in-Memory* (PLiM) architecture, and propose an endurance management scheme for it. The proposed endurance-aware compilation is capable of handling different trade-offs between write balance, latency, and area of the resulting PLiM implementations. Experimental evaluations on a set of benchmarks including large arithmetic and control functions show that the standard deviation of writes can be reduced by 86.65% on average compared to a naive compiler, while the average number of instructions and RRAM devices also decreases by 36.45% and 13.67%, respectively.**

I. Introduction

Resistive Random Access Memory (RRAM) has attracted high attention since it is providing new opportunities from memory to architecture design. RRAMs are known to be a non-volatile memory technology with lower power consumption and high scaling capability which has the potential to replace conventional memories [1]. In order to store information in RRAMs, their internal resistance state should be switched between a low or high value [2]. This property enables executing Boolean logic operations within RRAMs, which is of high interest for the design of in-memory computing circuits and systems. Integration of storage and computation units enabled by the resistive switching property of RRAMs, also provides an alternative solution for the CMOS technology used for today's computer architectures, which is already suffering from limited scalability, leakage power, and reliability issues [3], [4].

As a non-volatile memory technology, RRAMs outperform SRAM and eDRAM in the aforementioned cases. However, they have a limited write endurance. In the best existing RRAM architectures, a cell can endure about 10^{10} [5] to 10^{11} write counts [6]. There are many approaches that try to improve memory lifetime. Some approaches are based on a write balancing scheme [7], [8], [9] mainly proposed for another non-volatile memory technology known as *Phase-Change Memory* (PCM), which has a lower write endurance compared to RRAM [1]. There are also other approaches which use error correcting techniques to fix the hard failures [10].

In this paper, we focus on balancing the write traffic as a solution to extend the lifetime of RRAMs used for in-memory computing. We target the *Programmable Logic-in-Memory* (PLiM) computer architecture proposed in [11] as a case study. We survey the reasons of unbalanced write traffic in PLiM and propose an endurance management scheme for it to get a more uniform distribution of the writes over all memory cells. Experiments on large arithmetic and control benchmark functions show considerably improved write balance. For some functions, the standard deviations of the write counts are reduced by more than 90% compared to a naive approach.

II. Logic-in-Memory Write Traffic

When using RRAMs for in-memory computing, some memory cells face much higher write counts than others. In this section, we survey the reasons for unbalanced write traffic in some existing logic-in-memory computing approaches.

So far a variety of approaches has been proposed for synthesis of logic-in-memory computing circuits, mostly based on material implication (IMP: $p \rightarrow q = \bar{p} \vee q$) [12], [13] and some using other basic operations [14], [15], [11]. In all these approaches, the target function can only be executed using a certain number of RRAM devices and after a number of operations, which increases rapidly as the size of the function increases. Hence, lowering the number of operations and the number of resistive memories is considered the main target of current in-memory computing synthesis approaches.

In [16], an IMP-based NAND gate was proposed. The gate is implemented with two resistive switches and the NAND function is executed within three computational steps. Regardless of the required operations to load the primary inputs of the gate into the two allocated devices, only one of them, the so-called work device, is rewritten after each operation while the other keeps its initial value. Using IMP for synthesis, this unbalanced distribution of writes happens due to the lack of commutativity property, which results in higher write traffic in the memory cell storing the output of the operation. Nevertheless, the write traffic can be spread more evenly over the entire memory by considering extra RRAM devices to replace those with high write counts. This will also require additional operations to copy the contents of RRAMs into fresh ones or ones with lower write counts. However, being unwilling to spend more time or area, this intrinsic unbalanced write can become even worse. For example, [17] proposes a synthesis approach that considers only two work resistive devices besides N input devices, where N is the number of input variables of the Boolean function. It is obvious that the work devices used in implementations based on such an approach suffer from short lifetime.

In [11], an operation based on the 3-input majority function $\langle xyz \rangle = xy \vee xz \vee yz = (x \vee y)(x \vee z)(x \vee z)$ was proposed to perform Boolean functions on a crossbar array of *Bipolar Resistive Switches* (BRS). The operation called 3-input *Resistive Majority* (RM_3) exploits an intrinsic majority property of resistive memories. Denoting the value stored in the memory by Z, and the signals applied to its top and bottom terminals, by P and Q, respectively, the updated value of Z by RM_3 is formally expressed as $Z \leftarrow PZ \vee \overline{Q}Z \vee P\overline{Q} =$

978-3-9815370-8-6/17/$31.00

$\langle P\overline{Q}Z\rangle = RM_3(P,Q,Z)$. This basic operation incorporates both of a majority-of-three and an inversion and thus can be used as a universal computing operator.

