Generation of Verified Programs for In-Memory Computing

Saman Froehlich Rolf Drechsler
Group of Computer Architecture, University of Bremen, Germany and Cyber-Physical Systems, DFKI GmbH, Germany
froehlich@uni-bremen.de drechsler@uni-bremen.de

***Abstract*—In order to overcome the von Neumann bottleneck, recently the paradigm of in-memory computing has emerged. Here, instead of transferring data from the memory to the CPU for computation, the computation is directly performed within the memory. ReRAM, a resistance-based storage device, is a promising technology for this paradigm. Based on ReRAM, the PLiM computer architecture and LiM-HDL, an HDL for specifying PLiM programs have emerged.**

In this paper, we first present a novel levelization algorithm for LiM-HDL. Based on this novel algorithm, large circuits can be compiled to PLiM programs. Then, we present a verification scheme for these programs. This scheme is separated into two steps: (1) A proof of purity and (2) a proof of equivalence. Finally, in the experiments, we first apply our levelization algorithms to a well-known benchmark set, where we show that we can generate PLiM programs for large benchmarks, for which existing levelization algorithms fails. Then, we apply our proposed verification scheme to these PLiM programs.

I. Introduction

The von Neumann architecture as introduced by John von Neumann in 1945 [1] is used in most computer systems, today. In the von Neumann architecture, the memory is used for storage of instructions and data at the same time. It has been extended with sophisticated memory hierarchies, today. These memory hierarchies allows for fast access to a small amount of data, while requiring longer access times to the majority of data which is stored in higher levels of the hierarchy. This is also known as the *von Neumann bottleneck*. Consequently, the von Neumann architecture is efficient, as long as the time required for computation is significantly larger compared to the time used for data processing.

However, recently, new applications have emerged, such as deep learning and the *Internet of Things* (IoT). These applications come with their own requirements and challenges. Deep learning requires processing of large amounts of data. With the von Neumann bottleneck, the von Neumann architecture becomes very inefficient for deep learning and thus, specialized hardware devices have been designed to mitigate this inefficiency (e.g., [2], [3]). Characteristic for IoT are small devices, which are very constrained in terms of area and power consumption. In IoT, the majority of computation is shifted from data centers to edge devices [4].

Currently, a resistance based storage device called *Resistive Random Access Memory* (ReRAM), is emerging. It is especially appealing due to its inherent in-memory computation capabilities and allows for the computation of different universal functions. Consequently, ReRAM can compute any Boolean function as long as an implementation in terms of a set of universal functions is given. Additionally, ReRAM's low power consumption, fast switching capabilities and scalability make it an excellent candidate for a technological foundation for IoT and edge devices [5], [6], [7]. However, due to the lack of EDA and verification tools, this technology is not widely used, yet [8]. And while techniques for EDA have been the focus of several recent publications (e.g. [9], [10], [11]), the research in the field of verification is still very sparse. However, specially the field of verification is crucial for todays computing systems and has been the focus of several publications for several other applications in the last years.

In order to overcome the von Neumann bottleneck, the *Programmable Logic-in-Memory* (PLiM) computer architecture has been proposed in [7]. Besides the control logic, the ReRAM arrays are the core of the PLiM computer architecture. These banks are used as storage and computational unit at the same time and consequently, the PLiM computer architecture does not suffer from the von Neumann bottleneck. Additionally, the PLiM computer architecture is of particular interest for IoT and edge devices, as the resulting architecture operates at low power [7]. To allow for an easy and efficient implementation of programs for the PLiM computer architecture, in [11] an HDL-based synthesis scheme for in-memory computing was proposed. This synthesis scheme consists of a preprocessing step, a levelization step and the final compilation step. However, the synthesis scheme proposed in [11] is not suitable for large designs as the levelization step is very complex and requires many computations.

