LiM-HDL: HDL-Based Synthesis 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*—HDLs are widely used in EDA for abstract specification and synthesis of logic circuits. Despite the popularity and the many benefits of HDL-based synthesis, it has not yet been performed for in-memory computing. Hence, there is a need to design a particular HDL which supplies efficient and compatible descriptions.**

In this paper, we enable HDL-based synthesis for the *Programmable Logic-in-Memory* (PLiM) computer architecture. We present LiM-HDL - a Verilog-based HDL - which allows for the detailed description of programs for in-memory computation. Having the description given in LiM-HDL, we propose a synthesis scheme which translates the description into PLiM programs, i.e. a sequence of resistive majority operations. This includes lexical and syntax analysis as well as preprocessing, custom levelization and a compiler.

In our experiments, we show the benefits of LiM-HDL compared to classical Verilog-based synthesis. We show in a case-study that LiM-HDL can be used to implement programs with respect to constraints of specific applications such as edge computing in IoT, for which the PLiM computer is of particular interest and where low area is a key requirement. In our case-study, we show that we can reduce the number of ReRAM devices needed for the computation of an encryption module by 69%.

I. Introduction

As the integration of *Internet of Things* (IoT) in today's society is progressing, there is a shift from computation in data centers to edge devices [1]. More and more edge devices are used, which are at the same time of decreasing size and very restricted in terms of area, computation capabilities and admissible power consumption [2].

Most of today's computer systems are based on the von Neumann computer architecture. This architecture suffers from the von Neumann bottleneck which describes the limited transfer rate of the data from the memory to the CPU [3].

Recently, *Resistive Random Access Memory* (ReRAM), a resistance based storage device, is emerging. ReRAM is especially appealing due to its inherent in-memory computation capabilities. ReRAM allows for the computation of universal functions and thus, ReRAM can compute any Boolean function. ReRAMs low power consumption, scalability and fast switching capabilities make it an excellent candidate for a technological foundation for edge devices and IoT [4], [5].

In order to overcome the von Neumann bottleneck, an architecture for the *Programmable Logic-in-Memory* (PLiM) computer has been proposed [6]. In addition to the control logic, the core of the PLiM computer architecture are the ReRAM arrays, which are used as storage and computational unit. 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 [6]. In [7], the PLiM computer architecture has been improved to support computation of multiple operations in parallel.

Today, state-of-the-art synthesis is performed using *Hardware Description Languages* (HDLs) such as Verilog and VHDL. These languages allow for the definition of hardware modules which use resources of conventional computational devices such as registers etc. However, these languages feature no specific properties of the PLiM computer architecture, which are especially efficient for the computation of specific logic operations such as MAJ, AND and OR. Existing HDLs are used to define structures such as gates, multiplexers and flip-flops while a program for the PLiM computer is executed on a crossbar array and consists of a sequence of signals applied to the rows and columns without changing the way the devices are wired.

In this paper we propose *Logic-in-Memory* (LiM)-HDL, a HDL tailored for in-memory computing and the PLiM computer architecture. LiM-HDL is a Verilog-based definition language and features operations which can be efficiently synthesized in ReRAM. As LiM-HDL is Verilog-based, most of its syntax is very similar to that of conventional Verilog, allowing for easy translation of an existing code basis. Together with a compiler which optimizes the final implementation, LiM-HDL allows to synthesize hardware very efficiently into programs for the PLiM computer architecture. Additionally, LiM-HDL can be combined with conventional Verilog to allow for easy integration into existing systems and a low initial hurdle to entry.

In addition to LiM-HDL, we propose a synthesis scheme, which can be used to compile LiM-HDL. The synthesis scheme features preprocessing, levelization and a compiler for the LiM-HDL code. Finally, we propose the syntax for the final PLiM program, which can be executed on the PLiM computer architecture.