RM_3 was shown more efficient compared to material implication w.r.t. both area and latency when using *Majority-Inverter Graphs* (MIG, [18]) [19]. Regarding write endurance, RM_3 has more flexibility in comparison with IMP by sharing the writes between three operands instead of one. Nevertheless, RM_3 does not benefit from the commutativity property of the majority function due to the inversion of the second operand. Moreover, similarly to IMP, storing the result of the RM_3 in one of the involved memory devices can cause an unbalanced write if the same device is updated each time in a chain of operations.

III. Case Study: Endurance-Aware Compilation for PLiM Architecture

A. Preliminaries

1) Majority Inverter Graphs: MIG is a data structure for efficient representation of Boolean functions which consists of 3-input majority nodes and complemented edges [18], [20]. In general MIGs can be efficiently exploited for logic-in-memory computing due to benefiting from the resistive majority property enabled by RM_3 [19], [21].

In [18], a Boolean algebra was proposed for MIGs to obtain more efficient graphs w.r.t. the considered criteria. The following set (Ω) includes the primitive axioms which are theoretically sufficient to reach any logically equivalent MIG from an arbitrary one [18].

$$\Omega \begin{cases} \textbf{Commutativity} - \boldsymbol{\Omega.C} \\ \langle xyz\rangle = \langle yxz\rangle = \langle zyx\rangle \\ \textbf{Majority} - \boldsymbol{\Omega.M} \\ \begin{cases} \langle xyz\rangle = x = y & \text{if } x = y \\ \langle xyz\rangle = z & \text{if } x = \bar{y} \end{cases} \\ \textbf{Associativity} - \boldsymbol{\Omega.A} \\ \langle xu\langle yuz\rangle\rangle = \langle zu\langle yux\rangle\rangle \\ \textbf{Distributivity} - \boldsymbol{\Omega.D} \\ \langle xy\langle uvz\rangle\rangle = \langle\langle xyu\rangle\langle xyv\rangle z\rangle \\ \textbf{Inverter Propagation} - \boldsymbol{\Omega.I} \\ \overline{\langle xyz\rangle} = \langle \bar{x}\bar{y}\bar{z}\rangle \end{cases}$$

A more advanced set of identities called Ψ was also proposed in [18] to make the length of axioms required for transforming MIGs practically possible. Here, we only introduce one Ψ axiom, which is referred in this paper, called *Complementary Associativity* $\Psi.C = \langle x,u,\langle y,\bar{x},z\rangle\rangle = \langle x,u,\langle y,x,z\rangle\rangle$.

Using MIGs for resistive in-memory computing, the nodes with multiple complemented edges impose extra costs in both area and delay. An extended inverter propagation axiom from right to left denoted by $\Omega.I_{R\to L(1-3)}$ was proposed in [19] to control the extra costs caused by complemented edges. $\Omega.I_{R\to L(1-3)}$ includes the three transformations (1) $\langle\bar{x}\bar{y}\bar{z}\rangle = \overline{\langle xyz\rangle}$, (2) $\overline{\langle\bar{x}\bar{y}z\rangle} = \langle xy\bar{z}\rangle$, and (3) $\langle\bar{x}\bar{y}z\rangle = \overline{\langle xy\bar{z}\rangle}$.

2) PLiM Architecture: The PLiM architecture is capable of performing logic operations on a regular RRAM array. To execute a Boolean function, PLiM translates its MIG representation to a set of RM_3 instructions. Implementing logic operations within memory cells requires to control the distribution of signals and the scheduling of operations considering latency, area, and management [11], [21].

The PLiM controller consists of a wrapper for the RRAM array. It reads the instructions from the memory array and performs computing operations (RM_3) within the memory array. As a wrapper, PLiM uses the addressing and read/write peripheral circuitries of the RRAM array. When the control signal is equal to zero, the controller is *off* and the whole array works as a standard RAM system. When the controller is on, the circuit starts performing computation. The controller consists of a simple finite state machine and few work registers, in order to operate the RM_3 instruction [11]. The instruction format consists of the first operand A, the second operand B and the destination Z. Operands A and B are then read from constants or from the memory array, and RM_3 is performed during the write operation to the memory location Z by setting P to A and Q to B. This value of node Z is then updated by $Z \leftarrow \langle A\overline{B}Z\rangle$. When the write operation is completed, a program counter is incremented, and a new cycle of operation is triggered.