In this paper, we close the gap in the field of verification for in-memory computing and the PLiM computer architecture by proposing a verification strategy for LiM-HDL programs. First, we generate an SMT representation of the formal definition at behavioral/RTL-level. Then, after compilation to the final PLiM program, we present a method for transforming this PLiM program into another SMT representation. Using a miter structure, we can then compare these two SMT representations and check their equivalence using SMT solvers. Further, we propose a new levelization method for the synthesis scheme proposed in [11]. This levelization method which allows the compilation of more complex designs compared to the state-of-the-art. Using our proposed verification strategy, we can

This work was supported by the German Research Foundation (DFG) within the Project *PLiM* (DR 287/35-1). Additionally, we would like to thank Christan F. Coors for his contribution and support for this work.

ensure that the final PLiM programs implement the formal definition given as HDL-based designs. In the experiments, we show the efficiency and scalability of our proposed levelization method and verification strategy by presenting a way to generate LiM-HDL benchmarks from an *Majority Inverter Graph* (MIG)-based representation [12]. We then generate a large benchmark set from the state-of-the-art EPFL benchmarks.

Thus, to conclude the main contributions of this paper are as follows:

1) We are the first to introduce formal verification to synthesis of in-memory programs.
2) We present a method for levelization, which allows the current state-of-the-art synthesis method for the PLiM computer architecture to scale for larger designs.
3) We present a method to transform MIGs-based definition of circuits to LiM-HDL to allow for an easy definition of benchmarks.

II. Preliminaries

In this section, preliminary knowledge is introduced. First, in Section II-A, the RM_3 operation which is native to ReRAM is introduced. Then, in Section II-B the PLiM computer architecture, wordline parallelism and LiM-HDL are described briefly. Finally, in Section II-C SAT and SMT are introduced.

A. Resistive Majority Operation RM_3

A ReRAM device can be viewed as a two-terminal device with the terminals P (called the *wordline operand*), Q (called the *bitline operand*) and its internal resistance state Z (called the *host operand*). If P is set to 1 (i.e., $V/2$) and Q is set to 0 (i.e., $-V/2$), the resistance state Z is set to 1 (i.e., a low resistance state). Correspondingly, if P is set to 0 and Q is set to 1, the resistance state Z is set to 0. In all other combinations (($P = 0$, $Q = 0$),($P = 1$, $Q = 1$)), Z remains unchanged. This behavior can be mapped to the majority operation $MAJ(P, \bar{Q}, Z) = P\bar{Q} + PZ + \bar{Q}Z$, which is commonly defined as the RM_3 operation $\mathrm{RM}_3(P, Q, Z) = MAJ(P, \bar{Q}, Z)$.

B. PLiM

The PLiM computer architecture has been proposed in [13]. At its core, this computer architecture consists of several ReRAM banks. Together with a sophisticated control circuitry, these ReRAM banks can be used for in-memory computation.

1) Wordline Parallelism: In [14], the PLiM computer architecture has been extended to allow for wordline parallel computation. Here, multiple operations can be performed in parallel, if these operations share a wordline operand. First all cells are initialized with their host operand Z. This can be done in parallel, by first initializing all cells to constant 1, by applying logical 1 as wordline operand and logical 0 as bitline operand to them. Then, the respective operands $\bar{Z}$ are applied to their bitlines, while logical 0 is applied to the wordline. Note that, if only Z instead of the inverse $\bar{Z}$ is present, this inverse needs to be computed first. However, inversions can also be computed in parallel. After initializing the cells, the shared wordline operand is applied to the wordline of the cells and the respective bitline operands are applied to the bitlines. This way, multiple operations which share a wordline operand are computed in parallel.

2) LiM-HDL: Recently, in [11], LiM-HDL has been proposed. LiM-HDL is an HDL which allows for the easy definition of PLiM programs. The LiM-HDL definition is transformed into an RTL graph, which consists of several nodes and can then be synthesized. The synthesis strategy for this RTL graph consists of three steps: (1) a preprocessing step, (2) a levelization step and (3) the final compilation step. Specially, the levelization step is of importance. Due to its complexity, in [11], two different methods have been proposed: The first method is an exact levelization procedure based on Branch&Bound, while the second method is a heuristic based on *Monte Carlo Tree Search* (MCTS).

