Exploration of fine-grained self-healing concepts for approximate multipliers

Peters, Quinty C.

Award date:
2022

Exploration of fine-grained self-healing concepts for approximate multipliers

Q. C. Peters - q.c.peters@student.tue.nl - 0951927

***Abstract*—Approximate computing is a known method to save energy consumption in hardware. Approximate computing is trading accuracy of calculations in for decrease of energy consumption. Self-healing is a special method of creating approximate hardware. It is error correction for approximate computing, where computing elements are combined in such way that errors are canceled and the final error is minimized. In this research, methods of fine-grained self-healing are investigated for the array multiplier. Fine-grained refers to the detailed granularity of adapting the computing hardware. In this case, adaptations are made in the truth tables of full adder cells. The goal is to improve upon the state-of-the-art by finding an automated and generic method to create designs faster with less accuracy loss than when no self-healing is applied. Current self-healing methods barely include automation and little design space exploration. The main challenge is the trade-off between design time and the quality of the designs. Two fine-grained methods are tested. The first method partitions the multiplier in blocks and choices are made regarding the creation of those blocks and how to substitute them. The second method is choosing what each separate cell in the multiplier should be based on a cell's error using backtracking and pruning. The methods are tested on a 3x3 array multiplier. The found designs along with the run time show that there is no hard and consistent evidence that self-healing on this fine granularity is scalable in the long run. Recommended is to find a method which combines the non-automated state-of-the-art with the automated design space exploration used in this research. This way, more of the design space is explored than currently has been, but with a higher chance of optimizing the creation of approximate circuits. It is also recommended not to maximize the genericness of the methodology as the knowledge of the arithmetic unit can be essential to increase the quality of the found designs.**

***Keywords*—** approximate computing, self-healing, design space exploration, heuristic

I. Introduction

Technology has developed much over the last years. For example, mobile phones are not used for only calling anymore. Playing games, video processing, and navigation amongst others are tasks that are integrated in these small portable devices as well. All these applications on such small devices require hardware doing many computations. Even on bigger devices, such as a desktop, we want to do be able to do more complex tasks. Neural networks are becoming more advanced and are able to perform more complicated tasks such as more advanced recognition or prediction. All the calculations necessary for these tasks require much energy, which might limit the possibilities hardware-wise. Taking into considering what future applications might hold, it is desired to look for a method to limit energy consumption without compromising quality.

A well-known method to decrease hardware energy consumption is approximate computing. Just as it is for humans, it is less time and energy consuming for hardware to do simpler calculations with less precision. In other words, the outcome of calculations is not fully accurate. This can be done in several ways and on different levels of abstraction. Approximate computing involves a trade-off between three factors: accuracy, latency, and energy consumption. Accuracy is lower with improved latency and energy consumption. Approximate computing is a well-explored method for energy savings. [1] and [2] both discuss extensively multiple techniques to design approximate circuits, on different levels of abstraction even.

Using inaccurate answers without compromising the quality is used in neural networks and image processing amongst others. If an application is used to classify images to detect if a cat or dog is depicted on the image, it does not need to know the full-precision outcome of an calculation. If the calculation outcome is higher than a certain threshold value, the application categorizes the image as a dog and a cat otherwise. This way, both time and energy consumption is improved when making use of approximate computing.

A special form of approximate computing is self-healing. Approximate computing results in computation outputs which are not accurate. If the error of such calculation is known, one can make use of that and possibly correct that error with other approximate outcomes. In other words, when multiple approximate calculations are performed and their outcomes are accumulated, it might be possible to have a close-to-accurate outcome when those approximate outcomes could cancel each other out. Also with self-healing, there are multiple ways to achieve this effect on different levels of abstraction. Self-healing is merely a variant of using approximate computing, a special way to apply the concept of approximate computing. Using the self-healing ideology, it might be possible to create approximate hardware with less accuracy loss than conventional approximate computing hardware. Even though self-healing is less matured than conventional approximate computing, working methods are already developed such as [3], [4].

The relevance of the self-healing method is similar to that of approximate computing. Using a special technique to create approximate hardware, we obtain hardware which is faster and less energy-consuming, but now with the additional advantage of higher accuracy. Because in a datapath errors might accumulate too much when making use of approximate computing only.

The state-of-the-art offers self-healing solutions which do not include exploration of a large design space. The solutions require manual effort and little to no automation. Additionally, the methods are not generic. Hence, the solutions may only work for specific structures. The methods are not generic because adaptations were made on a high level (coarse-grained). In this research, the focus lies on changing bits in truth tables which is low-level (fine-grained).

The purpose of this research is to investigate several methods of applying the self-heal concept to design approximate computing hardware. This paper presents the following main contributions.
1) Exploring fine-grained self-healing methods, based on changes made on the truth tables. Since changes are made on a fine granularity, the methods could be generic and be used for multiple structures.
2) Automation of exploring the design space. Current methods require much manual effort. In this research, automation is key to explore a large design space. Limits must be set automatically as well, to prevent the design space from exploding.

The paper is structured as follows. Firstly, the state-of-the-art is explained in section II. This section covers the existing approximate and self-healing concepts in more detail. Secondly, required background knowledge is given in section III. Thirdly, the methodology is explained in section IV. In this section, it is explained what the reference of this research is and how each of the investigated self-healing methods work including their benefits and shortcomings.

Fourthly, the results of each methods are shown in section V. Further recommendations for future research on this topic are given in section VI. Lastly, there is a summary and conclusion in section VII.

II. Related work

Conventional approximate computing adapts circuitry to return inaccurate outcomes without trying to compensate for those errors further in the datapath. An elaborate survey on different methods of approximate computing are [1] and [2]. Examples of conventional approximate computing are [5], [6]. Approximate computing is already quite matured. There are even tools to run the state-of-the-art approximate techniques such as [7] to compare all existing methods.

As mentioned in the introduction, there are different levels of abstraction which are explained in these surveys as well. Adaptions can be made on software level, architecture level, and low level circuit level. The latter is the focus of this research. In particular, the focus lies on adapting truth tables, which is called the fine-grained approach.

Examples of fine grained approximating are [8]–[10]. In these researches, alterations were made of the full adder truth table. In the first case, a few full adders were available since the creation of the approximate full adders was done manually by looking the transistor schematic of the full adder. In [9], firstly non-pareto possible full adders cells were filtered out regarding energy and accuracy. Afterwards, it was manually checked which cells were the most usable for that research. In [10] automation of this process was applied, increasing the design space and decreasing manual effort. Both researches show decrease in energy consumption without severe loss of quality in image processing and neural network applications. The research in this paper searches for an automated method and hence, the approach of [10] is used and further explored using the self-healing concept.

Unfortunately, state-of-the-art self-healing is coarse-grained. In this research, it means that the changes made to approximate circuitry is done on a higher level than truth tables. Hence, it is difficult to combine the ideology of [8]–[10] with the current methods. Examples of these coarse-grained self-healing solutions are [3], [11]–[14] where either error signals or error correction modules are used to heal some of the errors, which causes extra overhead.