II. Preliminaries

A. Storing Values

ReRAM devices are ordered in crossbar arrays, which consist of multiple bitlines and wordlines. Each device is connected to one bitline and one wordline. By applying the voltage $V/2$ to the corresponding bitline and the voltage $-V/2$ to the corresponding wordline, the device which is connected to that bitline and that wordline can be put into a high resistance state. Similarly, by applying $-V/2$ to the bitline

This work was supported by the German Research Foundation (DFG) within the Project *PLiM* (DR 287/35-1).

and $V/2$ to the wordline, the device can be put into a low resistance state. The voltages $V/2$ and $-V/2$ and the low resistance and high resistance state can be interpreted a logic 1 and logic 0, respectively.

B. *Resistive Majority Operation* RM_3

Each ReRAM device can be interpreted as a two-terminal device with the terminals P (the *wordline operand*) and Q (the *bitline operand*) and the internal resistance state Z (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 $\mathrm{RM}_3(P, Q, Z) = MAJ(P, \bar{Q}, Z)$.

III. Related Work

In this section, related work is introduced. As we are the first to propose a HDL tailored for in-memory computing, we focus on other compilation strategies in this section.

The authors of [8] present a *Majority-Inverter-Graph* (MIG)-based compiler for the PLiM computer architecture. The authors use rewriting techniques and a translation algorithm for the optimization of MIGs. However, as the PLiM computer architecture introduced in [8] only supports single instructions, this work does not utilize parallelization and focuses on a different architecure.

In [7], the authors propose the AIG-based compiler ComPRIMe for the PLiM computer architecture. ComPRIMe supports parallel computation within regular ReRAM crossbar arrays. First, a given AIG is levelized. Each level is then computed in parallel by exploiting the fact that computation of operations can be parallelized if each operation shares the same wordline operand.

IV. LiM-HDL

In this section, we describe the syntax of our proposed LiM-HDL in detail. Besides RTL descriptions, LiM-HDL also gives support for descriptions at behavioral level. The described syntax can be integrated into lexers and parsers to parse files written in LiM-HDL.

As LiM-HDL is fully integrable into existing Verilog code and the syntax is very similar, the compiler has to be instructed to interpret code portions as LiM-HDL. In order to do this, code blocks can be framed by the instructions

```
LiMHDLbegin
LiMHDLend
```

A. *Logic Operations*

For the description of arbitrary hardware, it is imperative to allow for basic logic operations. Since the RM_3 operation is native to ReRAM, other logic operations are translated into equivalent RM_3 expressions. LiM-HDL supports the following operations, where the parameter WL is the wordline operand, the parameter BL is the bitline operand and the parameter H is the host operand:

- RM_3: As the RM_3 operation is native to ReRAM, it is essential to an HDL for in-memory computing. Any RM_3 operation is directly translated into a single instruction. The syntax is defined as:
  `assign out = RM3(WL, BL, H)`
- &: As proposed in [7], the & operation can be directly translated into an RM_3 operation. The syntax is defined as:
  `assign out = BL&H`
  The & operation is translated into $\mathrm{RM}_3(0, \bar{BL}, H)$
- |: Similar to &, the | operation can be directly translated into an RM_3 operation. The syntax is defined as:
  `assign out = BL|H`
  The | operation is translated into $\mathrm{RM}_3(1, \bar{BL}, H)$
- ~: As inversions are computed implicitly, the ~ operation is not translated into a RM_3 operation. However, it is still admissable to use within LiM-HDL:
  `assign out = ~OP`
- ^: The XOR operation ^ can not be translated into a single RM_3 operation, but needs three RM_3 operations to be computed. The LiM-HDL syntax is defined as:
  `assign out = OP1^OP2`
  As this operation can not be translated into a single RM_3 operation, it is translated into:
  $$\mathrm{RM}_3(1, \mathrm{RM}_3(0, OP1, OP2), \mathrm{RM}_3(0, OP2, OP1))$$
  We choose this representation since it maps to AND and OR operations and allows for parallel execution with other AND and OR instructions.

In addition, shift operations are supported. However, in LiM-HDL they are not directly translated into operations as they have a semantic meaning, i.e. as to connect which operand to which operation in the final graph.

B. *Behavior Level Modeling*

LiM-HDL offers support for modeling at behavioral level. PLiM programs consist of a sequence of signals applied to the rows and columns of crossbars. Thus, in LiM-HDL only blocking assignments are supported. However, LiM-HDL does feature full support for-loops and conditional statements. In this section, we detail the supported statements.

