January 27, 2019

BACHELOR ASSIGNMENT

COMPARING SELF-HEALING TECHNIQUES IN APPROXIMATE MAC ACCELERATORS

K. Raben s1568426

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Computer Architecture for Embedded Systems

Exam committee:
G.A. Gillani, A. B. J. Kokkeler, M.S. Oude Alink

UNIVERSITY OF TWENTE.

Abstract

Approximate computing is the technique of trading in accuracy for efficiency in calculations. This efficiency can come in many forms such as reduction in used hardware area or reduced power consumption. To even further these methods, circuits can be designed that introduce self-healing, where some portion of the reduced accuracy can get internally compensated by letting different errors cancel each other out. One of these circuits is a Multiply Accumulate (MAC) circuit, where a multiplication is made and then added to the total result.

This work focuses on comparing two different self-healing techniques that can reduce the cost of the multiplier in the MAC, one where a balancing multiplier is added and one where the multiplier is internally balanced. This work provides results on which technique is more effective in what situations executed on an FPGA. A model is made that evaluates different MAC configurations with parameters found by performing FPGA hardware simulations. From these evaluations the conclusion is drawn that a internal self-healing technique, where an approximate MAC is build with a multiplier that is internally balanced, is more effective.

Contents

1 Introduction

Approximate computing is the technique of computing with a smaller degree of precision which creates opportunities for improving systems in terms of used area, power consumption and performance efficiency. This technique can be utilized in applications which can tolerate approximation errors and will still output information that is useful. Examples of these applications are deep learning networks, machine learning, web searches and image and video processing[1].

To increase efficiency even further, error balancing can be introduced in certain structures that will reduce the average error of these structures. One of these structures is a Multiply-Accumulate Circuit (MAC), which is often used in radio astronomy[2].

A MAC calculates the dot product of two input vectors, meaning that every element of vector $\overrightarrow{A}$ gets multiplied with the corresponding element of vector $\overrightarrow{B}$ and the results get added up, as illustrated by Fig. 1.

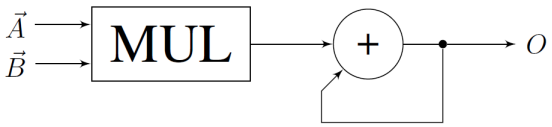


Figure 1: MAC block diagram[3]

When building a MAC with approximate multipliers, there is a chance that the multiplier will make an error, say δ_m. This δ_m can however be compensated in the accumulator stage if another error was made with the same magnitude, only with a different sign, i.e., $-\delta_m$.
Using this compensating ability, configurations for a MAC can be found that have lower measures of error than when using conventional approximate computing techniques while still using the same amount of resources.

Different techniques utilizing this Self-Healing ability have been developed. One technique focuses on balancing the error internally by configuring the multiplier in such a way that the produced errors can be healed[4]. The other technique uses an additional "mirror" multiplier that balances the first multiplier[5].
In this work, the goal is to answer the question: What approximate multiplier technique using an FPGA is best when used in accumulation based accelerators? A model is build to evaluate approximate multiplier configurations. The model outputs results for different quality metrics and an estimation for the resource costs. This work will discuss the subjects of approximate multipliers and Self-Healing techniques more extensively and will show the method for creating the model and obtaining the parameters used in it.

2 Background

2.1 Approximate Multipliers

One can create a multiplier of n×n-bits by constructing it using multiple elementary 2×2 multipliers. The realization of an accurate 2×2 multiplier in hardware can be found in Fig. 2. The corresponding truth table of Fig. 2 can be found in Fig. 3.

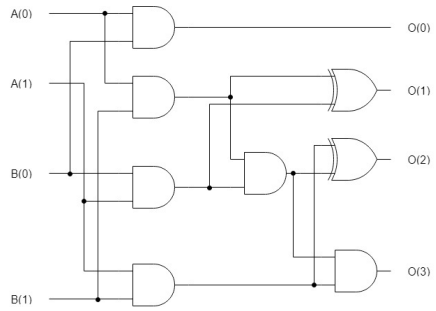


Figure 2: Accurate 2×2-bit multiplier

A / B	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	1001

Figure 3: Truth table of an accurate 2×2 multiplier[3]

Using four of these multipliers and some adder logic a 4×4-bit multiplier can be build. The functioning of the logic can be seen in Fig. 4. The 4-bit inputs get divided into two 2-bit elements, after which every possible 2-bit multiplication is made. Adding these partial products with Shifting the products of the more significant bits as shown in Fig. 4 will conclude in an 8-bit result of the 4×4-bit multiplier.

This same process can be repeated to create a 8×8-bit multiplier using four 4-bit multipliers, a 16×16-bit multiplier with four 8-bit multipliers etc.

To reduce hardware area and power cost, approximate circuits are build that use a reduced amount of logic to compute the result[6]. This preservation of resources comes with the cost that not all outputs are correct. One of these approximate circuits is shown in Fig. 5. This multiplier has one error case, namely that $3 \times 3 \rightarrow 7$ instead of 9. This multiplier thus has an error case of -2. Different circuits with different error magnitudes and amount of error cases have

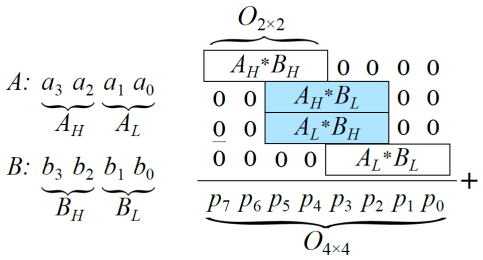


Figure 4: Design of a 4×4-bit multiplier[5]

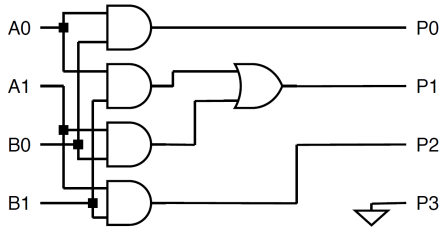


Figure 5: Approximate multiplier M2[4]

been generated[4]. Conventional approximate computing builds bigger multipliers out of these elemental multipliers to create systems that have a certain error probability but where the gains in hardware area and power consumption make up for this sacrifice.