C. SAT and SMT

The *Boolean Satisfiability Problem* (SAT) is one of the classic problems in computer science. For a given Boolean formula, the problem consists of determining whether an input combination exists such that the formula evaluates to true. This problem is NP-complete, meaning that there is no known algorithm for solving it in polynomial time complexity [15]. A concrete Boolean formula for which SAT can be checked is called an instance.

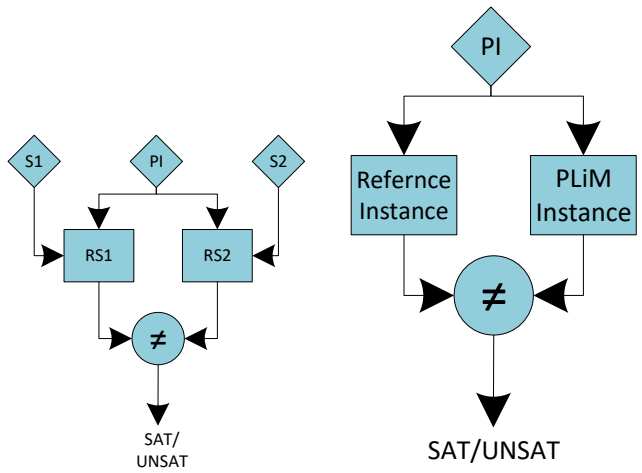
1) SMT: Since converting real-world problems into CNF can be difficult, an extension of SAT was invented, called *Satisfiability Modulo Theories* (SMT). SMT solvers are widely used to test, analyze and verify computer programs [16]. The problem is still focused on satisfiability of a given logical formula, but support for various theories based on practical computer science, such as integers, lists, strings and bitvectors are added. The input format for these solvers is much more sophisticated and generally based on the SMT LIB Standard [17], [18].

III. Related Work

To the best of our knowledge, we are the first to focus on verification of PLiM programs. The authors of [8] state, that there is a need for verification methods in in-memory computing, which further emphasizes the importance and the lack of research in this topic. However, work on verification and testing has been performed for specific applications and we briefly review some recent publications in this section.

In [19], the authors present a method for verifying circuit components of a phase change memory chip for artificial neural networks applications. The authors propose a design for test approach which partitions a module-based design and uses redundant memory circuits. Further, a scan chain is developed which allows monitoring of signals inside the circuit.

The authors of [20] propose a testing algorithm for 1T1R ReRAM crossbars. This algorithm allows for testing of specific cells inside the crossbar so that all cells can be tested for functionality, and consists of four stages which determine different parameters of the devices. During the test, the resistive switching parameters such as forming voltages, switching voltages, etc. of the cells are determined.



(a) Purity Miter (b) Equivalence Miter

Fig. 1. Miters

A compiler for automatic ReRAM generation and verification has been published in [21], including netlist generation and layout. Here, physical verification tools are used and a complete hardware system is generated. Note that, in contrast to our work, the focus is not on verifying programs for a general in-memory computer architecture, but on generating verified circuitries.

IV. Verification Strategy

Verification of circuits is crucial for todays computing systems. However, to the best of our knowledge, verification has not been applied to programs for the PLiM computer architecture, yet. In this section, we propose a verification scheme for these PLiM programs. As the initial state of the ReRAM devices might change the output of a program in case cells are not initialized during the execution, in addition to the *Primary Inputs* (PIs) of the circuit, we also have to take the initial states into account. Here, we need to make sure, that the result of the PLiM program is equivalent to that of the design regardless of the initial state. To reduce the complexity of the verification procedure, we divide it into two steps:

1) In the first step, we verify the *purity* of a program. A program can be called pure, if its behavior is independent of the initial state of the ReRAM crossbar. This is important, as the initial state of the crossbar is not controlled by the program.
2) In the second step, we prove the equivalence of the program to its HDL-based design.