To allow for easy integration, the syntax is equivalent to that of Verilog. Thus, behavior level modeling is indicated by always blocks, which can be used within LiM-HDL. LiM-HDL supports the three loop instructions *for-loops*, *while-loops* and *repeat-until-loops*. Besides Loops, LiM-HDL also features the conditional *if-statements* and *switch-case-statements*. Again, the same syntax as for Verilog is adapted.

Example 1. An example is given in Listing 1. Here, we have defined a full *Ripple-Carry Adder* (RCA). As we can use always blocks with for-loops we can easily create the full-adder tree, which is characteristic for RCAs. The output of a full-adder is computed by two XOR operations. However, using the substitution proposed in [9], it can be shortened as in Line 15 resulting in only three RM_3 operations. Further, we have optimized the computation of the carry bits in Line 16.

Note that such an optimization would not be possible without the expressiveness of LiM-HDL.

Listing 1. 8-Bit RCA

```
1   module Adder(IN1, IN2, cin, out);
2    parameter width = 8;
3    input[width-1:0] IN1;
4    input[width-1:0] IN2;
5    input cin;
6    output reg [width:0] out;
7    reg [width:0] c;
8    integer i;
9    LiMHDLbegin
10    always @ (IN1 or IN2)
11    begin
12     c[0]=cin;
13     for ( i=0; i<width; i=i+1)
14     begin
15      out[i]= RM3(RM3(IN2[i],IN1[i],c[i]), IN2[i], RM3(IN2[i],c[i],IN1[i]));
16      c[i+1]= RM3(IN2[i],~IN1[i],c[i]);
17     end
18     out[width]=c[i];
19    end
20   LiMHDLend
21  endmodule
```

To allow for easy implementation of more complex modules, we have implemented the $+$ and $-$ operations in LiM-HDL to be substituted by the RCA given in Listing 1. Thus, the code shown in Listing 2 generates the same PLiM program as that of Listing 1, if we set the carry input cin to 0.

Listing 2. 8-Bit RCA using the $+$ operator

```
1   module Adder(IN1, IN2, out);
2       parameter width = 8;
3       input[width-1:0] IN1;
4       input[width-1:0] IN2;
5       output reg [width:0] out;
6       LiMHDLbegin
7         out=IN1+IN2;
8       LiMHDLend
9   endmodule
```

C. Generate Statements

For easy assembly of complex hardware using hardware components as building blocks, LiM-HDL supports generate statements. Here, also the same loops and conditional-statements as in the behavioral level modeling are supported.

V. Synthesis of LiM-HDL-based Circuits

After implementing LiM-HDL into the lexer and parser, it can be transformed into an abstract syntax tree and subsequently into an RTL graph representation [10]. This RTL graph representation consists of cells, which represent the different operations in the program. In this section, we present a synthesis scheme which can be used to preprocess the RTL graph and synthesize it into a PLiM program. Our synthesis scheme is based on [7], however, while [7] is limited to AIGs, we propose a generalization which is capable of processing arbitrary RM_3-based RTL graphs. Figure 1 shows the flow of our proposed scheme. It can be divided into a preprocessing step, a levelization step and the final compilation.

A. Preprocessing

Before levelizing and compiling the RTL graph to a PLiM program, we perform three preprocessing steps:

1) Mapping to permitted cells: Before compilation, some cells need to be replaced, removed or optimized in order to allow for the compiler to transform the RTL graph into a PLiM program. As we aim to compile AND, NOT and RM_3-based graphs, other cells such as XOR, OR and the $+$ operation,

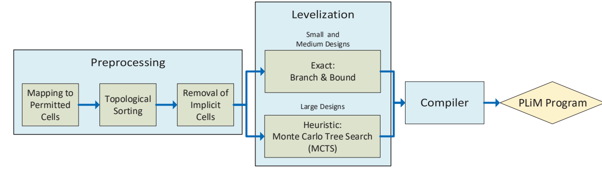

Fig. 1. Proposed Synthesis Scheme

which may have been introduced due to the use of Verilog or behavioral level descriptions, need to be replaced. First, the $+$ and $-$ operations are replaced by their corresponding RM_3-based representation (see Section IV-B). Further, all cells which do not represent the AND, NOT or RM_3 operations are replaced and optimized to AIG.