When using approximate elementary 2×2 multipliers inside of a larger multiplier like an 8×8 multiplier a certain error is introduced. Important to note is that the significance of that error is dependant on where the elementary 2×2 multiplier is placed in the larger multiplier. In Fig. 6 the functioning of an 8×8 multiplier build from elementary 2×2 multipliers can be seen. In a $n \times n$ multiplier, where n always is a power of 2, there are $n^2/4$ elementary 2×2 multipliers needed. In the case of a 4×4 multiplier like in Fig. 4 that results in 4 elementary multipliers, and in the case of a 8×8 multiplier like in Fig. 6, 16 elementary multipliers are used. The output $O_{8\times8}$ of the 8×8 multiplier is given in Eq.(1), from which it can be seen how significant a certain multiplier, and the possible error made, is to the end result.

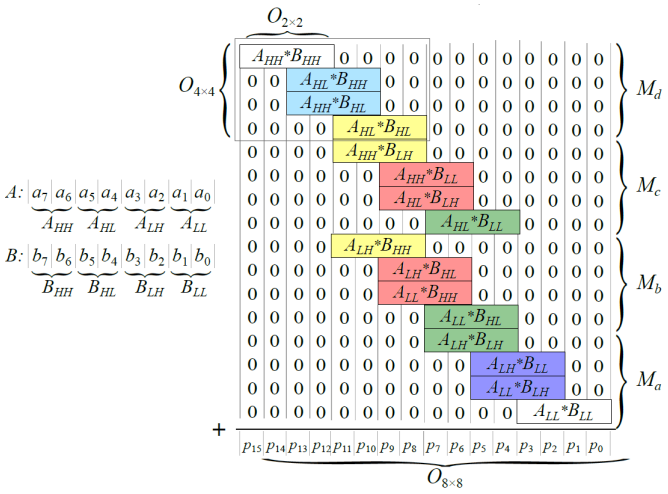


Figure 6: A 8×8-bit Multiplier Using 2×2-bit multiplier elements[5].

$$
\begin{aligned}
O_{8\times8} = {} & 4096(A_{HH}B_{HH}) \\
& + 1024(A_{HL}B_{HH} + A_{HH}B_{HL}) \\
& + 256(A_{HL}B_{HL} + A_{HH}B_{LH} + A_{LH}B_{HH}) \\
& + 64(A_{HH}B_{LL} + A_{HL}B_{LH} + A_{LH}B_{HL} + A_{LL}B_{HH}) \\
& + 16(A_{LL}B_{HL} + A_{HL}B_{LL} + A_{LH}B_{LH}) \\
& + 4(A_{LH}B_{LL} + A_{LL}B_{LH}) \\
& + A_{LL}B_{LL}
\end{aligned}
\tag{1}
$$

From this the mean error of a conventional approximate multiplier can be described as in Eq.(2), where i is one of the 2×2 multipliers, S_i is the shift factor of that multiplier, E_i is the error magnitude that can occur with that multiplier and $P(E_i)$ is the probability that an error will occur in that multiplier.

$$
\overline{E}_{MUL} = \sum_{i=1}^{n^2/4} |(S_i * E_i * P(E)_i)| \tag{2}
$$

2.2 Multiply Accumulate Circuits

A Multiply Accumulate (MAC) Circuit can be seen as having two stages, a multiplication stage and an accumulation stage. These stages can be clearly distinguished in Fig. 1. The multiplication stage will calculate the product of two elements in the input vector after which the accumulator stage will add this result to the already accumulated total until all elements of the input vectors

are handled. After this the MAC will output a single number. This output of a MAC is given in Eq.(3). Here M is the number of elements in the input vectors and A_m and B_m are the mth elements of the input vectors $\overrightarrow{A}$ and $\overrightarrow{B}$.

$$O_{MAC} = \overrightarrow{A} \cdot \overrightarrow{B} = \sum_{m=1}^{M} (A_m * B_m) \tag{3}$$

Using the field of approximate computing both stages can be reduced in cost by using approximate multipliers and approximate adders. This work solely focuses on the implementation of approximate multipliers in a MAC and all used adders in this work are accurate.

2.3 Self-Healing Techniques

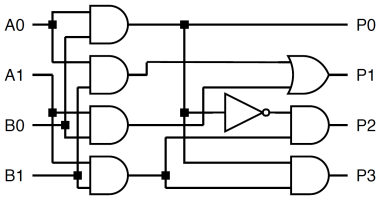
Using an approximate multiplier, the quality of the result is decreased in sacrifice for a more efficient circuit. In case of a MAC, errors will start to accumulate and the output will start to deviate from the accurate answer. The mean error of a MAC is described in Eq.(4).

$$\begin{aligned} \overline{E}_{MAC} &= \left| \sum_{m=1}^{M} \overline{E}_{MUL} \right| \\ &= \left| \sum_{m=1}^{M} \sum_{i=1}^{n^2/4} (S_i * E_i * P(E)_i) \right| \\ &= M \left| \sum_{i=1}^{n^2/4} (S_i * E_i * P(E)_i) \right| \end{aligned} \tag{4}$$

An important aspect to note is that the absolute value is taken *after* the summation of all the multiplications, which differs from Eq.(2) for just a multiplier. This is because the accumulation stage offers an opportunity to reduce the total error made by a MAC[4]. This Self-Healing effect occurs when the multiplier stage makes errors with opposing signs, which then can cancel (part of) another error in the accumulator stage. To achieve this effect, more types of approximate elementary 2×2-bit multipliers need to be used. For instance for the multiplier M2, which makes the error $3 \times 3 = 7$ the multiplier M3 from [4] can be introduced, which makes the error $3 \times 3 = 11$ and thus has an error of $+2$ that can balance the -2 error of M2. The circuit for M3 and the corresponding truth table can be found in Fig. 7.