To heal errors, errors must be tracked or known somehow. [3], [11] do this by using error signals and choosing to change certain operations in the following arithmetic units depending on the error signals. It increases the accuracy with the loss of gaining overhead. Similar is [12]. It also uses the known error, but only creates one correction module to fix the errors in adders, saving overhead. The research is tested for some commonly used approximate adders and the correction is based on the fact that the error can only be certain specific values in these adders. [14] shows that using using error correction this way does not always end up in favour of the healing solution compared to non-healing approximate variant and that much tweaking and parameterizing is possible and should be done in order to see profit. Methods using an error correction module such as [3], [11], [12], [14] involve overhead and they are not automated. Hence, these methods are not proven to be generic. For another multiplier or adder, this method might not work anymore.

Some other state-of-the-art self-healing solutions are specific to certain arithmetic adders, such as [15]. This makes automation and maintaining genericness difficult. Even machine-learning is possible to use to apply self-healing as shown in [16], but this will not be part of the current research.

A more generic method of self-healing, but limiting to design space is [4]. In this work, the datapath in which approximating and healing is done, consists of multiple multiplications and an accumulator at the end. The accumulator stage acts as an healing stage since the accumulated results of the multiplier should cancel out. For example, one multiplier outcome results a number higher than accurate, whereas the other in a lower number with the same magnitude. The limitation in this research is that the number of multipliers must be even.

A somewhat more fine-grained method is [17] where alterations to truth tables were made to small multipliers. However, in this work the truth tables used only had 1 approximate entry for the whole multiplier. Hence, much of the design space is not explored yet.

A research of which its ideology is similar to the research in this paper is [18], called MACISH. This work is more fine-grained than the previously mentioned methods, because it creates small 2x2 multipliers by changing their truth tables. Using these small 2x2 multipliers, larger multipliers such as a 4x4 or 8x8 can be created. The errors in the 2x2 multipliers are canceled out by other 2x2 multipliers, using the ideology of [4] but not necessarily in pairs of two only. The focus in [18] is not so much how to adapt the 2x2 multiplier truth tables, but mostly how to substitute the small multipliers in a larger multiplier. Hence, it could miss much of the possible design space as the small 2x2 multipliers are not created automatically and a limited number of variations is available.

Compared to [18], the goal is to explore more of the design space by making changes even more fine grained, on full adder truth table level. Additionally, more different truth tables are created and tested to explore even more of the design space.

To conclude, the current fine grained self healing methods for approximate computing are limited. When the approach is somewhat fine grained such as [17], it does not explore the design space of the truth tables automatically and it misses a large part of the total design space. Moreover, [18] is one of the very few methods which does the approximating and healing internally. This work is the most fine-grained research so far. Overall, there is a lack of genericness and automation. This research attempts to find a new method which does cover an even more fine-grained methodology which could be generic and makes use of automated design space exploration.

III. Background

A. Full adders and half adders

The starting point of this research are the full adder (FA) and the half adder (HA). Their truth tables are shown in Table I and Table II, respectively. The full adder accepts three inputs and outputs 2 bits. The full adder is essentially a 1-bit adder including a carry out bit. The half adder accepts only two inputs, but also outputs two bits. Each cell has its own energy, which is found by synthesizing the cells.

Table I: Truth table of an accurate full adder. It serves as a 1-bit adder with A and B as input and C_in as carry in.

Input			**Output**	
A	**B**	**C_in**	**C_out**	**S**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table II: Truth table of an accurate half adder. It serves as a 1-bit adder with only two outputs, A and B.

Input		**Output**	
A	**B**	**C_out**	**S**
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The truth tables of these cells are changed to make them approximate FA and HA. When errors are introduced to the truth tables, the cell will have an error rate for each output bit ($error\_rate_{C_{out}}$ and $error\_rate_S$). The error rate per output bit is the rate that

the approximate output is not equal to the accurate output bit. For example, one error means an error rate of 1/8. This can either be that the approximate output is 0 instead of 1, or vice versa. Using the error rate the error of the carry output bit and the sum output bit are calculated as follows, respectively.

$$error_{carry} = error\_rate_{C_{out}} \cdot 2^1 \tag{1}$$

$$error_{sum} = error\_rate_{S} \cdot 2^0 \tag{2}$$

The carry output is twice as significant as the sum output bit. When a carry propagates through a circuit, it has a greater influence on the outcome. Therefore, the multiplication factor is twice as high than the sum output bit when multiplying with the error rate. The total error of a cell is then found by

$$error_{cell} = error_{carry} + error_{sum} \tag{3}$$

B. Array multiplier

The starting point of testing different methods is the array multiplier as shown in Figure 1. This array multiplier accepts two inputs A and B, each 3 bits, and outputs the multiplied product of these inputs as an unsigned 6-bit number (Output 5 to Output 0, most significant bit (MSB) to least significant bit (LSB). The array multiplier consists of FA and HA cells which can be accurate or approximate.

There will be many variations of this array multiplier and to assess the quality of each variation, two metrics are used. The estimated energy consumption, and the average absolute error over all input combinations. Firstly, it is infeasible time-wise to do design space exploration for a large design space when the array multiplier must be synthesized for every different design to obtain the energy. Hence, only the FA and HA are synthesized and the estimated energy for the total multiplier is the addition of each FA and HA energy consumption separately. Secondly, it is assumed that each input combination for A and B occurs evenly distributed. Therefore, for every design all input combinations are tested, the absolute error compared to the accurate outcome is saved, and the average of those errors is used as quality metric.

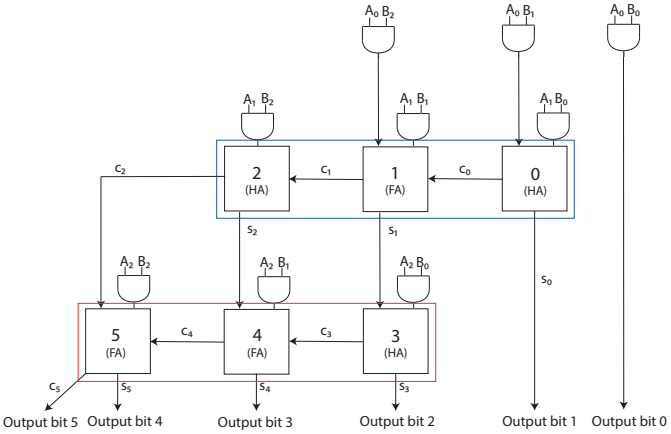


Figure 1: An unsigned array multiplier accepting 2 3-bit inputs A and B, resulting in a 6-bit number. Each coloured rectangular is the definition of a block for the method in subsection IV-B.

IV. Methodology

In this research, fine-grained self-healing concepts are tested to create approximate computing. The methods are more fine-grained than the state-of-the-art, but there can be still variation in granularity. The basis of all is still adapting the truth tables of FA and HA cells. There is one method which is based on the state-of-the-art self-healing [18]: partitioning by using blocks and canceling out using the blocks decimal errors but in a different way. A second method is tested which is even more fine-grained. The latter makes choices for each smallest element in the multiplier. It does this by using backtracking and pruning. The following subsections cover the proposed methods and are structured as follows.