3) PLiM Compiler: An MIG-based compiler for PLiM was proposed in [21], which aims at optimizing the resulting programs w.r.t. the number of instructions and RRAM devices. The sequential nature of PLiM makes the number of MIG nodes a determining factor in the length of instructions required for computing a Boolean function. However, other factors such as complemented edges and fanouts can also be a reason for more instructions. According to RM_3, any MIG node with more or less than a single complemented edge requires extra instructions. Moreover, an MIG node needs at least one single fanout child which can be rewritten by the result of the computation. Otherwise, more instructions must be performed to copy one of the children in another RRAM to be used later at its fanout target. Any of these issues with complemented edges and fanouts requires two additional instructions and one RRAM and thus should be addressed during optimization procedure.

Algorithm 1 shows the MIG rewriting proposed for PLiM compiler [21]. A cycle of the algorithm starts with conventional MIG rewriting for node minimization proposed in [18], [20] (steps 2–4) and terminates with the extension of inverter propagation axioms (steps 5–6) to tackle the issue of complemented edges.

```
1 for (cycles = 0; cycles < effort; cycles++) do
2 |   Ω.M; Ω.D_{R→L};
3 |   Ω.A; Ψ.C;
4 |   Ω.M; Ω.D_{R→L};
5 |   Ω.I_{R→L(1−3)};
6 |   Ω.I_{R→L};
7 end
```

Algorithm 1: MIG rewriting for PLiM compiler [21]

A compilation approach including two parts of node selection and translation was also proposed in [21] to reduce the number of RRAM devices for PLiM implementations. The node selection aims at finding an order for computing nodes of a given MIG which results in a smaller number of RRAMs, while the node translation chooses the operands of RM_3 to compute a node with the minimum number of instructions and RRAMs.

The nodes are selected for computation according to a priority list of all computable candidates, i.e. nodes whose children have been already computed. The nodes are first compared based on the number of RRAMs which can be released after computing the node. This reduces the number of RRAMs because the released ones can be reused for the next computations. In case of an equal number of releasing fanins for two nodes, their position in the priority list is decided based on their fanout level indices. In this case, the compiler chooses a node whose fanout has lower level index, i.e. lower waiting

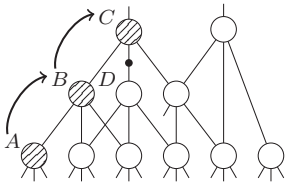

Fig. 1. Example MIG vulnerable to unbalanced write caused by PLiM's area latency considerations

time for the node's value to be used at the fanout target. This way, the RRAMs allocated for nodes with long waiting times are blocked for shorter duration and can be reused sooner, which reduces the number of RRAMs.

In this work, we utilize the same node selection scheme but try to integrate endurance considerations into the node selection procedure.

B. Endurance Management for PLiM

In this section, we propose four techniques which jointly balance the write traffic for the PLiM computer. Two techniques work directly based on the write counts of the memory cells, and the others based on MIG rewriting and compilation to address the intrinsic sources of unbalanced write existing in MIGs.

1-2) Direct Endurance Management: During PLiM compilation, write endurance can be easily considered whenever an RRAM device is requested such that if there are any freed RRAMs at the time of request, the one with the smallest write count is returned. We call this *"minimum write count strategy"* which is a simple but also an essential technique to lower the deviation of writes. However, it is not still sufficient to obtain a good write balance. In this work, we also consider the conflicts of area and latency with the write distribution of the resulting PLiM programs. Indeed, we try to address those parts of the compilation procedure that worsen the write traffic but keep the number of instructions and RRAMs as low as possible.

For example, computing the MIG shown in Fig. 1 by PLiM, the writes cannot be balanced completely over all allocated RRAMs unless additional instructions and RRAM devices are consumed. We assume that node A is already computed and node B is the best candidate for PLiM to be computed next. In this case, the compiler chooses the RRAM storing the value of node A, let us call it X, as the destination of RM_3. This is because the RRAMs storing the two other children nodes of B have more than one fanout. Selecting a node with multiple fanouts as the destination of RM_3 will require two extra instructions and one extra RRAM to copy the values of the nodes to use them later at their other fanout targets. Therefore, node A is written again regardless of its current number of writes, and its content is replaced by the result of computation of node B. A similar situation occurs when computing node C. To compute node C, the compiler first sets the only complemented child shown with a dotted edge, i.e. node D, as the second operand of RM_3 for the sake of efficiency. Between the two other remaining children, only node B has a single fanout whose value is stored in RRAM X. Accordingly, again X will be selected as the destination of operation in order to avoid extra costs, while the RRAM keeping the value of node D might have much smaller write count.