2.3.1 Internal Self-Healing

Using multiple elementary approximate multipliers with opposing errors a circuit can be build that can achieve the Self-Healing effect. One option proposed



(a) Circuit Layout

B \ A	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	1011

(b) Truth Table

Figure 7: Approximate multiplier M3[4] and the corresponding truth table[5]

by [5] is to configure a larger multiplier to use both types of elementary multipliers inside of its multiplication stage. The created multiplier stage will then functions as is described in Fig. 8, where both errors of magnitude $+\delta$ and $-\delta$ will occur, which will cancel each other in the accumulator stage.

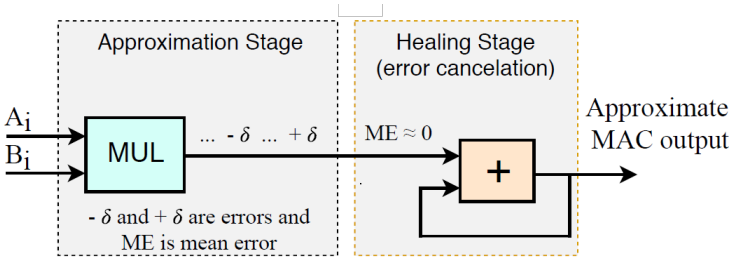


Figure 8: Internal Self-Healing methodology[5]

For example, looking at Fig. 6 again, if the multiplier $A_{LH} * B_{LL}$ is set to use M2 (which has an error magnitude of -2) and $A_{LL} * B_{LH}$ is set to use M3 (which has an error magnitude of +2), the errors can cancel each other once the results get accumulated. In a situation where the input vectors would have infinite elements that are uniformly distributed, Eq.(5) would then hold, where the mean error of the MAC becomes zero since the sum of the multiplier errors becomes zero.

$$\begin{aligned} \overline{E}_{MAC} &= M\left|\sum_{i=1}^{n^2/4}(S_i * E_i * P(E)_i)\right| = 0 \\ \implies &\sum_{i=1}^{n^2/4}(S_i * E_i * P(E)_i) = 0 \end{aligned} \tag{5}$$

2.3.2 Mirror Self-Healing

The technique suggested in [4] proposes to instead of balancing the error in one multiplier internally, an additional multiplier is constructed which produces the same error, but with an opposite sign. When the results of the approximate multiplier and the "mirror" multiplier are added by the accumulator section, errors would be canceled. The technique is also described in Fig. 9. The mathematical description of the error for this technique is given in Eq.(6), where i and j are the different elementary 2×2 multipliers in the two larger multipliers.

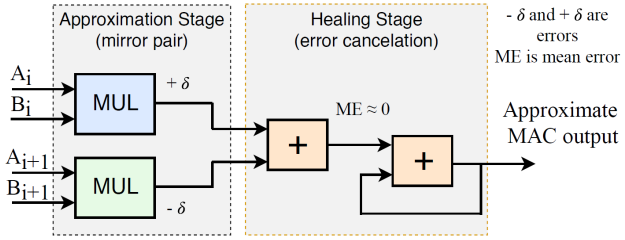


Figure 9: Mirror Self-Healing methodology[5]

$$
\begin{aligned}
\overline{E}_{MAC} &= \left| \sum_{m=1}^{M/2} \left(\overline{E}_{MUL1} + \overline{E}_{MUL2} \right) \right| \\
\overline{E}_{MAC} &= \left| \sum_{m=1}^{M/2} \left(\sum_{i=1}^{n^2/4} (S_i * E_i * P(E)_i) + \sum_{j=1}^{n^2/4} (S_j * E_j * P(E)_j) \right) \right| \\
&= \frac{M}{2} \left| \left(\sum_{i=1}^{n^2/4} (S_i * E_i * P(E)_i) + \sum_{j=1}^{n^2/4} (S_j * E_j * P(E)_j) \right) \right|
\end{aligned} \tag{6}
$$

In the case when the input vectors would have infinite elements that are uniformly distributed, Eq.(7 would hold where $\overline{E}_{MUL1}$ would converge to δ and $\overline{E}_{MUL2}$ would converge to $-\delta$, which would result in the total mean error of the MAC to reduce to zero.

$$
\begin{aligned}
\overline{E}_{MUL1} &= \sum_{i=1}^{n^2/4} (S_i * E_i * P(E)_i) = +\delta \\
\overline{E}_{MUL2} &= \sum_{j=1}^{n^2/4} (S_j * -E_j * P(E)_j) = -\delta \\
\implies \overline{E}_{MAC} &= \delta - \delta \\
&= 0
\end{aligned} \tag{7}
$$

Since two multipliers are used, the resources needed for the multiplier stage doubles in comparison with the internal self-healing technique. However, since the elements of the input vectors are divided over the two multipliers the throughput is also doubled, resulting in the same performance as two internal self-healing MACs in parallel.