2) Topological Sorting: In order to compute the cells in the right order, the cells are sorted topologically. This step ensures that all inputs needed for a cell are already computed when performing compilation.

3) Removal of Implicit Cells: Some cells are obsolete for in-memory computing. Synthesis tools generate sequential elements, however, as the basic concept of in-memory computing is not timing reliant, elements which are used for synchronization can be ignored. Additionally, as our synthesis algorithm computes inversions implicitly, NOT operations can be removed from the design. Instead of NOT operations, each cell stores if it needs its inputs to be inverted or not. As inversions are computed implicitly, the corresponding values can be loaded from the memory or computed where needed and applied in the final PLiM program.

B. Levelization

Levelization is an important step for parallel computation in in-memory computing, as a small number of levels reduces the delay and also impacts the area of the final PLiM program. The RTL graph is levelized in such a way, that all cells which can be computed, if the previous level is computed, are placed on one level. This is sufficient, if no further restrictions are given and thus used in the synthesis scheme of [7], as all cells share the same wordline operand.

However, as only cells which share the same wordline operand can be computed in parallel, the complexity of the problem increases for our synthesis scheme. Note that this constraint is also not taken into account by existinig levelization schemes for MIGs and AIGs. To meet these constraints, we propose two approaches. The first approach gives an exact levelization based on *Branch & Bound* (B&B) (i.e. a levelization with the least number of levels - note that multiple levelizations with the same number of levels may exist), while the second approach is a heuristic levelization based on the *Monte Carlo Tree Search* (MCTS) algorithm.

In the following, we call a cell computable, if all its inputs are either computed in a previous level or are constant or a primary input.

1) Exact Levelization using Branch & Bound: The first approach for the levelization is an exact algorithm: We use a

B&B algorithm to find the best levelization. As this approach can be parallelized in multiple threads, it is very efficient.

The B&B algorithm performs an exploration of the search space in a depth-first manner. During the exploration, consecutively better upper bounds for the final result are calculated, while lower bounds for the current search branch are determined. If the lower bound of the current search branch is exceeding the upper bound for the final result, the current branch can be cut, i.e. the current branch is no further explored.

First, when performing the B&B algorithm, one possible levelization is computed by choosing a valid computation order at random. Consecutively, we compute new levels by adding computable cells which share a wordline operand with the other cells on the new level. This way, we can add levels to the levelization until no cells remain. Finally, the number of levels of this levelization is used as upper bound and is improved on during the execution of the B&B algorithm. Whenever a better levelization is found, its number of levels is used as new upper bound.

After the computation of this upper bound, the search space is explored. As lower bound for the final number of levels in the current branch, we can use the number of already computed levels plus the number of different wordline operands for each cell which has not yet been added to the levels, as this is the minimum number of total levels the final levelization will have.

2) Heuristic Levelization using Monte Carlo Tree Search: For more complex designs, we propose to use MCTS [11], [12] as a heuristic method.

Unlike B&B, instead of exploring the search space in a depth-first manner, MCTS explores different possible next levels and picks the most promising one. MCTS can be divided into four steps, which are repeatedly executed until all cells are assigned to a level. The *state* contains the current levelization (i.e., cells which have already been assigned to a level and their respective level) and cells, which are not yet assigned.

1) *Selection*: From the current *state* select a possible next levelization, which has not yet been explored.
2) *Expansion*: If this levelization is only intermediate, i.e., it does not include all cells, create one or more possible next levels and select one of them.
3) *Simulation*: Compute a random levelization starting from the selected levelization, which includes all cells.
4) *Backpropagation*: Use the random levelization to update the score for the intermediate levelizations on the path from the current *state* to the selected levelization.

These four steps are repeated until a time limit is reached. Subsequently, the most promising next level is added to the current *state*.