As shown in the example above, rewriting the same RRAM repeatedly can continue for a large number of node computations depending on the situation of single fanouts and complemented edges in the MIG. This condition causes an unbalanced write distribution which cannot be controlled without extra costs in terms of execution time and area. A solution for this problem is to consider a maximum allowed number of writes per RRAM device. This way, the RRAMs which reach this maximum write count are kept out of the free set and cannot be requested anymore. As a result, a fresh RRAM should be allocated which increases the number of RRAMs. Moreover, the number of instructions can also increase since the ideal case of nodes with a single fanout child can only be exploited if the write threshold is satisfied. Otherwise, two extra instructions must be performed for computing the candidate node to load one of its children nodes into a fresh RRAM or one with a valid write count to be used as the destination. We refer to this endurance management technique as the *"maximum write count strategy."*

```
1  for (cycles = 0; cycles < effort; cycles++) do
2      Ω.M; Ω.D_{R→L};
3      Ω.I_{R→L(1−3)};
4      Ω.I_{R→L};
5      Ω.A;
6      Ω.I_{R→L(1−3)};
7      Ω.I_{R→L};
8      Ω.M; Ω.D_{R→L};
9      Ω.I_{R→L};
10 end
```

Algorithm 2: Endurance-aware MIG rewriting for PLiM architecture

The two strategies mentioned above can be utilized for any in-memory computing method and memory endurance management in general. To achieve a more comprehensive endurance management scheme for the PLiM computer, in the following we refer to some key characteristics specifically applicable to PLiM. We first study how a slightly different MIG rewriting algorithm can enhance the distribution of writes over the memory. Then, we propose an endurance-aware node selection during compilation to get a better write traffic.

3) Endurance-Aware MIG Rewriting: Algorithm 2 describes the proposed endurance-aware MIG rewriting. The algorithm improves the write balance as well as maintaining the main role of MIG rewriting, i.e. minimizing the number of instructions, by applying $\Omega.M$ and $\Omega.D_{R\to L}$ to reduce the number of nodes, and inverter propagation axioms, $\Omega.I_{R\to L(1-3)}$ and $\Omega.I_{R\to L}$ to control the extra instructions required for complemented edges.

In comparison to Algorithm 1, we have removed $\Psi.C$ which was disadvantageous due to removing a single complemented edge of an MIG node, which is actually considered an ideally low cost case for the RM_3 operation. Instead, $\Omega.A$ is sandwiched by two sets of inverter propagation axioms (Steps 3–7) to maximize the gain by presence of ideal nodes with a single inverted child. This also reshapes the MIG and creates more opportunities for further reduction in the number of nodes (Step 8). At the end, $\Omega.I_{R\to L}$ (Step 9) is applied to eliminate the costly nodes with three inverted children.

Algorithm 2 does not explicitly target cases such as repeatedly writing a memory cell due to the condition of fanouts or complemented edges, but it can decrease the number of nodes further and results in a more useful distribution of complemented edges. This effects lead to a lower total number of writes and increase the possibility of regular change of the switching device.

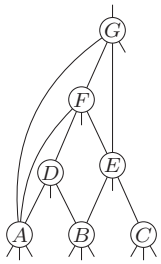

Fig. 2. Example MIG vulnerable to unbalanced write due to possessing nodes with long storage durattions

```
1  forall the MIG nodes A and B ∈ Candidate Set do
2      if (A/B has the smaller fanout level index) then
3          Select A/B;
4      end
5      else if |releasing_RRAMs(A)| >
       |releasing_RRAMs(B)| then
6          Select A;
7      end
8      else
9          Select B;
10     end
11 end
```

Algorithm 3: Comparison of MIG nodes for the endurance-aware node selection

4) Endurance-Aware Node Selection: As explained before, the write traffic highly depends on the MIG features, including the condition of inverters and fanouts in the graph. Here, we discuss another MIG feature which can have a considerable effect on the write distribution.

The structure of the MIG can help to distribute the writes more evenly if RRAMs in use are released and reused with a similar frequency. Indeed, the issue with endurance management happens when some RRAM devices are blocked for a long time and some others are rewritten very often. In such a situation, the number of writes for the RRAMs used to execute the PLiM program can vary widely. Thus, some RRAM devices reach to the end of their lifetime much faster than others which shortens the normal duration that the PLiM computer can work reliably.

Fig. 2 shows an MIG with the problem explained above. Let us assume that node A is already computed and X_A designates the RRAM keeping its value. It is obvious that X_A is still required and cannot be released until node G is computed. Node A targets several nodes in higher levels which means longer waiting time for X_A, while all other nodes only target the nodes in the very next levels. In comparison, the RRAMs storing the values of nodes B and C, indicated by X_B and X_C, respectively, can be released when nodes D and E are computed and one of them can be rewritten again to compute node F. X_B and X_C can also be rewritten before by the values of their target nodes D and E. Considering that the initial number of writes has been equal for all the mentioned RRAMs, X_B and X_C might have been rewritten at lease one or two times more than X_A, which we call a blocked RRAM, when the root node G is computed.