Both steps are detailed in the following sections.

A. Proof of Purity

In order to proove the purity of a program, we model the initial state of the used ReRAM devices as additional, secondary inputs. Then, we build a miter-structure to check if two programs with the same primary inputs can have different results, if these additional, secondary inputs can be chosen freely.

Algorithm 1 Greedy Levelization

```
1: function LevelizeGreedy(C)
2:   \\ C is the set of nodes, which is to be levelized
3:   CSorted = SortByWLOp(C)
4:   while !CSorted.empty() do
5:     computableLevels = DetermineComputatbleLevels(CSorted)
6:     l = computableLevels.largest()
7:     currentLevelization.add(l)
8:     CSorted.remove(l)
9:   end while
10:  return levelization
11: end function
```

The miter-structure is shown in Figure 1(a). The models $RS1$ and $RS2$ each represent the given PLiM program - both PLiM programs have the same primary inputs. Then, the additional secondary inputs model the initial state of the cells. Consequently, for each cell, which is used during the computation an additional input has to be added. This miter can then be modeled as a SAT or SMT instance and checked for satisfiability.

B. Proof of Equivalence

After verifying the purity of the program, we now can proove its equivalence to the design before compilation to a PLiM program. We extract an SMT representation of the design. Then, after compiling the design to a PLiM program, it can be transformed to SMT as well. Since the PLiM program only consists of RM_3 operations, the transformation is straightforward. Finally, we again build a miter. Since we have shown that the program is pure, we do not need to proove equivalence for all initial states of the ReRAM array, but can restrict ourselves to fixed initial values and chose the PIs to be variable.

The corresponding miter is depicted in Figure 1(b). Here, the reference instance is the SMT representation of the design, while the PLiM instance is the SMT instance of the PLiM program. Both instances have the same PIs as inputs in this miter. The solver now checks, if a combination exists, for which the outputs of the instances differ.

V. Greedy Levelization

Efficient levelization is crucial for PLiM programs. Here, after compilation to an RTL graph, all nodes on the same levels need to share a wordline operand, and are then computable in parallel [11]. Consequently, a large number of levels leads to a large number of computational cycles.

In order to enable efficient parallelization and mapping to the crossbar, different levelization methods have been proposed in [11]: One exact Branch&Bound-based method and one heuristic based on MCTS. The Branch&Bound-based method finds a levelization with the smallest number of levels, while the MCTS-based method is a heuristic that can be applied to more complex designs. However, both methods do not scale well to larger designs.

In this section, we propose a heuristic, greedy algorithm for levelization of RM_3-based graphs. During each step, the largest set of nodes that share a wordline operand are added to the levelization greedily.

Our proposed heuristic algorithm can be seen in Algorithm 1. Here, in Line 3 we first sort the nodes by their wordline operand. Then, in Line 5 all computable levels are determined. Here, all nodes, which are computable are identified. A node is computable if all its three inputs are either already computed, are primary inputs or are constants. Then, these nodes are grouped by their wordline operand forming a set of computable levels. From these computable levels, the largest level is identified and then added to the levelization in Line 7. Further, the nodes are removed from the set of sorted nodes. This process is repeated until all nodes are added to the levelization.

VI. Experimental Results

In this section, we describe our experimental results. First, we describe our benchmark set in Section VI-A. Then, in Section VI-B, we show the benefits of our proposed levelization scheme compared to the state-of-the-art. Finally, in Section VI-C, we show the results for our proposed verification strategy.

All experiments have been performed on an Intel® Xeon® CPU E5-2630 v3 @ 2.40GHz with 64GB memory running Linux (Fedora release 30) for a crossbar with the wordsize 16.