1) Creating a reference
2) Blocks: Multiplier partitions
3) Use of blocks 1: Grid search
4) Use of blocks 2: Use error of blocks
5) Backtracking
6) Weights and pruning

The methods are all tested on the 3x3 multiplier as shown in Figure 1, and they are compared to a created reference. The reference is the best achievable pareto front of found multiplier designs within the set criteria and resources. The goal is to find a method which can find these pareto designs and that method can ultimately be used for larger multipliers with some guarantee that it will be close to the actual optimal result without the necessity to create another reference beforehand.

Instead of synthesizing every multiplier design, it is chosen to do only a functional simulation in Python to obtain the error of the multiplier.

A. Creating a reference

The fine-grained methodology adapts the truth tables of the FA and HA. For the full adder, there are 2^{16} different truth tables possible. There are 2^8 different HA variations. The number of all different 3x3 array multiplier designs is equal to

$$\text{Nr. different designs} = 65536^3 * 256^3 = 4.7 * 10^{21}. \tag{4}$$

Finding designs this way is not scalable, because there are more cells when the multiplier size increases. Hence, it is not time feasible to continue with all possible FA variations. Therefore, a selection of FA and HA is necessary at first. With only those selected FA and HA, the best possible pareto front is obtained by substituting those cells exhaustively. The result obtained by the exhaustive simulation is the reference and is the best possible front. A reference is necessary because one must be able to asses the quality of the found fine-grained self-healing designs.

Even though it should be possible to heal errors in the FA and HA cells in the multiplier, one cannot assume this can be done indefinitely. Hence, a first limit is set here on the maximum $error_{cell}$ as calculated by Equation 3, which is 0.5. This still introduces errors and possibilities to heal them. Even though applications may be error-tolerant, there is always a limit on the error. Therefore, already setting a limit on $error_{cell}$ prevents finding designs with disproportional high errors and low energy savings which will never be used in practice.

Setting a limit to $error_{cell}$ of 0.5 shrinks the number of cells down to 487 FA. This is still not feasible to do an exhaustive search with. Therefore, a design parameter is introduced: the number of nearby-pareto iterations. One could only take the pareto FA and HA regarding the cell's absolute error and their synthesized energy consumption. This would end up with 9 FA and 5 HA cells. However, it is possible to include the following pareto cells as well, the cells which are pareto when you do not take into account the previous pareto ones. These are not actual pareto, but nearby-pareto and they are part of a fat-pareto selection. This process can be repeated until the designer is satisfied with the reference line and the number of the cells is still feasible to experiment with.

It is worth to include non-pareto cells as well, because the error rate of a cell does not take into account where the errors are. It might be necessary to heal pareto cells with a non-pareto cells. When only pareto cells are selected, these solutions would be disregarded. Of course, the combined energy consumption should not exceed that

of accurate cells. Hence, it is not worth to look too far from the fat-pareto curve.

Since the concept of self-healing is canceling overestimated answers with underestimated answers or vice versa, it is important to make a distinction between those. Therefore, all cells are categorized into positive (net overestimating output bits) and negative (net underestimating the output bits). For each category, the nearby-pareto iteration process is done. This categorizing beforehand can result in better design space exploration.

B. Blocks: Multiplier partitions

Inspired from the state-of-the-art [18], there is still possibility to partitioning the multiplier in a fine-grained method. In this research, this is done by creating blocks: rows of FA and HA as shown in Figure 1. Using the same FA and HA as for the reference, as described in subsection IV-A, the blocks are created exhaustively. There is no additional logic between blocks and no overhead. Hence, the blocks are not treated as actual self-functioning multipliers. However, their output is treated as a decimal number to quantify their error such that only the best blocks can be chosen.

Since creating the blocks exhaustively results possibly in too many blocks, a selection is necessary as well. The choice of blocks is similar to the selection of FA and HA cells, selecting pareto blocks only or include nearby-pareto blocks. Note that creation must be done for a first row block and non-first row blocks separately, as they differ.

The blocks are created using the FA and HA cells which were found in a fine-grained manner. Hence, using blocks is still a fine-grained method. Partitioning the multiplier in blocks is a method to limit the design space, because no full exhaustive search is necessary anymore. The disadvantage of using blocks similar to these, is that they are not actual multipliers. Their error output is not actual a result of a multiplication, and it also does not store information on which bits of the blocks cause errors and how they propagate. Because of this, the healing effect will not be 100%. However, that does not mean that no healing is done. Error correction can still occur.

C. Use of blocks 1: Grid search

Grid search is similar to exhaustive search, in the sense that all designs are created with the available resources. The difference in this case, is that this grid search includes extra filtering steps. Firstly, blocks were created and either only pareto blocks or extra nearby-pareto blocks were selected. Those blocks were used to exhaustively substitute in the array multiplier to find designs.

Even though grid search substitutes blocks exhaustively, it can still be beneficial for the exploration because it can find many designs but the design time has been limited because of the creation and filtering of blocks beforehand. Instead of six positions to substitute for, there are only two. Therefore, the main advantage of a block grid search is the reduction of design time.

Disadvantage is that it might not be scalable or easy to make it scalable for larger multipliers. For instance, a 16x16 array multiplier could be divided in different forms of blocks. One could keep using rows, but substituting 16 rows has two disadvantages. Firstly, creating rows is exhaustively substituting 16 cells which is time infeasible. Secondly, substituting 16 rows might also not be the fastest method to find a good pareto front. Therefore, blocks might have to be created differently for larger multipliers which adds up to the design time. Furthermore, what the block looks like is different for every arithmetic unit and its size.

In a 16x16 multiplier when using rows of 16 cells, if one wanted to create these rows with only the pareto FA and HA, it would already create $9^{14} * 5^2$ different rows for the first row, and $9^{15} * 5$ different rows for the non-first rows. In case you would only select 10 rows, as an example, there would be 10^{16} designs using grid search. This is less feasible.

A solution could be to have extra filtering steps when creating rows for grid search. One could take into account the significance of each cell such that the most significant cells in a block contain less errors than the least significant cells. Furthermore, each row can also have different error criteria when selecting, depending which row is substituted for in the multiplier. However, these solutions are not scalable, nor generic for other multipliers or adders.

D. Use of blocks 2: Use error of blocks

Instead of doing a block grid search, one could also replace the blocks in the array multiplier in a more intelligent way. This method is inspired by [18]. Each block has an error which is treated as a decimal number. Using the same analogy as in the state-of-the-art coarse grained healing, one can substitute blocks based on the errors of previously substituted blocks in the multiplier.

For example, if the blue block in Figure 1 has an error of 3, then the red block should counter this error. Since the red block carries more significance, the red block's error should only be -1.5 and not -3. One could build in a margin which can choose how much off the error is of the red block such that an error of -1.3 is also tried in the design space for example. This can be a design parameter as well to limit design time. In this research, three blocks are chosen with their error closest to the desired error.