The impact of blocked RRAMs on write traffic can be much more noticeable in large MIGs with high level differences. Nevertheless, write balance can be enhanced if we compute the nodes with long waiting time as late as possible. In the example above, the write traffic can be managed if the compiler computes nodes B and C before A. For this purpose, we reverse the priority of nodes in the compilation procedure. Thus, the candidate nodes with the smallest fanout level index, which means the shortest storage duration, are computed first. The second node selection principle, i.e. number of releasing RRAMs, is considered if there are several candidate nodes with equal fanout level indices. In this case, the node with the largest number of releasing RRAMs is selected for computation. Algorithm 3 describes the procedure to select between two candidate nodes.

Although using this endurance-aware node selection strategy for PLiM compiler improves the memory write balance, it has slightly lower efficiency in comparison with the initial node selection scheme. Nevertheless, the small increase in the number of instructions and RRAMs might not matter regarding the write traffic improvement.

It is worth noting that the unbalanced write distribution caused by blocked RRAM devices waiting for quite long time cannot be eliminated but only decreased. In fact, an evenly distributed write traffic for an MIG with large differences between fanout origins and targets is not possible by only postponing computation of nodes with longer waiting times. The sequential nature of PLiM architecture anyway imposes a waiting list of blocked RRAMs which cannot be released until the root node or nodes close to it are computed. We consider this factor as the main difficulty of endurance management for a generic MIG-based in-memory computing architecture constructed upon a resistive crossbar array.

The issue of blocked RRAMs could be considered as an objective during MIG rewriting to keep the level differences between connected nodes low. However, MIGs optimized this way might not be favorable w.r.t. the length of instructions. The proposed MIG rewriting focuses on reducing the number of nodes and complemented edges which have a direct impact on latency. Reducing the number of nodes already makes the connections between nodes more complicated which means higher number of nodes targeting farther levels.

IV. Experimental Evaluations

We have experimented our results on a set of benchmarks including 18 functions.[1] The benchmark set includes large arithmetic and random control functions possessing up to 1204 primary inputs and 1231 primary outputs. The number of cycles for MIG rewriting, i.e. effort, is set to 5 in all experiments. The standard deviation, which is known as a robust statistical metric, is used to describe the distribution of writes over entire memory cells required for computing an MIG representing a Boolean function. Minimum and maximum number of writes are also given for a more precise statistical description.

Table I shows the standard deviation, and minimum and maximum number of writes performed for the RRAM devices required by the PLiM computer for each benchmark function. The results are given in an incremental manner in different columns to clearly show the effect of each proposed endurance management technique, excluding the maximum write strategy. The improvement of standard deviation obtained by each technique is compared to the corresponding naive implementations, which only benefit from node translation but not MIG rewriting and node selection.

To see the effect of the general purpose compilation, we have also provided the statistical values representing the write distribution obtained by the PLiM compiler [21] in the second group columns. The compiler proposed in [21] tackles the