A. Benchmarks

In our experiments, we include benchmarks from the well-known EPFL Combinational Benchmark Suite. The EPFL Combinational Benchmark Suite is a set of multiple natively combinational circuits designed to challenge modern logic optimization tools [22]. It consists of several arithmetic, random/control and 3 very large benchmarks. Each benchmark is well-documented and available in multiple file formats (Verilog, VHDL, BLIF and AIGER). A general overview over the benchmarks used and their *And-Inverter Graph* (AIG) implementation can be seen in Table I.

The EPFL Combinational Benchmark Suite was designed to fit a wide spectrum of optimization algorithms due to its variety of circuit types and complexity. We choose it because of its wide adoption and because the native combinational nature of the benchmarks corresponds to the purity property of PLiM programs.

1) Conversion to LiM-HDL: While all synthesizable Verilog modules can naively be made compatible with LiM-HDL by adding `LiMHDLbegin` and `LiMHDLend`, to take full advantage of the PLiM architecture, modules containing statements that are easily mapped to RM_3 operations and produce a shallow search tree for the levelization are beneficial to the performance and results of the synthesis process. Recently, MIGs have emerged as a promising alternative to AIGs [22]. MIGs are directed, acyclic graphs consisting of three-input majority nodes and regular/complemented edges.

Since $\mathrm{MAJ}_3(a, b, c) = \mathrm{RM}_3(a, \overline{b}, c)$, an MIG can easily be converted into LiM-HDL containing RM_3 operations. The order of the operands also has an influence on the result, since they determine the wordline and bitline values. The inversion of the second operand can be ignored, since the PLiM compiler tracks inverted edges internally.

TABLE I
An overview over the benchmarks used

Name	Inputs	Outputs	AND nodes	Levels
Adder	256	129	1020	255
Barrel shifter	135	128	3336	12
Divisor	128	128	44762	4470
Hypotenuse	256	128	214335	24801
Log2	32	32	32060	444
Max	512	130	2865	287
Multiplier	128	128	27062	274
Sine	24	25	5416	225
Square-root	128	64	24618	5058
Round-robin arbiter	256	129	11839	87
Coding-cavlc	10	11	693	16
Decoder	8	256	304	3
Int2Float converter	11	7	260	16
Priority encoder	128	8	978	250
Lookahead XY router	60	30	257	54
Voter	1001	1	13758	70

TABLE II
Results after converting the EPFL benchmarks to LiM-HDL

Name	MAJ nodes		Depth	
	before	after	before	after
Adder	1020	888	255	253
Barrel shifter	3336	2888	12	14
Divisor	57247	44991	4372	4437
Hypotenuse	214335	190331	24801	9498
Log2	32060	30285	444	456
Max	2865	2839	287	287
Multiplier	27062	25429	274	274
Sine	5416	5135	225	224
Square-root	24618	20402	5058	7035
Round-robin Arbiter	11839	11711	87	87
Coding-cavlc	693	586	16	18
Decoder	304	304	3	3
Int2Float converter	260	217	16	16
Priority encoder	978	654	250	187
Lookahead XY router	257	246	54	54
Voter	13758	9539	70	83
$\sum$	396048	346445	36224	22926

An algorithm for generating MIGs from AIGs has first been presented in [12]. An advanced implementation of the algorithm is available as part of the mockturtle[1] software library, which is part of the EPFL logic synthesis libraries. The algorithm works by resubstituting suitable nodes in AIGs and then performing optimizations by eliminating nodes and reshaping the MIG to find more elimination opportunities.

We modify a fork of the library to include support for exporting LiM-HDL code containing native RM_3 operations. Here, we replace original implementation which expands the MAJ-operation into its Boolean expression and compiled this modified fork into a library. Then, we have developed a conversion tool using this library, which reads an AIGER file, resubstitutes the nodes and produces a LiM-HDL file. The resubstitution process can be adjusted by various resubstitution parameters[2], but we use the default values.