C. Compiler

After preprocessing and levelization, the final RTL graph only consists of AND and RM_3 operations, for which is stored which inputs are to be inverted. As the AND operation of two inputs a and b can be represented as $\mathrm{RM}_3(0, \sim a, b)$, we only need to describe how to synthesize RM_3-based graphs.

1) Enabling Parllelism: In order for two cells to be computed in parallel, the following conditions have to be met:

1) All child cells of both cells must have already been computed.
2) The cells must share a wordline operand.
3) The cells have to be placed on the same wordline.

The first and second condition are already met due to the topological sorting and the levelization.

However, the third condition needs to be met by the compiler. For the compiler, it is imperative to not override devices that contain information which may be needed for future computations. Thus it needs to keep track of whether the information stored in a devices is still needed.

The number of free devices in a word is called a $hole$ and we use a parameter called *hole size* h to determine if a hole can an be used for new computations or not. This increases the parallelization of the computation: If the parameter hole size h is too small, levels of the levelized RTL graph will be distributed among multiple wordlines which will ultimately lead to a reduction in the parallelization of the computation of the graph. However, if the hole size h is too large the compilation may be inefficient and the number of devices needed will grow.

2) Synthesis Scheme: Our compiler synthesizes the RM_3-based graph level by level into a final PLiM program. In order to compute cells in parallel, first their host operands are loaded into the devices. As we use inversion for initialization, it is beneficial if the host operands are inverted. In addition, as these values need to be read from the ReRAM array, it is also beneficial, if all host operands reside on the same wordline. Thus, during compilation, we split the levels into groups, where each group consists of the cells for which the host operands reside on the same wordline. Subsequently, we try to fill the holes with these groups. Finally, the wordline operand and bitline operands are applied to the corresponding wordline and the bitlines.

D. Syntax for Parallel Computing

We now define the output format of the compiler, i.e., the format of the final PLiM program which is to be executed on the PLiM computer.

As the inputs to the PLiM program need to be placed somewhere on ReRAM array, the first lines of the program show where the inputs are expected to be placed. If the inputs should be placed anywhere else, either the program has to be changed or the inputs have to be copied to the respective devices first.

Listing 3. Example Input Placement

```
0 4 a 5 b 6 c
```

Listing 3 shows an example, where we expect the inputs a, b and c to be placed on the devices 4, 5 and 6 of the first wordline. The first number denotes the number of the wordline, followed by a list of device numbers and their corresponding content.

Every instruction consists of the wordline number, the address of the wordline operand and a list of addresses of bitline numbers and bitline operands. Note that like in the original PLiM computer architecture [6], each single device needs to have a unique address. If an operand is constant, the values TRUE or FALSE are given in the instruction.

Listing 4. Example Instruction

```
0 0xe 0 0x1b0 1 0x9c 2 FALSE 3 TRUE 8 0x9a
```

An example is given in Listing 4. Here, the content of the device with the address *0xe* is applied as wordline operand to the wordline 0. In addition, the contents of the devices *0x1b0*, *0x9c* and *0x9a* are applied to the bitlines *0*, *1* and *8*, respectively. If an input is constant, it can be directly applied to the wordlines and bitlines. Here, this is done for the second and third bitline.

Finally, the last lines of the PLiM program show the addresses of the primary outputs. This allows to locate the outputs without having to analyze the whole program.

Listing 5. Example Output Placement

```
out[0]: 0x15
out[1]: 0x11
out[2]: 0x10
```

Listing 5 shows an example where the output values *out[0]*, *out[1]* and *out[2]* are placed at the addresses 0x15, 0x11 and 0x10, respectively.

VI. Experimental Results

In this section, we describe the experiments performed to evaluate the benefits of LiM-HDL, in addition to being an easy to use HDL. All experiments have been executed on an Intel® Xeon® CPU E5-2630 v3 @ 2.40GHz with 64GB memory running Linux (Fedora release 30). We have assumed a wordsize of 16 and used 12 for the parameter holesize *h*. Note that, as we focus on the reduction of the area, we have merged the number of reads and computations into delay in our results for clarity. However, the usage of LiM-HDL often influences the number of computations more than the number of reads (often by a factor of about 2). In addition, it should be noted that the AIGs are heavily optimized by Yosys, while no complex optimization procedures have been implemented for the RM_3-based graphs used in LiM-HDL, yet.