[1] http://lsi.epfl.ch/benchmarks

TABLE I. EXPERIMENTAL EVALUATION OF THE PROPOSED ENDURANCE MANAGEMENT TECHNIQUES

		naive		PLiM compiler [21]			Minimum write strategy			Minimum write strategy+endurance-aware MIG rewriting			Minimum write strategy+endurance-aware MIG rewriting and compilation		
	PI/PO	min/max	STDEV	min/max	STDEV	impr.	min/max	STDEV	impr.	min/max	STDEV	impr.	min/max	STDEV	impr.
adder	256/129	0/84	12.60	0/34	6.09	51.66%	2/9	0.90	92.86%	2/9	2.49	80.24%	2/13	1.55	87.70%
bar	135/128	0/87	11.97	0/122	7.97	33.41%	0/106	5.64	52.88%	0/26	3.07	74.35%	0/18	1.69	85.88%
div	128/128	2/459	121.98	1/757	227.73	-86.69%	2/596	233.38	-91.33%	1/381	140.49	-15.17%	1/340	119.53	2.01%
log2	32/32	1/286	49.92	2/89	15.73	68.48%	14/57	3.62	92.75%	7/55	3.24	93.51%	10/55	3.27	93.45%
max	512/130	1/77	11.80	1/105	11.35	3.81%	1/88	7.72	34.58%	2/88	7.41	37.20%	2/17	2.65	77.54%
multiplier	128/128	0/1196	179.90	2/289	32.77	81.78%	5/206	18.74	89.58%	4/24	1.69	99.06%	5/24	1.53	99.15%
sin	24/25	0/166	27.17	0/50	8.47	68.82%	6/40	2.79	89.73%	3/42	3.04	88.81%	3/48	4.42	83.73%
sqrt	128/64	2/440	145.16	1/360	94.11	35.16%	4/228	73.90	49.09%	4/267	82.34	43.28%	2/230	68.36	52.91%
square	64/128	0/577	95.43	0/48	4.35	95.44%	1/43	1.87	98.04%	1/28	1.95	97.96%	1/22	1.96	97.95%
cavlc	10/11	0/73	15.36	0/44	10.30	32.94%	0/18	3.16	79.43%	0/18	3.17	79.36%	0/13	2.36	84.64%
ctrl	7/26	0/28	7.44	1/38	9.10	-22.31%	1/15	3.12	58.06%	1/17	3.93	47.18%	1/11	1.79	75.94%
dec	8/256	1/6	0.46	2/5	0.45	2.17%	2/6	0.57	-23.91%	2/6	0.57	-23.91%	2/6	0.57	-23.91%
i2c	147/142	0/114	14.07	0/88	10.87	22.74%	0/24	4.23	69.94%	0/31	4.44	68.44%	0/16	3.04	78.39%
int2float	11/7	0/49	13.28	0/52	11.87	10.61%	0/22	4.70	64.61%	0/19	3.93	70.41%	0/12	2.69	79.74%
mem_ctrl	1204/1231	0/553	80.47	0/175	21.07	73.81%	0/89	11.78	85.36%	0/56	9.12	88.67%	0/97	11.03	86.29%
priority	128/8	0/496	45.57	0/101	14.37	68.46%	2/35	5.98	86.88%	2/45	7.52	83.50%	2/49	7.43	83.70%
router	60/30	0/52	8.71	0/33	6.30	27.66%	0/21	3.57	59.01%	1/23	3.91	55.11%	1/19	3.57	59.01%
voter	1001/1	2/156	31.66	0/188	35.09	-10.83%	13/793	19.11	39.64%	13/25	1.53	95.17%	10/23	1.59	94.98%
AVG		0.5/272.16	48.49	0.55/163.72	29.33	30.95%	2.94/133.11	22.48	57.07%	**2.38**/67.70	15.07	64.42%	2.33/**56.27**	**13.27**	**72.17**%

PI/PO: number of primary inputs/primary outputs, min/max: minimum/maximum number of writes, STDEV: standard deviation of write counts, impr.: improvement of standard deviation is calculated compared to naive

TABLE II. NUMBER OF INSTRUCTIONS AND RRAMS REQUIRED FOR ENURANCE-AWARE COMPILATION OF PLIM

		naive		Endurance-aware MIG rewriting		Endurance-aware rewriting and compilation	
Benchmark	PI/PO	$\#I$	$\#R$	$\#I$	$\#R$	$\#I$	$\#R$
adder	256/129	2844	512	1656	258	1657	355
bar	135/128	8136	523	5103	264	5103	391
div	128/128	146617	687	100071	620	101318	496
log2	32/32	78885	1597	48692	1344	48665	1347
max	512/130	6731	1021	3902	660	4042	749
multiplier	128/128	76156	2798	45966	4297	45847	4301
sin	24/25	12479	438	9156	377	9211	373
sqrt	128/64	60691	375	39434	333	39464	371
square	64/128	54704	3272	29757	2387	30044	2714
cavlc	10/11	1919	262	1045	148	1058	186
ctrl	7/26	499	66	268	38	273	48
dec	8/256	822	257	777	258	777	258
i2c	147/142	3314	545	1946	305	1971	365
int2float	11/7	648	99	368	50	378	63
mem_ctrl	1204/1231	113244	8127	74577	4357	74588	4724
priority	128/8	2461	315	1417	138	1464	140
router	60/30	503	117	371	66	364	73
voter	1001/1	38002	1749	20208	1337	20406	1667
AVG		33814.16	1264.44	**21373**	**957.61**	21479.44	1034.50

PI/PO: number of primary inputs/primary outputs, $\#I$: number of RM_3 instructions, $\#R$: number of RRAMs

number of instructions and RRAMs as the cost metrics, however, the techniques employed also improve write traffic due to the shorter release time of RRAMs. Comparison of the first three group columns shows that the write balance improvement of the PLiM compiler [21] is increased from the overall average value of 30.95% to 57.07% after only adding the presented minimum write strategy. This improvement increases further up to 64.42% by use of the proposed endurance-aware MIG rewriting instead of that used in [21]. Finally, adding the endurance-aware compilation to the proposed MIG rewriting and minimum write count strategy reduces the standard deviation of writes further by 72.17% in comparison to the naive approach.