2.4 Error Analysis

To analyze the two self-healing techniques there are different measures for quality that can be used. A straightforward error metric is the mean error which is given in Eq.(8) where $O_{acc}(n)$ is the kth accurate output of a MAC and $O_{app}(n)$ the approximate output.

$$\text{ME} = \frac{1}{K}\sum_{k=1}^{K}|O_{acc}(k) - O_{app}(k)| \tag{8}$$

Another commonly used error metric is the Mean Squared Error (MSE). Because of the square the MSE is less forgiving to larger errors than it is to smaller errors resulting in more distinction between different designs. The MSE is calculated as given in Eq.(9).

$$\text{MSE} = \frac{1}{K}\sum_{k=1}^{K}(O_{acc}(k) - O_{app}(k))^2 \tag{9}$$

A problem with the mean error and mean squared error is that the outcome highly depends on how large the results from the MAC are. A MAC which uses an input vector of 200 elements with a uniform distribution will have a much smaller output than a MAC which uses an input vector of 1000 elements with that same distribution. To be able to still compare these results, another error metric is introduced, the mean percentage error.
The mean percentage error (MPE) is calculated as shown in Eq.(10). Since every error is divided by the accurate result the size of the result is compensated for.

$$\text{MPE} = 100 * \frac{1}{K}\sum_{k=1}^{K}\frac{|O_{acc}(k) - O_{app}(k)|}{O_{acc}(k)} \tag{10}$$

3 Method

To be able to compare the different techniques, they are evaluated in the same setup. The conventional approximation technique, the internal self-healing technique and the mirror self-healing technique are implemented in a MAC with 2 multipliers in the multiplier stage, the results of these multiplications are added and given to the accumulator. This setup can be seen in Fig. 10. Where the techniques differ in setup is in the configurations of the multipliers. For both the conventional approximate computing methodology (Where approximate multipliers are used without a possibility for error balancing) and the internal self-healing methodology the two multipliers will use the exact same configurations, M_c and M_{ISH} in Fig. 10. The mirror self-healing methodology however will have one multiplier configured with multipliers used by the conventional methodology (M_c) and will mirror that multiplier in the second multiplier used(M_c'). Using this setup, all techniques will use the same amount of multipliers.

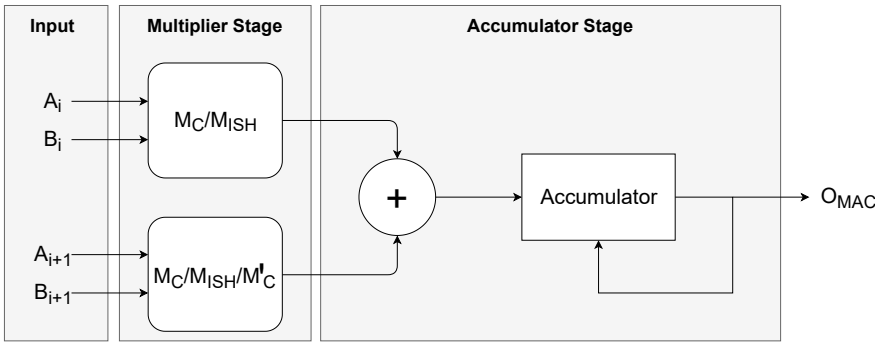


Figure 10: MAC design under evaluation

To evaluate the techniques, a model is build in MATLAB that outputs figures for quality against hardware area and quality against power usage for a certain chosen multiplier configuration. A block diagram of the functioning of the model is given in Fig. 11. The model computes these figures for quality from parameters given as input, such as the chosen input distribution, the amount of elements in the input vector and the configuration of the elementary multipliers. From these inputs the model computes an estimation for the used hardware area and power consumption for FPGA, based on parameters synthesized by the Quartus tooling and quality metrics as described in section 2.4 are calculated.

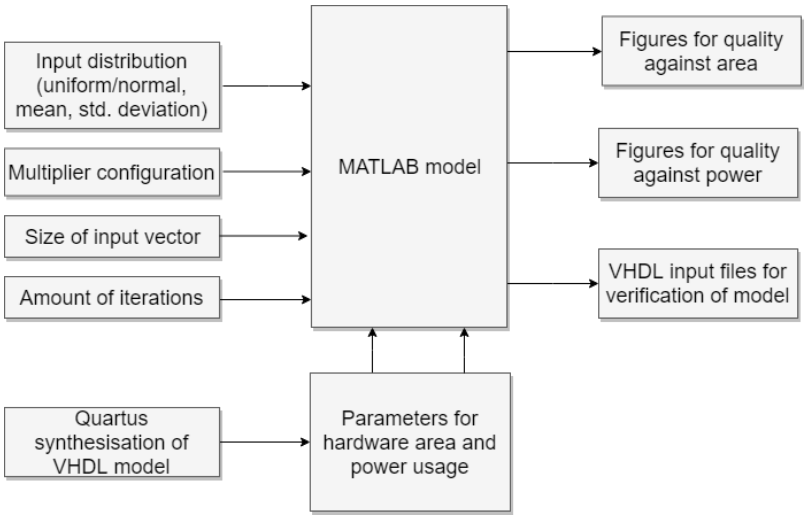


Figure 11: Block overview of the used MATLAB model

3.1 Multiplier Creation

For this work a total of seven multiplier designs are used, one accurate and six approximate. The error cases of these designs can be seen in Table 1. The circuits and corresponding truth tables can be found in appendix A. M1[7] is a multiplier with three error cases of magnitude -1, M2[6] is the multiplier already described in section 2.1. These two multipliers are often used for building conventional approximate multipliers, where there is no error balancing. M3 is introduced by [4] as a mirror multiplier for M2 as discussed in section 2.3. M4 is the multiplier introduced by [5] to compensate the bit overflow in adders that M3 can cause.
The mirror self-healing methodology as introduced by [4] works in that for every conventional n×n approximate multiplier in a MAC that is constructed, another n×n approximate multiplier is constructed that exactly mirrors the error of the first multiplier, as described in Fig. 9. To mirror M2, M3 is already introduced in [4], but to apply this methodology on approximate MAC systems also containing M1 and M4, M5 and M6 are introduced int his work. Through the use of a technique described by [8] using multiple Karnaugh maps Boolean expressions are formed. This results in two multipliers M5 and M6 that introduce mirror errors of M1 and M4 respectively.