A. Implementation Details

We have implemented an extended version of the Yosys synthesis tool [10]. Here, we have implemented a pass called $PLiMmap$ into Yosys V0.9+3696 to perform the first step of the preprocessing. As future work may introduce optimizations for the mapped RTL graph, we perform the last two steps and the levelization only during the compilation command *write_PLiM*, which also performs the final compilation. All experiments were performed using the Yosys command sequence $proc, flatten, opt, PLiMmap, opt, write_PLiM$. Note that the command PLiMmap performs mapping to AIG, if LiM-HDL is not used.

TABLE I
Results for Arithmetic Circuits

	AIG [7]		LiM-HDL		Reduction	
Name	Area	Delay	Area	Delay	Area	Delay
RCA_16	352	446	167	332	52.6%	25.6%
RCA_64	1376	1763	672	1328	51.2%	24.7%
RCA_512	10784	14058	5312	10624	50.7%	24.4%
RCA_1024	21536	28107	10592	21248	50.8%	24.4%
MUL_16	3136	2401	1120	6384	64.3%	-165.9%
MUL_32	11808	8714	4288	26341	63.7%	-202.3%
MUL_64	44640	33860	16736	106828	62.5%	-215.5%

B. Arithmetic Circuits

In order to evaluate the advantages offered by LiM-HDL, we have implemented a RCA. Using this RCA and shift operations, we have further implemented a multiplier in LiM-HDL and compared it to its AIG counterpart, where the main difference is the implementation of the $+$ operator. For compilation of the AIGs, we use the state-of-the art method presented in [7]. We use the B&B algorithm for levelization.

The results can be seen in Table I. The first column shows the name of the circuit, the second and third column show the number of used devices and delay (the total number of computations and reads) for the AIG implementation compiled with the methodology presented in [7]. The fourth and fifth column show the number of used devices and delay for our LiM-HDL-based implementation. The last two columns show the reduction of the number of used devices and delay of the LiM-HDL-based implementation compared to the AIG-based implementation. Here, positive numbers denote a reduction, while negative numbers indicate an increase. The first four rows show the results for a RCA with 16bit/64bit/512bit/1024bit wide inputs, respectivly. The next three rows show the results for the multipliers with 16bit/32bit/64bit wide inputs. All experiments have been performed within 90 seconds.

It can be seen, that for the adders our efficient implementation allows to significantly reduce the number of used devices by up to 52.6% and the delay by up to 25.6%. This is due to the use of RM_3 operations and the corresponding optimizations in the LiM-HDL implementation detailed in Section IV.

For the multiplier we can see that the reduction in used devices is even larger than that of the adder with up to 64.3%. However, this time the delay is increased by up to 215.5%. This is caused by the parallelization scheme: As the implementation of [7] is AIG-based, it allows for the parallel computation of all instructions given that their inputs are already computed, since they all share the same wordline operand.

C. Case Study: Using LiM-HDL to Implement PRESENT

In this section, we conduct a case study to test LiM-HDL for its applicability in a real-world scenario with focus on edge devices. As [6] suggests, the PRESENT block cipher [13] is a meaningful application for in-memory computing and specially the PLiM computer architecture. In addition, as the PRESENT block cipher is light weight, it is especially designed for deployment on tiny devices [13] and for com-

TABLE II
RESULTS FOR ENCYPTION MODULE

	AIG [7]		LiM-HDL		Reduction	
Name	Area	Delay	Area	Delay	Area	Delay
AddRoundKey	384	92	384	116	0.0%	-26.1%
sBoxLayer	624	524	496	1196	20.5%	-128.2%
pLayer	64	4	64	4	0.0%	0.0%
KeyUpdate	192	105	176	100	8.3%	4.8%
Key Copy	80	5	80	5	0.0%	0.0%
Cipher Copy	64	4	64	4	0.0%	0.0%
Encryp. Module	29696	29445	9200	56837	69.0%	-93.0%

putation on the edge. Thus, we present the results for the implementation of the encryption module using LiM-HDL.