Table II compares the number of instructions and RRAM devices required for the PLiM programs using only endurance-aware rewriting and both endurance-aware rewriting and compilation with the naive approach. As shown in the table, the average number of instructions and RRAMs slightly increases by adding the endurance-aware compilation.[2] Nevertheless, considering the 72.17% improvement achieved in the write traffic, the PLiM programs obtained by the endurance-aware MIG rewriting and compilation also have average reductions of 36.48% and 18.18% in the number of instructions and RRAMs, respectively. It should be noted that the minimum write count strategy does not influence the number of required instructions and RRAMs. The minimum write strategy selects an RRAM with the smallest write count from the set of free RRAMs when requested. This only contributes to a better distribution of the write counts, and cannot increase or decrease the number of instructions or RRAM devices.

Affording a number of instructions and RRAMs higher than that shown in Table II, almost any desired write traffic is accessible using the suggested maximum write count strategy. Table III shows the results of full endurance management exploiting the minimum and maximum write strategies as well as the endurance-aware MIG rewriting and compilation. The number of instructions, RRAM devices and the standard deviation of the writes performed to execute the PLiM programs under maximum write constraints of 10, 20, 50, and 100 are presented to show the relation between the improvement in write distribution and additional penalties w.r.t. delay and area. Indeed, Table III provides different trade-offs between write endurance, latency, and area for the resulting implementations.

According to Table I, results for some benchmarks remain unchanged for constraint values which are higher than the benchmarks' natural maximum number of writes. This unchanged values are indicated by dashes in the table and their value can be found on the corresponding cells with a lower write constraint. As expected, the number of instructions and RRAMs decrease as the write constraint becomes looser, while the write deviation worsens. An average improvement of 96.8% in standard deviation is obtained when a maximum allowed write value of 10 is set in the compiler. This improvement in the write distribution is obtained at a cost of 50.59% increase in the number of RRAMs compared to the naive approach. Nevertheless, the number of instructions has slightly increased and still shows a considerable reduction in comparison to the naive solution. Results for a maximum write

[2] Applying endurance-aware compilation, #R increases by 8.02%. Nonetheless, according to Table I AVG STDEV reduces more by 11.94% (from 15.07 to 13.27) which confirms the effectiveness of the compilation technique.

TABLE III. RESULTS OF FULL ENDURANCE MANAGEMENT WITH MAXIMUM WRITE STRATEGY FOR WRITE VALUES OF 10, 20, 50, AND 100

		10			20			50			100		
Benchmark	PI/PO	#I	#R	STDEV	#I	#R	STDEV	#I	#R	STDEV	#I	#R	STDEV
adder	256/129	1659	362	1.12	1657	355	1.55	–	–	–	–	–	–
bar	135/128	5351	603	1.37	5103	391	1.69	–	–	–	–	–	–
div	128/128	103057	11691	0.91	102187	5551	2.04	101615	2327	11.76	101431	1295	31.59
log2	32/32	51063	5559	0.85	49603	2643	1.36	48666	1347	3.25	48665	1347	3.27
max	512/130	4076	776	2.06	4042	749	2.65	–	–	–	–	–	–
multiplier	128/128	47819	5225	0.97	45854	4301	1.51	45847	4301	1.53	–	–	–
sin	24/25	9562	1069	1.18	9336	534	2.28	9211	373	4.42	–	–	–
sqrt	128/64	40991	4548	1.04	39956	2248	3.29	39642	994	14.55	39514	598	33.81
square	64/128	31411	3478	1.07	30051	2714	1.95	30044	2714	1.96	–	–	–
cavlc	10/11	1067	186	2.05	1058	186	2.36	–	–	–	–	–	–
ctrl	7/26	274	48	1.74	273	48	1.79	–	–	–	–	–	–
dec	8/256	777	258	0.57	–	–	–	–	–	–	–	–	–
i2c	147/142	1985	377	2.54	1971	365	3.04	–	–	–	–	–	–
int2float	11/7	381	63	2.47	378	63	2.69	–	–	–	–	–	–
mem_ctrl	1204/1231	78216	9194	2.07	75403	5940	5.32	74619	4777	10.57	74588	4724	11.03
priority	128/8	1545	206	2.27	1494	140	6.27	1464	140	7.43			
router	60/30	367	74	2.91	364	73	3.57	–	–	–	–	–	–
voter	1001/1	21538	2350	0.86	20408	1667	1.58	20406	1667	1.59	–	–	–
AVG		22285.50	2559.27	**1.55**	21661.94	1568.11	2.66	21507.61	1173.77	4.27	**21488.50**	**1091.50**	6.47

PI/PO: number of primary inputs/primary outputs, $\#I$: number of RM_3 instructions, $\#R$: number of RRAMs, STDEV: standard deviation of write counts, – shows that the value has not been changed

count of 100 can be considered as a good trade-off, due to an overall improvement of 86.85% in the write balance as well as reducing the average number of instructions and RRAMs by 36.45% and 13.67%, respectively.