In this method, the blocks are treated as separate multipliers because their outcome is treated as a decimal number. In reality this is not truly correct, because a block is not a multiplier in itself. Hence, the canceling out will not fully occur. It is not taken into account which connections and which cells cause the error within a block.

The advantage of this method, is that it is quicker than grid search. Not all possible designs are simulated. Another advantage is that tweaking is possible using the design parameters: how many blocks to select, and the error margin. The disadvantage is that it is not fully reliable since the blocks are not actual multipliers. The error of a block is represented as a decimal number. If the blue block has an average error of 3, it is not clear which bits cause the error and how the error is propagating. Assuming an error of -1.5 for the red block can fix some of the propagation errors, but not all of them.

This method is much more scalable than the grid search, because not all possible combinations are tried. In a larger multiplier of 16x16 there could still be 16 rows, but the number of tried designs are limited based on the design parameters. Hence, when a good error margin is set which still limits the design space, this method could be scalable.

As an example, if there were 10 options per row, in this method only 1 or 2 (depending on the design parameters) rows could be tried depending on the error margin set. This would limit the design space from 10^{16} to 2^{16}. Of course, when more design time is possible, a wider margin could be set and more options per row could be included.

E. Backtracking

Backtracking is a common algorithm used in many applications [19]. In this research, it can be explained as follows, using Figure 2. The tree depicts the design space. The first top node is the design node for the first cell HA0 in Figure 1. There are 4 options in this example for HA0, hence, four outgoing arrows to next nodes. From there on, one must choose what FA1 is going to be. This can be the same number (as in the picture), but also more or less. The depth of the tree is equal to the number of cells in the multiplier. The number of outgoing connections leaving each node, is the number of possible FA and HA for that cell. In backtracking, the search is depth-first. Figure 2 serves as an explanation and is not depicting the actual design space, as the design space is too large to visualize properly.

Backtracking is a method to search designs systematically. However, when the design space is large it consumes much time. To disregard subtrees (part of the design space), backtracking uses pruning. Pruning is stopping the search from that point on because it will not give better results anymore. When and how to prune is

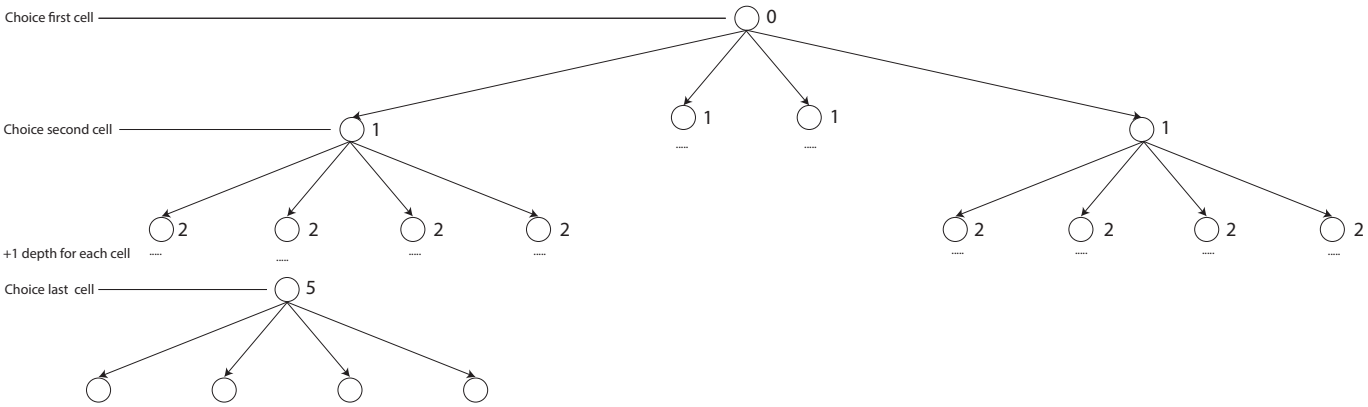


Figure 2: A backtracking tree depicting the design space, each node has as many child nodes as there are choices for the next cell in the array multiplier. The number at each node indicates for which cell a choice is made at that point. Backtracking is a dept-first search method, which is an exhaustive search when there is no pruning. Three dots suggest that there are sub-trees as well but left out for visual purpose.

defined by the designer, who creates 'rules' how to prune. Those rules indicate which paths should or should not be explored. When the pruning rules are implemented correctly, the algorithm can find the optimal solutions without searching the whole design space.

In backtracking, heuristics are used to give 'score' to each solution, a value to indicate the quality of the solution so far. If a certain design choice has a low score, it can be chosen not too continue this path and prune it. In this case, if options are chosen for the first three cells and those options can not lead to an optimal design anymore according to a set heuristic, it prunes this part of the design space.

The advantage of backtracking is that it is a well-known and matured method and easy to implement. Secondly, it could be generic. It is tested on this array multiplier, but it can work for any arithmetic unit. As long as the connections between cells are known, backtracking is possible. However, pruning rules can differ per arithmetic unit.

The disadvantage is that it is difficult to find good heuristics and knowing the limits of pruning. There is a trade-off between pruning and design time in this research. One could choose to set strict pruning rules to limit design time, at the cost of not finding the reference designs.

F. Weights and pruning

In this research, the pruning of the design space is making use of the knowledge of the array multiplier. Each connection between cells contribute to the output eventually. However, not each connection is as significant as the other. For instance, c_4 in Figure 1 could cause a greater error than c_0 on the final output if chosen with a high $error_{cell}$ for two reasons. Firstly, because c_0 is early in the chain and hence, some correction might still be applied. Secondly, c_4 contributes with a higher impact on Output 4 and Output 5 which are the two most significant output bits.

The goal with this method, is to assign weights to each connection and use those weights to estimate the error and therefore, the quality of a choice. If the weight assignment is done correctly, the weights together with the known errors of each cell could make an estimation of the actual error of multiplier designs. There is an infinite number of ways to assign weights. In this research two weight assignments are tried. One based on [20], and one based on output bit significance.

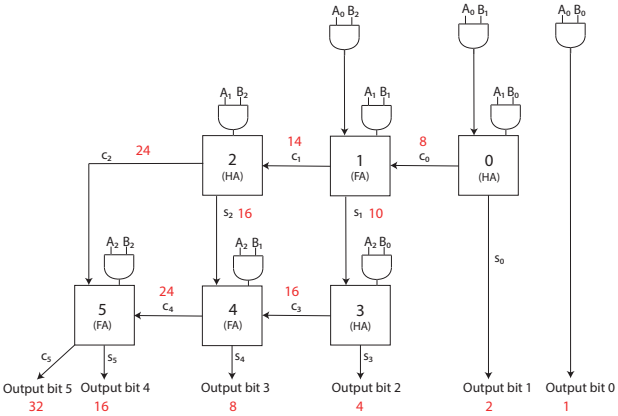
Both weight assignments start at the same point. Each output bit 0-5 is assigned a significance. This is simply 2^n where n is the output bit number. From there on, the two assignments differ. The final weights assignments are shown in Figure 3a and Figure 3b.