[1] https://github.com/lsils/mockturtle

[2] https://mockturtle.readthedocs.io/en/latest/algorithms/resubstitution.html

TABLE III
COMPARISON OF DIFFERENT LEVELIZATION ALGORITHMS

Benchmark	**Branch&Bound** [11]			**MCTS** [11]			**Greedy** (proposed)		
	Area	**Delay**	**CT [s]**	**Area**	**Delay**	**CT [s]**	**Area**	**Delay**	**CT [s]**
Adder	2176	2748	0.038	2176	2814	13.595	2176	2748	0.014
Barrel Shifter	2912	3218	0.192	3008	3395	7.756	2928	3328	0.154
Divisor	TO	TO	TO	TO	TO	TO	54992	158608	241.059
Hypnotenuse	TO	TO	TO	TO	TO	TO	221888	542861	2354.24
Log2	TO	TO	TO	28464	57142	5230.53	31536	67611	44.977
Max	TO	TO	TO	3472	5018	20.290	3552	5245	0.175
Multiplier	TO	TO	TO	25152	43772	3251.06	28576	56209	26.448
Sine	TO	TO	TO	5232	10865	65.757	5552	12097	0.880
Square-root	TO	TO	TO	TO	TO	TO	37600	80716	29.140
Round-robin Arbiter	2416	13531	2.872	2544	13576	32.082	2416	13531	4.621
Coding-cavlc	TO	TO	TO	608	1319	4.6124	624	1311	0.006
Decoder	352	151	0.004	352	151	0.300	352	165	0.001
Int2Float converter	288	522	0.346	288	516	2.257	288	528	0.001
Priority encoder	TO	TO	TO	1184	2050	9.91392	1328	2231	0.008
Lookahead XY router	TO	TO	TO	400	710	3.360	384	786	0.002
Voter	TO	TO	TO	11696	35006	3142.23	15920	48335	7.552

2) Results: By applying our modified fork to the benchmarks from the EPFL Combinational Benchmark Suite, we produce the results shown in Table II. The optimized MIGs contain 12.52 % less MAJ nodes than the unoptimized ones and have their depth reduced by 36.70 % on average. While the amount of MAJ nodes is equal or reduced in all benchmarks, the amount of levels actually increased for 7 benchmarks compared to their naively converted counterparts.

B. Levelization

In this section, we evaluate our proposed levelization strategy. We compare our results to those of [11]. Here, we set the time limit to four hours.

The computation time, the resulting number of needed devices and delay for the computed levelization can be found in Table III. In the first column, the name of the benchmark is given. Then, the next columns show the results for the Branch&Bound and the MCTS-based methods proposed in [11]. The last three columns show the results for our proposed greedy method. For all levelization methods, the needed number of devices, the total delay (number of computations combined with the number of reads) of the final PLiM program and the *Computation Time* (CT) of the levelization itself in seconds is shown. As proposed in [11], we have used a time limit of 50ms for the exploration phase of the MCTS-based strategy.

We can see that the Branch&Bound-based method has a *TimeOut* (TO) for most benchmarks and consequently fails to compute a levelization and even the MCTS-based levelization strategy fails to compute a levelization for three benchmarks. Our proposed method is able to levelize all benchmarks in significantly less time compared to the state-of-the-art for all benchmarks, except for the Round-robin Arbiter. Here, the Branch&Bound-based method is faster. This is due to the fact, that the Branch&Bound-based method is heavily parallelized and an optimal branch is discovered very early, leading to an early termination. In terms of area and delay, our proposed heuristic can compete with the state-of-the-art. For three benchmarks, our results even outperform those of the MCTS-based method for both area and delay, while for all other benchmarks comparable results are achieved.

To conclude we can say, that our greedy algorithm computes comparable results in a significantly smaller time frame compared to the state-of-the-art.