Type	Errors
M1	$1*1 \rightarrow 0; 3*1 \rightarrow 2; 1*3 \rightarrow 2$
M2	$3*3 \rightarrow 7$
M3	$3*3 \rightarrow 11$
M4	$3*3 \rightarrow 5$
M5	$3*3 \rightarrow 13$
M6	$1*1 \rightarrow 2; 3*1 \rightarrow 4; 1*3 \rightarrow 4$
M7(M_{acc})	none

Table 1: Elementary 2×2 multipliers used

3.2 Overflow in the multiplier

When introducing approximate circuits which can have a higher maximum result than the accurate case, like M3 and M5, the problem arises that within a larger multiplier (build of these elementary 2×2 multipliers) overflow can occur. For instance if a 4×4 multiplier is build using only the M3 multiplier, the maximum output that can occur is 275, which exceeds the maximum for an 8-bit number, which is 255. Therefore this should be taken into account when picking designs as designs that overflow are not usable.

3.3 Design Space Exploration

To see which self-healing technique functions better, the best configurations for these techniques are needed. To find these configurations the design space exploration algorithm from [5] is used. This algorithm will evaluate every possible configuration from a given set of multiplier types and will output the Pareto optimal configurations that have the smallest mean error as calculated by Eq.(2) for a given area or power consumption. The algorithm can calculate these Pareto optimal configurations for conventional approximate multipliers and for approximate multipliers that use the internal self-healing technique. The algorithm also computes if a configuration will have the possibility to overflow, and if so, it will discard that configuration.

3.4 Cost Analysis using Quartus

To get figures for the cost in hardware area and power consumption on an FPGA the Quartus tooling is used. The VHDL code can be found in appendix B. This VHDL code is largely based on the work of [3], where the code for a MAC with configurable multipliers was already created. The addition made to this code enables it to synthesize a configurable MAC with two 8×8 multipliers complying to the mirror self-healing methodology. The Register Transfer Level (RTL) view of the synthesis can be seen in appendix C. Another addition is the code for the newly created multipliers from section 3.1.
The outcomes of these simulations are discussed in section 4. From these outcomes for hardware area and power consumption parameters are obtained that

are used in the MATLAB model to create cost estimations for multiplier configurations.

3.4.1 Area Calculation

In Table 2 the hardware area costs are shown for an FPGA of the Cyclone IV E family, EP4CE115F29C8. To find the hardware area for all the types of multipliers used in the MATLAB model, a MAC is synthesized that always uses a multiplier section consisting of 2 8-bit multipliers. This means that in total 32 elementary 2×2 multipliers are used. The area used by the larger multipliers consists of combinational logic and adders. Synthesis optimizes the resource usage of the setup and therefore is not always using exactly the same amount of logic cells, which is the hardware building block of an FPGA, for every elementary multiplier. To compensate for this deviation an 8-bit MAC is synthesized which uses only 1 type of multiplier. From the area report of that MAC the amount of logic cells used for the accumulator section is subtracted, leaving the amount of logic cells used by the 2 multipliers. This number is divided by 32 to conclude the value for 1 multiplier. In this value the accompanying adder logic is also taken into account. The areas are also shown for the accumulator section. For the MATLAB model an average of all the accumulator areas is used as a parameter.

Type	Multiplier Section	Accumulator Section	Total Area	Average area per multiplier
M1	286	49	335	8.9375
M2	226	51	277	7.0625
M3	294	48	342	9.1875
M4	226	51	277	7.0625
M5	294	49	343	9.1875
M6	270	48	318	8.4375
Macc	302	48	350	9.4375

Table 2: Hardware area of different multiplier types expressed in logic cells

3.4.2 Power Analysis

Power analysis is performed using both Quartus and ModelSim. The process is based on the tutorial given in [9]. Using a file with input vectors created by the MATLAB model and a circuit timing file produced by the Quartus synthesis, ModelSim simulates the functioning of the system. ModelSim creates a file with all the signal toggles that occurred when handling the input vector. Quartus can read this file and create a power consumption report. As with the results for area, the simulations are performed on an 8-bit MAC with a multiplier section consisting of only one type of elementary multiplier. The power simulations are performed four times for each multiplier type. Each time with either an input

vector with 496 elements or 4960 elements and a uniform or normal input distribution with mean 128 and a standard deviation of 40. The clock speed is set to 50 MHz, which came close to the maximum clock speed Quartus indicated was possible for this circuit on this FPGA. The results for power usage with the different input vectors can be found in appendix E. In Table 3 the results are shown which were computed with an input vector of 4960 elements, uniformly distributed. The results shown are the dynamic power consumption numbers as computed by Quartus, meaning that these results report on the power consumption caused by switching activities of the circuit when it is performing calculations. Therefore certain aspects like static power consumption to keep the FPGA running are not taken into account, which gives us a clearer view on the functioning of the circuit configuration.