V. CONCLUSION

In this paper, we addressed the issue of unbalanced write traffic in resistive crossbar in-memory computing architectures. As a case study, we proposed techniques for endurance management of a programmable logic-in-memory architecture. The proposed approach maintains a trade off between write endurance and other cost metrics of the resulting implementations including area and latency. The experimental results show a reduction of 86.65% in the standard deviation of writes as well as noticeable improvements in the lengths of instructions and number of RRAM devices.

ACKNOWLEDGMENTS

This research was supported by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative, by H2020-ERC-2014-ADG 669354 CyberCare, and by the Swiss National Science Foundation projects 200021_146600 and 200021_169084.

REFERENCES

[1] J. Wang, X. Dong, Y. Xie, and N. P. Jouppi, "Endurance-aware cache line management for non-volatile caches," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 1, pp. 4:1–4:25, 2014.

[2] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 449–464, 2008.

[3] S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, vol. 19, no. 4, pp. 23–29, 1999.

[4] J. W. McPherson, "Reliability trends with advanced CMOS scaling and the implications for design," in *IEEE Custom Integrated Circuits Conference*, 2007, pp. 405–412.

[5] H. Y. Lee, Y. S. Chen, P. S. Chen, P. Y. Gu, Y. Y. Hsu, S. M. Wang, W. H. Liu, C. H. Tsai, S. S. Sheu, P. C. Chiang, W. P. Lin, C. H. Lin, W. S. Chen, F. T. Chen, C. H. Lien, and M. J. Tsai, "Evidence and solution of over-reset problem for HfOX based resistive memory with sub-ns switching speed and high endurance," in *IEEE International Meeting on Electron Devices*, 2010, pp. 19.7.1–19.7.4.

[6] Y. B. Kim, S. R. Lee, D. Lee, C. B. Lee, M. Chang, J. H. Hur, M. J. Lee, G. S. Park, C. J. Kim, U. I. Chung, I. K. Yoo, and K. Kim, "Bi-layered rram with unlimited endurance and extremely uniform switching," in *Symposium on VLSI Technology*, 2011, pp. 52–53.

[7] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *International Symposium on Computer Architecture*, 2009, pp. 14–23.

[8] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 14–23.

[9] N. H. Seong, D. H. Woo, and H. S. Lee, "Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping," in *International Symposium on Computer Architecture*, 2010, pp. 383–394.

[10] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ecp, not ecc, for hard failures in resistive memories," in *International Symposium on Computer Architecture*, 2010, pp. 141–152.

[11] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (PLiM) computer," in *Design, Automation & Test in Europe*, 2016, pp. 427–432.

[12] E. Lehtonen and M. Laiho, "Stateful implication logic with memristors," in *IEEE/ACM International Symposium on Nanoscale Architectures*, 2009, pp. 33–36.

[13] 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 Trans. VLSI Syst.*, vol. 22, no. 10, pp. 2054–2066, 2014.

[14] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic – memristor-aided logic," *IEEE Trans. Circuits Syst. II*, vol. 61, no. 11, pp. 895–899, 2014.

[15] E. Linn, R. Rosezin, S. Tappertzhofen, U. Böttger, and R. Waser, "Beyond von neumannlogic operations in passive crossbar arrays alongside memory operations," *Nanotechnology*, vol. 23, no. 30, 2012.

[16] 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, pp. 873–876, 2010.

[17] E. Lehtonen, J. Poikonen, and M. Laiho, "Two memristors suffice to compute all Boolean functions," *Electronics Letters*, vol. 46, pp. 230–231, 2010.

[18] L. G. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Design Automation Conference*, 2014, pp. 194:1–194:6.

[19] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for RRAM-based in-memory computing using majority-inverter graphs," in *Design, Automation & Test in Europe*, 2016, pp. 948–953.

[20] L. G. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE T-CAD*, vol. 35, no. 5, pp. 806–819, 2016.

[21] M. Soeken, S. Shirinzadeh, P.-E. Gaillardon, L. G. Amarú, R. Drechsler, and G. De Micheli, "An MIG-based compiler for programmable logic-in-memory architectures," in *Design Automation Conference*, 2016, pp. 117:1–117:6.