TABLE IV
VERIFICATION TIMES

Benchmark	**Purity**		**Equivalence**	
	Gen	Proof	Gen	Proof
Adder	0.07	1.30	0.19	1.02
Barrel Shifter	0.13	5.90	0.39	3.02
Divisor	2.01	OM	5.67	52.51
Hypnotenuse	7.89	TO	23.08	225.04
Log2	1.27	OM	3.73	34.97
Max	0.16	5.82	0.41	3.15
Multiplier	0.99	TO	3.07	28.32
Sine	0.29	OM	0.62	5.53
Square-root	0.95	OM	2.49	21.05
Round-robin Arbiter	0.51	18.34	1.38	13.17
Coding-cavlc	0.06	0.66	0.10	0.56
Decoder	0.03	0.31	0.08	0.25
Int2Float converter	0.04	0.23	0.05	0.20
Priority encoder	0.06	0.89	0.12	0.74
Lookahead XY router	0.03	0.30	0.07	0.26
Voter	0.49	OM	1.32	11.00

C. Verification

In this section, we evaluate our proposed levelization strategy. For this, we have tried to verify the PLiM programs generated with our proposed greedy levelization method.

The results of our proposed verification strategy can be seen in Table IV. Here, for all benchmarks, the times for the two parts of our levelization strategy are given. First, the time needed for the *generation* (Gen) of the miter is shown. Then, the time needed for proving of the purity and equivalence is given (Proof). All times are shown in seconds. As SMT solver, we have utilized CVC4 [23] and we have again set the time limit to four hours. Note, that we have also tried Z3 and

MathSAT, however, CVC4 yielded the best results. It can be seen that in general proving the purity of a program is a lot harder than proving equivalence. Here, we either had a TO or ran *Out of Memory* (OM) for eight of the 20 benchmarks. For the remaining 12 benchmarks, where we succeeded to proove purity, the computation time is significantly larger compared to the proof of equivalence. The proof of equivalence has been successfully performed for all benachmarks.

This can yield interesting consequences. For example, one could skip the proof of purity, if all used cells are initialized at the beginning of a program, putting them in a defined state. This makes the verification of PLiM programs significantly easier.

VII. Conclusion

In-memory computing is a promising paradigm for future computation with ReRAM being a candidate as technological foundation. Based on this, the PLiM computer architecture has been proposed.

In this paper, first, we have shown how to verify a PLiM program for the PLiM computer architecture. Our verification strategy consists of two steps: First, we verify the purity of a program and then the equality to its HDL description. In addition, we have proposed a greedy levelization method for efficient generation of these programs. Finally, we have generated a valuable benchmark set, which can be used for future research in this field. In the experiments we have shown that our proposed levelization scheme is significantly faster compared to the state-of-the-art while yielding comparable results. We have shown that we can levelize large benchmarks, where the state-of-the-art fails. Additionally, using our proposed verification strategy, we observed that the proof of purity is significantly harder compared to the proof of equivalence. Consequently, in order to improve the verifiability of PLiM programs, we propose to initialize all used cells at the beginning of a program, making the proof of purity obsolete.

References

[1] J. von Neumann, "First draft of a report on the edvac," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[2] Y. Wang, H. Du, M. Xia, L. Ren, M. Xu, T. Xie, G. Gong, N. Xu, H. Yang, and Y. He, "Correction: A hybrid cpu-gpu accelerated framework for fast mapping of high-resolution human brain connectome," *PloS one*, vol. 8, p. e62789, 05 2013.

[3] C.-H. Chu, X. Lu, A. A. Awan, H. Subramoni, B. Elton, and D. K. Panda, "Exploiting hardware multicast and gpudirect rdma for efficient broadcast," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 575–588, 2019.

[4] E. A. Lee, B. Hartmann, J. Kubiatowicz, T. Simunic Rosing, J. Wawrzynek, D. Wessel, J. Rabaey, K. Pister, A. Sangiovanni-Vincentelli, S. A. Seshia, D. Blaauw, P. Dutta, K. Fu, C. Guestrin, B. Taskar, R. Jafari, D. Jones, V. Kumar, R. Mangharam, G. J. Pappas, R. M. Murray, and A. Rowe, "The swarm at the edge of the cloud," *IEEE Design Test*, vol. 31, no. 3, pp. 8–20, 2014.