Type	Multiplier Section	Accumulator Section	Total Dynamic Power	Average Power per Multiplier
M1	1.33	1.15	2.48	0.04156
M2	1.26	1.09	2.35	0.03938
M3	1.52	1.10	2.62	0.04750
M4	1.46	1.19	2.65	0.04563
M5	1.52	1.24	2.76	0.04750
M6	1.38	1.18	2.56	0.04313
Macc	1.51	1.23	2.74	0.04719

Table 3: Power usage (in mW) of different multiplier types, computed with an input vector of 4960 uniformly distributed elements.

From these results a few observations are made. When comparing the results for different input distributions in appendix E it can be seen that the difference between results obtained from uniform and normal distributions is minimal. The consistency in the results holds not only in case of a uniform or normal distribution, but also for the different input sizes. The deviation in power usage between different types of multipliers tends to be bigger than the deviation in power usage between the different sizes of input vectors, which can be seen as a positive indication for the accuracy of the results.

Another observation is that unlike the situation for area, the influence of the accumulator section on the total result is much higher for the case of power usage. For example, the accumulator section for M1 in terms of hardware area takes up 14.6% of the total area. In the case of power, the dynamic power usage of the accumulator section of M1 takes up 46% of the total dynamic power usage of the MAC.

3.5 MATLAB Model

The MATLAB model calculates the quality of a MAC configuration and plots that quality against the hardware area or power usage of that configuration. The code for the entire model can be found in appendix D.

3.5.1 Error Analysis

The functioning of model is also described in pseudo-code in algorithm 1. The model creates inputs in a desired distribution and uses these inputs to calculate the output of a MAC system. For the multiplication section a library from [1] is used that calculates the multiplication result using approximate multipliers. The outputs of the MAC are compared to the accurate result and the error metrics as described in 2.4 are applied. These error metrics are then plotted against an estimation of the hardware area and power consumption.

Algorithm 1 Pseudo-code for quality evaluation

Input:

VecAmount: The amount of input vectors that are made, increasing this number increases the accuracy of the result.

VecSize: The amount of elements in the input vectors.

Config: The configuration of the elementary multipliers in the MAC, given in a matrix with one configuration on every row.

InputType: Whether the configuration should be evaluated with a uniform or normal input distribution.

Mean: The mean to use when evaluating with a normal distribution.

Dev: The standard deviation to use when evaluating with a normal distribution.

Mirrored: Whether the MAC uses the mirror Self-Healing methodology or not.

Variables:

Area: Vector with the hardware area costs calculated by the model for every configuration.

Power: Vector with the power usage costs calculated by the model for every configuration.

Input: Variable to store the created inputs.

App_Result: Vector that accumulates the outputs of the approximate multiplier.

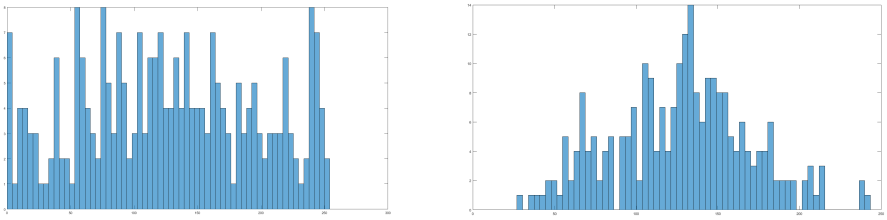
Acc_Result: Vector that accumulates the outputs of the accurate multiplier.

```
1: for i = {1,...,rows(Config)} do
2:     Area(i) = CalculateArea(Config_row(i),Mirrored)
3:     Power(i) = CalculatePower(Config_row(i),Mirrored)
4:     for j = {1,...,VecAmount} do
5:         if InputType = 'Uniform' then
6:             Input = CreateUniformInput(VecSize)
7:         else
8:             if InputType = 'Normal' then
9:                 Input = CreateNormalInput(VecSize, Mean, Dev)
10:        End if
11:        for k = {1,...,VecSize} do
12:            App_Result(j) = App_Result(j) +
13:            App_Multiplication(Config_row(i), Input(k), Mirrored)
14:            Acc_Result(j) = Acc_Result(j) + Acc_Multiplication(Input(k))
15:        End for
16:    End for
17:    ME(i) = Calculalte_ME(App_Result, Acc_Result)
18:    MPE(i) = Calculalte_MPE(App_Result, Acc_Result)
19:    MSE(i) = Calculalte_MSE(App_Result, Acc_Result)
20: End for
21: Plot(ME, MPE, MSE, Area)
22: Plot(ME, MPE, MSE, Power)
```

3.5.2 Input Creation

The quality of an approximate multiplier is dependant on the distribution of the input it receives. Uniform and normal distributed inputs are created to evaluate a MAC on how it behaves in different circumstances since the input vectors in practical situations will often have a certain distribution. This can highly effect the performance of a MAC since different multipliers handle different sections of the input numbers. For instance looking at Fig. 6 again, if a normal distribution is used with a mean of 128, the probability that a multiplier calculating the highest significance multiplication, so $A_{HH} * B_{HH}$, will have to multiply $3 * 3$ is very slim. Making it less prone to errors if an approximate multiplier is used in that position.