Firstly, [20] assumes that each connection contributes to the whole chain this connection is part of. Even though c_4 is more significant than c_0, c_0 could still have impact on Output 4 and Output 5, because c_0 impacts HA1 which impacts HA3 and so on. This applies for all connections. Hence, the weight of each output bit is taken and accumulated to each connection which is on the path to this output bit, just as [20] suggests. However, in this research it is taken into account that the impact of c_4 on Output 4 is greater than the impact of c_0 on Output 4. Namely, instead of directly accumulating the weight of $2^4 = 16$ to connection c_0, it is divided by $2^4 = 16$ because four is the logical depth of the cell from the primary output. This is done for each output bit to each connection.

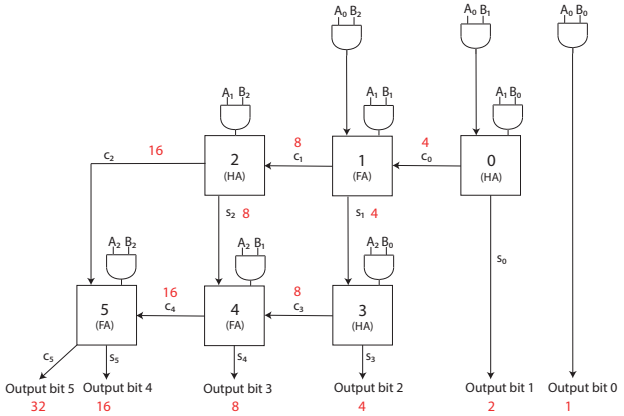
Secondly, the weights can also be assigned more scientifically. The array multiplier can be seen as columns of cells in which each column contributes to an output bit. Each column to the left is twice as more impactful than the column to the right. All cell connections in the column of Output 2 have the same weight, namely, $2^2 = 4$. The carries are twice as significant as the sum bits so, c_1 and $c3$ have the same weight as the sums in the column of Output 3, namely, $2^3 = 8$.

Using these weights, it is possible to choose cells for each position. The pesudo-code is shown in algorithm 1. The algorithm works as follows. Each cell has its own error rate for the carry and sum output bits. When you multiple the error rate with the corresponding weight, you get the expected error that that cells gives for that connection. When you do this for a whole datapath and accumulate those results, it is possible to calculate what the error is at a certain point in the datapath. It is important to take into account all paths leading up to a point, an example is shown in Figure 4. In this example, a choice must be made for FA4. The incoming paths are chosen already so their error rates and weights are known. These paths will accumulate into an error in FA4.
When this error is known, the current cell FA4 should counter this. The desired error of FA4 should be the same in magnitude but different in sign of the incoming accumulated error. This is the desired healing effect. The error rates of each possible cell is known, as well as the weights of each connection. Hence, only cells with suitable error rates to heal the incoming accumulated error are suitable. The rest of the cells should not be tested anymore and that part of the design space can be pruned.

There are different levels of strictness here. One can choose to only take the cheapest suitable cell, but also to take a top three of the cheapest or something else. It is worth to use more than only 1 option, because the error rate of multiple cells can be the same, their error distribution is not. The error rate does not qualify which inputs are approximated and this may play a role in the quality of the designs.



(a) Accumulative weight assignment, inspired by [20]



(b) Weight assignment based on output bit significance

Figure 3: Two types of weight assignments to connections in the array multiplier. Weights are used to estimate the error and to set pruning rules for backtracking.

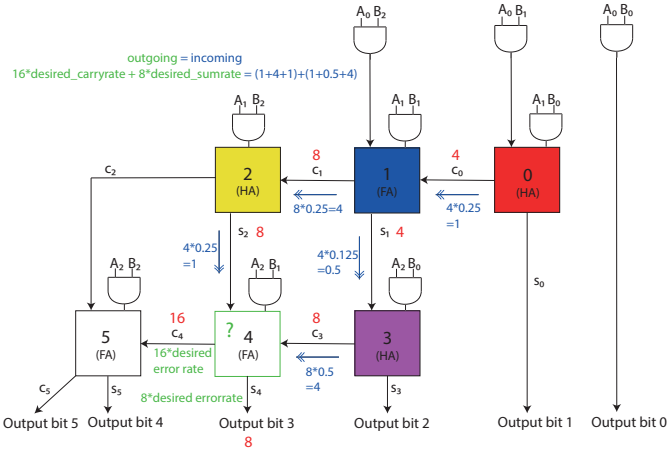


Figure 4: Example with example error rates to illustrate how weights and error rates are used. Here, a choice for FA4 must be made. Colored cells are cells which have been chosen already before this point.

Algorithm 1: Backtracking

```
Function backtrack_main(available_cells,
 mult_graph, current_cell, pareto_front):
    if current_cell is last cell from mult_graph then
        current_energy, current_error =
         simulate_design(mult_graph)

        add current_energy and current_error to
         pareto_front if they are pareto
    end
    for cell in available_cells do
        current_cell.cell_type = cell
        if prune(mult_graph, current_cell) then
            continue
        end
        backtrack_main(available_cells, mult_graph,
         next cell in mult_graph, pareto_front)
    end

    return pareto_front

Function prune(mult_graph, current_cell):
    set variable intermediate_error to accumulated
     error of all cell errors up to current_cell
    desired_error = -intermediate_error

    if current_cell.sum_error +
     current_cell.carry_error is not desired_error then
        return True
    end

    return False
```

V. Results

Similar to the methodology section, the result section shows and explains the findings in subsections for each method. The result section consists of showing the reference pareto front of this research, the results of each method, and an analysis on the runtimes and scalability of each method. There are design parameters involved which influence the results and the time, these are summarized in Table III.

A. Reference

The design parameter in creating the reference is the number of nearby-pareto iterations to perform when choosing the FA and HA for the exhaustive simulation. The design space is already limited by this so, it is not fully exhaustive. Hence, this will be referred to as the selective exhaustive reference. Figure 5 shows the reference front and the impact of choosing more FA and HA.

The front becomes more dense and shows better cheaper designs when choosing more FA and HA. The gain from pareto only to one iteration extra is larger than the gain from one iteration to two iterations extra. No conclusion can be drawn yet if this trend continues when selecting even more different cells, as this is infeasible to simulate. Nevertheless, the fronts can still serve as a reference as they show the best achievable front for each selection of FA and HA.