[5] C.-X. Xue, J.-M. Hung, H.-Y. Kao, Y.-H. Huang, S.-P. Huang, F.-C. Chang, P. Chen, T.-W. Liu, C.-J. Jhang, C.-I. Su, W.-S. Khwa, C.-C. Lo, R.-S. Liu, C.-C. Hsieh, K.-T. Tang, Y.-D. Chih, T.-Y. J. Chang, and M.-F. Chang, "16.1 a 22nm 4mb 8b-precision reram computing-in-memory macro with 11.91 to 195.7tops/w for tiny ai edge devices," in *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 245–247.

[6] T. Mikawa, R. Yasuhara, K. Katayama, K. Kouno, T. Ono, R. Mochida, Y. Hayata, M. Nakayama, H. Suwa, Y. Gohou, and T. Kakiage, "Neuromorphic computing based on analog reram as low power solution for edge application," in *2019 IEEE 11th International Memory Workshop (IMW)*, 2019, pp. 1–4.

[7] 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 Conference Exhibition (DATE)*, 2016, pp. 427–432.

[8] S. Rai, M. Liu, A. Gebregiorgis, D. Bhattacharjee, K. Chakrabarty, S. Hamdioui, A. Chattopadhyay, J. Trommer, and A. Kumar, "Perspectives on emerging computation-in-memory paradigms," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1925–1934.

[9] D. Bhattacharjee, Y. Tavva, A. Easwaran, and A. Chattopadhyay, "Crossbar-constrained technology mapping for reram based in-memory computing," *IEEE Transactions on Computers*, vol. 69, no. 5, pp. 734–748, 2020.

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

[11] S. Froehlich and R. Drechsler, "Lim-hdl: Hdl-based synthesis for in-memory computing," in *Design, Automation and Test in Europe*, 2022.

[12] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[13] P. 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 and Test in Europe*, 2016, pp. 427–432.

[14] S. Frerix, S. Shirinzadeh, S. Froehlich, and R. Drechsler, "Comprime: A compiler for parallel and scalable reram-based in-memory computing," in *2019 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, 2019, pp. 1–6.

[15] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71. New York, NY, USA: Association for Computing Machinery, 1971, pp. 151–158. [Online]. Available: https://doi.org/10.1145/800157.805047

[16] N. Bjorner, "Smt solvers for testing, program analysis and verification at microsoft," in *2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2009, pp. 15–15.

[17] C. Barrett, P. Fontaine, and C. Tinelli, "The satisfiability modulo theories library (smt-lib)," 2016. [Online]. Available: https://smtlib.cs.uiowa.edu/

[18] ——, "The smt-lib standard: Version 2.6," Department of Computer Science, The University of Iowa, Tech. Rep., 2017. [Online]. Available: https://smtlib.cs.uiowa.edu/

[19] M. Golmohamadi, R. Jurasek, W. Hokenmaier, D. Labrecque, R. Zhi, B. Dale, N. Islam, D. Kinney, and A. Johnson, "Verification and testing considerations of an in-memory ai chip," in *2020 IEEE 29th North Atlantic Test Workshop (NATW)*, 2020, pp. 1–6.

[20] E. Kondratyuk, Y. Matveyev, A. Chouprik, E. Gornev, M. Zhuk, R. Kirtaev, A. Shadrin, and D. Negrov, "Automated testing algorithm for the improvement of 1t1r reram endurance," *IEEE Transactions on Electron Devices*, vol. 68, no. 10, pp. 4891–4896, 2021.

[21] D. D. Antoniadis, P. Feng, A. Mifsud, and T. G. Constandinou, "Open-source memory compiler for automatic rram generation and verification," in *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2021, pp. 97–100.

[22] L. Amarú, P. E. Gaillardon, and G. D. Micheli, "The epfl combinational benchmark suite," in *International Workshop on Logic & Synthesis*, 2015.

[23] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 171–177.