For the creation of input vectors the model uses the function *randi* and *randn*, where *randi* creates random uniformly distributed integers in a certain number range and *randn* creates standard normal distributed random numbers. The standard normal distribution has a mean of 0 and a standard deviation of 1. These random numbers are multiplied by the desired standard deviation given as input to the model and shifted to create a distribution that has a mean value as that specified as input. Since *randn* does not output integers, but only integer values are used in the system, the numbers are rounded to the nearest integer. Any value that is out of the range of the multiplier is rounded to either 2^n for an n-bits multiplier, or to 0, depending on what side the number is out of range. Examples of a uniform distributed input and normal distributed input generated by the model for an 8-bit MAC can be seen in Fig. 12, where the normal distribution has a mean of 128 and a standard deviation of 40. These values are chosen to ensure that almost all of the generated numbers are within the bounds but the spread is still enough to not lay the focus too much on a single section of the multiplier. In this example 248 elements are created in this input vector. Four of these input vectors are used in the model as input to the two multipliers in the multiplication stage. This to create the same situation as if the MAC would handle 2 input vectors of 496 elements which would get divided over the two multipliers as in the case for radio astronomy applications [1]. As can be seen from the figures, the distribution is not smooth, since it is for a limited number of elements. This is to simulate practical scenarios where there also will be limited inputs from a non-perfect distribution. This aspect introduces the need for the model to perform the MAC operation multiple times to create an accurate answer.



(a) Uniform Distribution

(b) Normal Distribution

Figure 12: Distribution of input vectors with 248 elements to simulate radio astronomy applications [1]

3.6 Model Verification

To ensure the validity of the created MATLAB model, verification using Modelsim is performed. The MATLAB model can output automatically generated VHDL, code where input vectors created by the MATLAB model are written down. This VHDL testbench file is then opened in Modelsim and executed on the VHDL model of the MAC. This way, the MATLAB model and the model in Modelsim will receive the exact same inputs. The output of Modelsim is then compared to that of MATLAB to verify it behaves in the same way.

4 Results

4.1 Area and power of 2×2 multipliers

Through Quartus synthesis results are found for the amount of hardware area used by the different multipliers in a MAC setup. A table of the results is already shown in section 3.4.1. These average area results are also graphically displayed in Fig. 13.

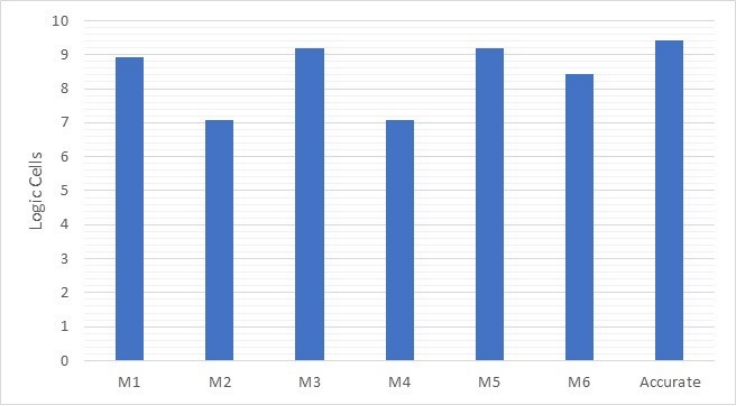


Figure 13: Area in logic cells of different multiplier types

These results show that all constructed multipliers use a smaller amount of logic cells in comparison with the accurate multiplier with M2 and M4 standing out as having the lowest area. M3, M5 and M6 use a significantly larger area, but are used as mirror pairs for other multipliers and therefore are still relevant. The mirror pair of M1 and M6 are different in that the circuit produces a low error magnitude but with a higher error frequency and therefore interesting as comparison to the other circuits which have a high error magnitude but a low error rate.

For power, a graphical representation of the performance of the different types of multipliers can be found in Fig. 14. In contrary to the results for area, not all multipliers use less power in comparison with an accurate multiplier; M3 and M5 exceed the power usage of the accurate multiplier by a small amount. It can be seen that the multiplier M2 again has the lowest result. M4 performs less in power usage than it did for area, but it is still superior to the accurate case.

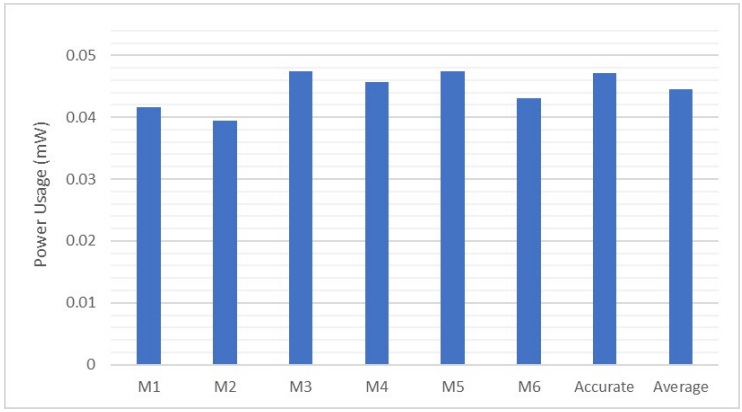


Figure 14: Power in mW of different multiplier types

4.2 Error Analysis of Self-Healing Configurations

Using the created MATLAB model several configurations are evaluated. Pareto optimal configurations are found using the design space exploration algorithm designed by [5]. This algorithm outputs configurations for conventional approximate multipliers and configurations for the Internal Self-Healing methodology. On the configurations for the conventional approximate multiplier the mirror self-healing methodology is applied and thus every approximate multiplier is mirrored. For example, if a conventional configuration for a 4×4 multiplier would be [M2,M2,M1,M4], the configuration for the mirror multiplier would then be [M3,M3,M6,M5]. This results in configurations for the three methodology's: Conventional, Internal Self-Healing and Mirror Self-Healing.

The elementary multiplier parameters found by the Quartus synthesis as discussed in section 4.1 are implemented into the algorithm. Configurations are found that have the lowest analytic mean error for either hardware area or power consumption. Configurations are either optimized for a uniform distribution or a normal distribution. The resulted configurations can be found in appendix F.

The results of the quality evaluation for all the evaluated configurations can be found in appendix H. In this section, a selection of these plots are discussed. In Fig. 15 an enlarged graph can be seen of a single result plot. This graph shows the results for Pareto optimal configurations computed for the least area and using a uniform input. The vector size in this case is 4960.