1) Background: The PRESENT block cipher has a block length of 64 bits and supports keys with a length of 80 and 128 bit. As in [6], we focus on the 80-bit key length in this paper. The PRESENT cipher uses 31 rounds for the encryption of the STATE, where each round consists of an XOR operation to introduce round keys, a non-linear substitution layer and a linear permutation layer. Due to page limitations, we can not give a description of the modules here. However, detailed information on the PRESENT block cipher can be found in [13].

2) Implementation: As LiM-HDL is a HDL, unlike [6], we don't need to implement every operation applied to the ReRAM array in detail, but can make use of the ample possibilities, LiM-HDL has to offer:

First, we implement the operations AddRoundKey, sBoxLayer and pLayer. For AddRoundKey, we can use the XOR operation and since pLayer is a permutation, this can be easily implemented as well. However, the sBoxLayer requires a sophisticated approach as it makes up for a large portion of the total area [13]. For an efficient AIG implementation, we use the algorithm of Quine and McCluskey to find a good minimal polynomial for each output of the used S-box operation. For a good LiM-HDL representation, we first transform these minimal polynomials to an MIG using CirKit [14] and subsequently implement this MIG in terms of RM_3 operations in LiM-HDL.

We have used B&B for the levelization of the Key Copy, Cipher Copy, AddroundKey, pLayer and KeyUpdate procedures. However, due to their complexity, we have used MCTS to levelize the sBoxLayer and the final encryption module. Here, we set the time limit to one second. As MCTS is a heuristic, we have run it three times for both the sBoxLayer and the encryption module and choose the best result out of these three runs.

3) Results: The results can be seen in Table II. The first six rows show the results for the different parts of the encryption module, while the last row shows the results for the complete implementation of the encryption module in a PLiM program. Note that, while not shown here, both approaches significantly outperform the results of [6].

Since Key Copy and Cipher Copy are copy operations, the complexity in terms of area and delay is equivalent for both synthesis algorithms. As the pLayer performs permutations, again, the complexity remains equivalent. However, our approach provides significant reductions in terms of area for the sBoxLayer and the Key Update procedure. Specially, the size of sBoxLayer, which occupies a large portion of the overall area of the encryption module, can be reduced by 20.5%. This results in a reduction of 69% of the area for the final implementation of the complete encryption module, due to a more efficient final levelization, as can be seen in the last row of the table. Note that this comes at the cost of increased delay, however, we assume that the area is more critical for encryption on edge devices.

VII. CONCLUSION

In this paper we have introduced the language LiM-HDL. LiM-HDL can be used to describe PLiM-programs in HDL code. Additionally, we have defined the syntax of PLiM-programs which includes instructions for the parallelized PLiM computer architecture presented in [7]. We have presented a methodology to compile the proposed LiM-HDL into PLiM-programs including a preprocessing step and we have designed custom levelization algorithms.

In our experiments, we have shown that using LiM-HDL, we can write programs which need significantly less devices and thus area compared to the state-of-the art. For arithmetic circuits, we could reduce the area by up to 64.3% and in our case-study, we have shown that we can reduce the size of an encryption module of the block cipher PRESENT by up to 69%. This is of importance for edge devices for IoT, which play an important role in today's society and where low area is a key requirement.

REFERENCES

[1] 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.

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[3] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st Conference on Computing Frontiers*, ser. CF '04, 2004, pp. 162–167.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] M. Soeken, P. Gaillardon, and G. De Micheli, "Rm3 based logic synthesis (special session paper)," in *IEEE International Symposium on Circuits and Systems*, 2017, pp. 1–4.

[10] C. Wolf, "Yosys - yosys open synthesis suite." [Online]. Available: http://www.clifford.at/yosys/about.html

[11] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," ser. CG'06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 72–83.

[12] T. Pepels and M. H. M. Winands, "Enhancements for monte-carlo tree search in ms pac-man," in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, 2012, pp. 265–272.

[13] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *Proceedings of the 9th International Workshop on Cryptographic Hardware and Embedded Systems*, ser. CHES '07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 450–466.

[14] M. Soeken, "Cirkit." [Online]. Available: https://github.com/msoeken/cirkit