Analyzing Figure 5, the cheaper designs with the same average error come mainly from an effect that is typical for the array multiplier. The array multiplier has input dependencies. This leads to the effect that certain inputs do not occur. If the cell's truth table

Table III: The values of the design parameters in this research

Method	**Parameter**	**Value**
Exhaustive reference	FA nearby-pareto iter.	2, total of 25 FA
	HA nearby-pareto iter.	2, total of 10 HA
Block grid search: pareto only	First row blocks nearby-pareto iter.	0 (only pareto), total of 23 blocks
	Non-first row blocks nearby-pareto iter.	0, total of 20 blocks
Block grid search: more blocks	First row blocks nearby-pareto iter.	3, total of 109 blocks
	Non-first row blocks nearby-pareto iter.	5, total of 155 blocks
Blocks: based on error	First row blocks nearby-pareto iter.	3, total of 109 blocks
	Non-first row blocks nearby-pareto iter.	5, total of 155 blocks
	Error margin	Three blocks with their error closest to desired error

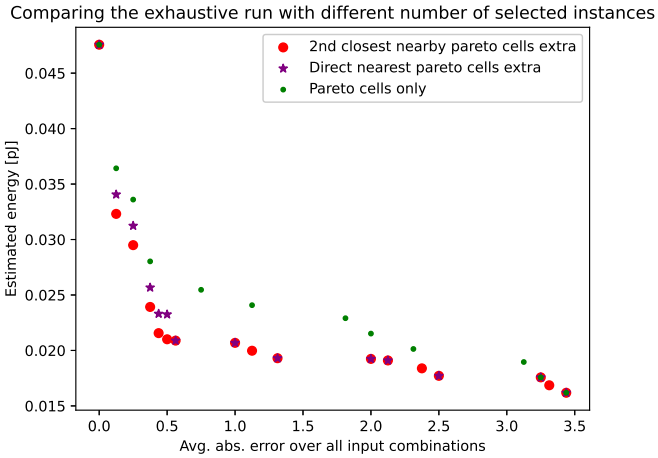


Figure 5: Results of exhaustive search with a selection of FA and HA cells. It also shows the impact of choosing more FA and HA. The red dots serve as the reference for this research.

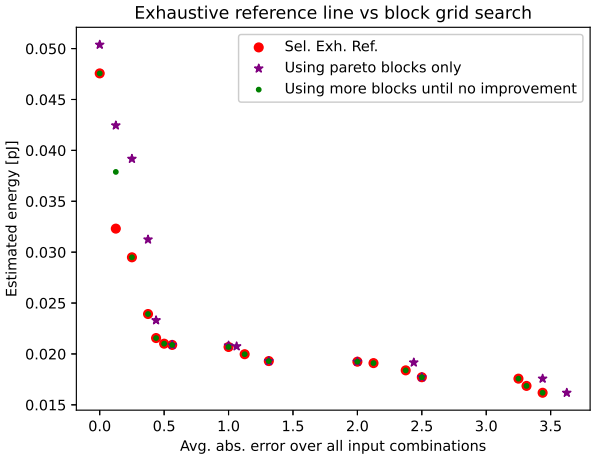


Figure 6: Results of a block grid search, showing the impact of choosing more blocks in the grid search. The red dots are the reference.

has an error for such input, then that approximate output is never triggered. Hence, this cell might act as if it is accurate.

For example, the approximate cell variant with such property was substituted in place 1 and 5 in Figure 1. The input combination for which cell gave an approximate output never occurred. This is verified by tracking what inputs A and B should be in order for this input to occur and that could not happen. It required bits of A and B to be 0 and 1 simultaneously.

Another particularity in the fronts, is that after an average absolute error of 0.5, the designs contain a higher error but the energy savings become less. The interesting part of the front, is where the front is steep and that the increase in error is still worth because of significant energy savings.

B. Block grid search

When creating the blocks, it is again a design parameter how many blocks out of all are selected to do the block grid search with. The results of the block grid search are shown in Figure 6. When selecting only the pareto blocks, the front does not fully cover the reference. However, when doing more nearby-pareto iterations when selecting, it is possible to find the reference points.

The block grid search has potential to find the reference designs. The grid search front in Figure 6 that is almost fully covering the reference, was obtained by doing 5 nearby-pareto iterations of selection for the first row, and 3 nearby-pareto iterations for the non-first row. The gain obtained by doing another pareto iteration every run, was only a few points or little movement closer to the reference. This is trade-off between design time and the quality of the found designs.

It is possible to analyze the errors of the blocks to verify if a pattern of self-healing is found. Unfortunately, the pareto designs contained combinations of blocks of which their errors were both negative. There was no consistent evidence of self-healing.

When analyzing which blocks were used to create the reference, many blocks were close to the block selection front. However, there were blocks which would only be picked when taking a lot of nearby-pareto iterations. This would slow down the design time significantly and hence, it is up to the designer if that is worth it. In this research, it was chosen not to since the current grid search with the current block selection came already close to the reference with much time savings and this is only to prove a concept.

C. Replace blocks using their errors

The result of choosing blocks based on their errors is shown in Figure 7. As was expected when analyzing the results of Figure 6, this front does not overlap with the reference. The same blocks were used in this method as the one in Figure 6 which overlapped with the reference. Therefore, with these blocks it should be possible to achieve the reference front.

It was attempted to increase the margin. Instead of choosing only the one block with their error closest to the desired error, the three closest errors were picked. This way, there are two extra options. Unfortunately, this addition did not move up the front at all. This confirms previous suspicion that the margin must be increased in such way that it becomes close to the grid search again.

Even though the front does not deviate from the reference front drastically, it cannot be guaranteed what happens when the multiplier is larger. The fronts may deviate more the bigger the multiplier becomes. In that case, there is no certainty if it will remain close to the best possible front.

D. Backtracking

The results of the backtracking with pruning are shown in Figure 7. There are two fronts for this method, one for each weight assignment.

Unfortunately, using the same ideology regarding error cancellation as for the blocks, the backtracking method does not come close to the reference front. It is assumed that the weight assignment with the error rates is simply too restricting for the design space exploration. When analyzing Figure 5, the pareto solutions gave no consistent sign of making use of canceling out, and no direct correlation was found using error rates.

Additionally, from Figure 5, it can be assumed that the error of each cell alone is not sufficient to make a choice. It is possible to obtain zero error but still include approximate cells due to the input dependencies. In the backtracking, those pseudo-accurate cells still have an error and might not be chosen even though they do not cause an error in the multiplier.

The difference between the two weight assignments is also peculiar. The output bit-based assignment has the same weights multiple times for connections in the multiplier, whereas the weight values in the accumulated assignment are all different. In the latter, this makes the outcome of the calculation of the desired error quite unique and hence, restricting to the design space. This explains why there are more data points for the output bit-based assignment than as for the accumulated assignment.

E. Comparison

All designs found by the tested methods are shown in Figure 7. To quantify the quality of each front, the area below the curve (AUC) is calculated using the trapezoidal rule. Since the interesting part of the fronts is where the curve is steep, the area of the curves up to an error of 2.5 is also shown in Table IV.

The zoomed-in version (Figure 7b) is more representative for comparison than Figure 7a as the interesting part is the part where the energy savings are worth the accuracy loss. When comparing the AUC, the previous analysis is confirmed regarding the quality of the found designs per method.

F. Run times

Besides the energy consumption and accuracy, the run time of each method is crucial in the trade-off as well. The run times are shown in Table V. For comparison purposes, the grid search was also run for a slightly larger 4x4 array multiplier.

Creating the reference for the 3x3 multiplier was already time-consuming and hence, for a 4x4 multiplier or even larger, no reference such as this can be created. The block method and the backtracking method decreased the run time in such magnitude that tweaking of design parameters was possible without significant increase of run time.

When applying a method with blocks, one must take into account that creation of the blocks also take time. For larger multipliers, this is even more because creation of blocks takes more time and also substituting these blocks. This is shown when comparing the 3x3 multiplier run times with the available 4x4 times. As mentioned in section IV, it could be possible to create large blocks from previously created 4x1 blocks for example which would limit the increase of design time when creating blocks.

For the 4x4 multiplier, the grid search takes up a few minutes but increases drastically to hours when using more blocks. From Figure 6, it can be assumed that using more blocks in the grid search does move up the front closer to the reference, and again, it is the choice of the designer to decide how much time is spent. Important to note here, is that besides the larger blocks, the total size of the multiplier is also larger. Each functional simulation of a 4x4 multiplier takes more time than a 3x3 multiplier. Hence, it would be advised to use a scalable method to simulate the multiplier functionally.

When comparing the run times and number of configurations run to [18], this research used methods which were significantly slower. [18] was able to explore $5.42 * 10^{44}$ 8x8 multiplier designs within 4 minutes. The methods that were run below 4 minutes in this research, did not even come close to this number of configurations.

Nevertheless, the methods proposed in this paper showed that it is possible to limit the design space significantly as shown in Table V, with still the possibility to find pareto reference designs using the grid search. This indicates that using blocks has the potential to shrink down the design space without pruning away optimal results.

G. Scalability

Each method has its own advantages and disadvantages regarding the designs and design time. The tests have been performed only on a 3x3 multiplier. Therefore, it is important to assess the scalability of each method for further research.

Firstly, the exhaustive search is not scalable. Even when pareto cells are taken only, it is not feasible to run. It could only be possible when a stricter selection is performed, but in that case, it would not be exhaustive anymore but already including some sort of heuristics.

Secondly, the block grid search showed that it was possible to obtain the reference designs using blocks. However, this method is not perfectly scalable either. Creation of blocks become more difficult when scaling to a large multiplier such as 8x8. A different partition than rows will be necessary because a row of 8 cells takes up much time to create. In this case, the approach of [18] can be an inspiration. It is not recommended to have different size of blocks every time the multiplier becomes larger, because this increases design time and manual effort significantly.

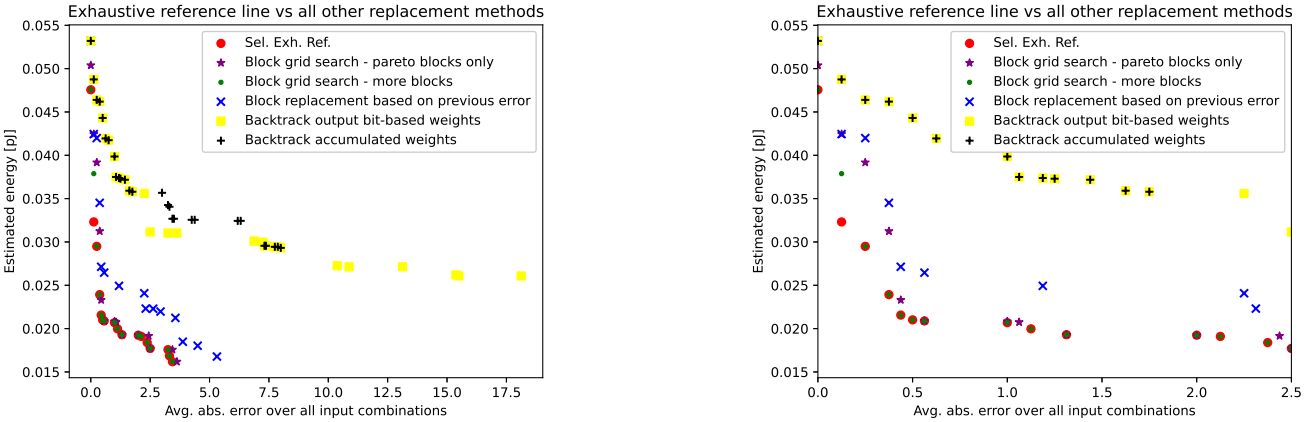
Thirdly, choosing blocks based on previous errors can have some potential. However, no clear evidence of using the blocks' error like this has been found. This was verified by analyzing which blocks the reference designs consist of. In case this method would be pursued, the same recommendations apply as for the grid search. The block size must be consistent for every size of multiplier to prevent manual effort. Additionally, a clear replacement strategy is necessary because a block in a larger multiplier might be connected to multiple blocks. Hence, treating the output of a block as a decimal number might even have a worse effect than in this research.

Lastly, the backtracking could be scalable. However, it is not efficient to use substitution per cell in the multiplier. The input distribution was not taken into account, and neither the error distribution of a cell. Even if weights of the connections are assigned correctly, a lot more criteria must be set in order for backtracking to be working. Additionally, for other multiplier types or adders, pruning rules must be created again. Hence, the method in itself is generic, but the implementation is not with respect to pruning. However, 02 showed the positive sides of backtracking when making use of blocks. Therefore, perhaps the block approach could be combined with backtracking.

VI. Future work

Regarding self-healing to create approximate arithmetic units, there are several factors to take into account. One can continue the path of fine-grained self-healing, or go back to more coarse-grained healing. In both categories, there are some recommendations and also some general recommendations.

Firstly, in case one wants to explore this fine-grained approach more, it is advised to find a good weight assignment and pruning rules for backtracking. Moreover, it is important to to trade-in some of the genericness to find better pruning. The input distribution is important to do good pruning which was not done in this research. Good pruning and making use of the error rates speeds up the process because no full simulation is necessary to obtain the error, since the weights could estimate the results decently. Error estimation could also be done in another way, as described in [21] for example. This is another topic of research, but in case of approximate computing it can be useful.



(a) Full range

(b) Close-up

Figure 7: Comparison of all pareto fronts of tested methdos and a close-up up until an error of 2.5

Table IV: AUC (area under curve) of the pareto fronts of all tested methods. The AUC is calculated using the trapezoidal rule.

Method	**AUC [x10^{-3}]**	**AUC up to error of 2.5 [x10^{-3}]**
Reference (exhaustive)	308.26	54.16
Block grid search: pareto blocks only	312.40	58.00
Block grid search: more nearby-pareto blocks	308.95	54.85
Choosing blocks based on their errors	338.94	69.04
Backtracking: accumulative weights	573.64	99.17
Backtracking: output bit-based weights	539.78	98.52

Table V: The run time of each method along with the total number of designs found. Unless specified, all methods were run on a single thread on a AMD Ryzen Threadripper.

Method	**#Designs**	**Run time**	**Remarks**
Exhaustive reference	15625000	3.9h	5 machines in parallel: 4x 24 threads on AMD Ryzen Threadripper 2920X 1x 16 threads on Intel(R) Core(TM) i9-9900K
Creating blocks	-	26s	
Block grid search: pareto blocks only	460	1.29s	24 threads on AMD Ryzen Threadripper 2920X
Block grid search: more nearby-pareto blocks	16895	37s	24 threads on AMD Ryzen Threadripper 2920X
Block replace based on error	445	16s	
Backtracking: accumulative weights	2187	72s	
Backtracking: output bit-based weights	2277	74s	
Creating blocks 4x4	-	107s	
Block grid search 4x4: pareto blocks only	22464	9m	24 threads on AMD Ryzen Threadripper 2920X
Block grid search 4x4: more nearby-pareto blocks	809900	14.5h	24 threads on AMD Ryzen Threadripper 2920X

Additionally, the current methods in this paper were not tested on a different multiplier or an adder. The initial purpose of the research was to find a generic method, but that has not been tested yet. The same methods could be implemented for another multiplier or adder to verify if the methods were in fact generic or not.

For the array multiplier, making use of redundant inputs led to cheaper designs without error increase. It can be chosen to exploit this fact to create better array multiplier designs, or to explore this effect on other arithmetic units too. This would not make use of the self-healing property, but it is another way to possibly create approximate circuits.

Secondly, as the results might suggest, a more coarse-grained approach could result in better results which are closer to a set reference. To improve upon the state-of-the-art, it is recommended to include more variations of possible truth tables when creating approximate circuits. Additionally, automation of the process could still explore much of the design space even if the granularity is coarse. Suggestions on automated design space explorations are made in [22], [23].

Lastly, regardless the granularity, there are several general recommendations as well. In this research the simulations were done purely functional. In the future, one might look at a more efficient way to do this than Python. Moreover, the input distribution was assumed to be homogeneous. In reality, when knowing the application, one knows which inputs occur more often or if certain inputs do not occur at all. When making use of this, it could open doors to more optimization.

VII. Conclusion

The state-of-the-art self-healing concept to create approximate circuits is not automated, nor fine-grained, and nor exploring a large design space. This research aimed to find a method which includes those three properties, and test it in an array multiplier. The results showed that the attempted methods did not fully succeed on all three properties. The block grid search was able to find results close to the set reference, but the method might not be scalable for larger multipliers and is not generic. Using backtracking in combination with good pruning, fine-grained self healing can be scalable. Unfortunately, the pruning was unsuccessful in this research.

Recommended is to focus on a slightly less fine-grained method, but include automation of the design space exploration, contrarily to the state-of-the-art. When the design space is large, it is also recommended to focus on a good functional simulation model to speed up the process or to have model which can estimate the error close to the actual error. It is the designer's choice if the method should remain as generic as possible, or to include knowledge of the arithmetic unit and input distribution of the application. The latter is advised.

REFERENCES

[1] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, L. Pozzi, and S. Reda, "Approximate logic synthesis: A survey," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2195–2213, 2020.

[2] M. Shafique, R. Hafiz, S. Rehman, W. El-Harouni, and J. Henkel, "Invited: Cross-layer approximate computing: From logic to architectures," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2016.

[3] M. A. Hanif, F. Khalid, and M. Shafique, "Cann: Curable approximations for high-performance deep neural network accelerators," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2019.

[4] G. A. Gillani, M. A. Hanif, M. Krone, S. H. Gerez, M. Shafique, and A. B. J. Kokkeler, "Squash: Approximate square-accumulate with self-healing," *IEEE Access*, vol. 6, pp. 49112–49128, 2018.

[5] S. Venkataramani, V. J. Kozhikkottu, A. Sabne, K. Roy, and A. Raghunathan, "Logic synthesis of approximate circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2503–2515, 2020.

[6] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1367–1372, 2013.

[7] D. Hernandez-Araya, J. Castro-GodÃnez, M. Shafique, and J. Henkel, "Auger: A tool for generating approximate arithmetic circuits," in *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*, pp. 1–4, 2020.

[8] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: Imprecise adders for low-power approximate computing," in *IEEE/ACM International Symposium on Low Power Electronics and Design*, pp. 409–414, 2011.

[9] S. De, J. Huisken, and H. Corporaal, "Designing energy efficient approximate multipliers for neural acceleration," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, pp. 288–295, 2018.

[10] S. De, J. Huisken, and H. Corporaal, "Designing energy efficient approximate multipliers for neural acceleration," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, pp. 288–295, 2018.

[11] S. Mazahir, O. Hasan, and M. Shafique, "Adaptive approximate computing in arithmetic datapaths," *IEEE Design Test*, vol. 35, no. 4, pp. 65–74, 2018.

[12] S. Mazahir, O. Hasan, R. Hafiz, M. Shafique, and J. Henkel, "An area-efficient consolidated configurable error correction for approximate hardware accelerators," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2016.

[13] J. Hu, Z. Li, M. Yang, Z. Huang, and W. Qian, "A high-accuracy approximate adder with correct sign calculation," *Integration*, vol. 65, pp. 370–388, 2019.

[14] T. K. Kodali, Y. Zhang, E. B. John, and W.-M. Lin, "An asynchronous high-performance approximate adder with low-cost error correction," *J. Inf. Sci. Eng.*, vol. 36, pp. 1–12, 2020.

[15] C.-H. Lin and I.-C. Lin, "High accuracy approximate multiplier with error correction," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 33–38, 2013.

[16] M. Masadeh, O. Hasan, and S. Tahar, "Machine learning-based self-compensating approximate computing," in *2020 IEEE International Systems Conference (SysCon)*, pp. 1–6, 2020.

[17] S. Mazahir, O. Hasan, and M. Shafique, "Self-compensating accelerators for efficient approximate computing," *Microelectronics Journal*, vol. 88, pp. 9–17, 2019.

[18] G. A. Gillani, M. A. Hanif, B. Verstoep, S. H. Gerez, M. Shafique, and A. B. J. Kokkeler, "Macish: Designing approximate mac accelerators with internal-self-healing," *IEEE Access*, vol. 7, pp. 77142–77160, 2019.

[19] P. Civicioglu, "Backtracking search optimization algorithm for numerical optimization problems," *Applied Mathematics and Computation*, vol. 219, no. 15, pp. 8121–8144, 2013.

[20] M. Mousavi, S. De, H. R. Pourshaghaghi, and H. Corporaal, "Fault tolerant fpgas: Where to spend the effort?," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*, pp. 651–654, 2019.

[21] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, and L. Pozzi, "A formal framework for maximum error estimation in approximate logic synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2021.

[22] D. Hernandez-Araya, J. Castro-GodÃnez, M. Shafique, and J. Henkel, "Auger: A tool for generating approximate arithmetic circuits," in *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*, pp. 1–4, 2020.

[23] N. Wu, Y. Xie, and C. Hao, "Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning," 2021.