Overflow configurations

A very important thing to note in Fig. 15 is that many designs of the mirror self-healing technique will cause an overflow. In the graph the distinction is made between designs that overflow (black) and designs that do not (magenta). These overflow designs are caused by the fact that the mirror self-healing technique exactly mirrors the conventional healing configurations. The conventional healing technique can use multiplier types M1, M2 and the accurate multiplier. But since the M1 multiplier is using significantly more resources than M2 the

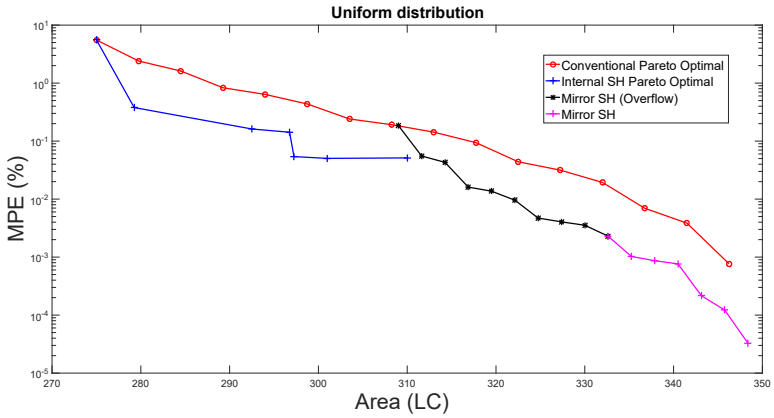


Figure 15: Results optimized for area and uniform input distribution

Pareto optimal configurations use only M2 and the accurate multiplier. This results in that the mirror self-healing configurations use only M3 and the accurate multiplier, and creates many cases where there are too many M3 multipliers in one 4×4 multiplier creating the possibility for overflow as described in section 3.2. To remedy this problem for these configurations it is possible to switch certain elementary multipliers between the two large multipliers. For instance if a certain 4×4 multiplier in the conventional setup uses only M2 multipliers and thus the mirrored 4×4 multiplier uses only M3 multipliers, one can switch the most significant elementary multiplier, resulting in 1 multiplier that uses the configuration [M3,M2,M2,M2] and one that uses [M2,M3,M3,M3] and both 4×4 multipliers will not be able to overflow. The result of a system with those configurations does not change in comparison. Doing this however is deviating from the methodology as proposed in [4]. Another option to gain more insight of the mirror self-healing methodology is to create a design space exploration specifically for the mirror self-healing technique, which would also filter out designs that overflow.

minimal cost for mirror self-healing technique
Interesting to note in Fig. 15 is that the mirror self-healing methodology will always have a relatively high minimal area it uses, this is caused by the fact that every multiplier is mirrored and the mirror multiplier types (M3,M5 and M6) have a higher cost. The minimal area configuration that this technique can achieve is a configuration where one 8-bit multiplier is completely build of M2 multipliers, and therefore the other 8-bit multiplier will be completely build of M3 multipliers.

Comparison of the self-healing techniques From the results the conclusion can be drawn that both the internal self-healing and the mirror self-healing outperform the conventional approximate computing technique. Interesting to see is that the two self-healing techniques work in their own regions. The internal self-healing technique finds the Pareto optimal configurations for the smallest

mean error as described in Eq.(2) in section 2.1 for a certain area. This means that as soon as a multiplier configuration is fully balanced it has mean error = 0. In Fig. 15 this is the point where the internal self-healing (blue) line stops. The mirror self-healing however always achieves this mean error = 0, since every conventional approximate multiplier will get mirrored.
In Fig. 16 the results are shown not for Pareto optimal configurations, but for configurations where the mean error as described in Eq.(2) is always 0. I.e., every approximate multiplier will get mirrored with another multiplier with the same significance. The configurations shown are increasing in approximation and use only the multipliers M2, M3 and the accurate multiplier. The exact configurations can be found in appendix G. In this case it is clearly visible that the mirror self-healing technique is superior to the internal self-healing technique. The reason for this is that with the mirror technique absolute self-healing can already happen with the least significant multiplier (multiplier $A_{LL} * B_{LL}$ in Fig. 6). This gives the mirror technique an advantage as it can gain the same area reduction as the internal self-healing technique, but can place the approximate multipliers on lower significant places.

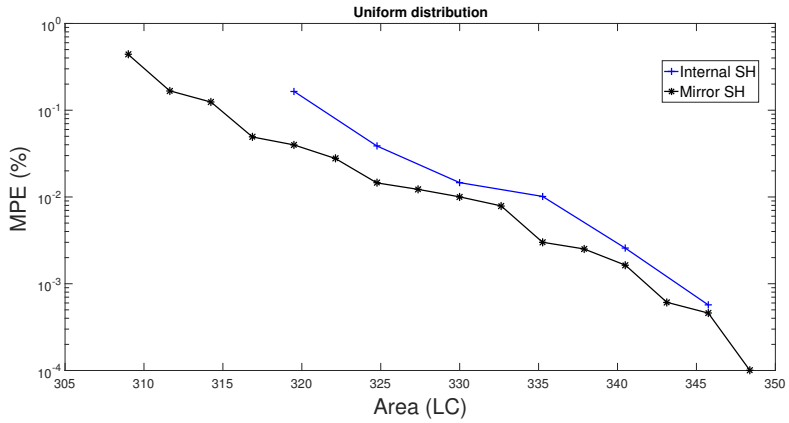


Figure 16: Results for configuration with increasing approximation, Vector size: 496

Difference in Area and Power Fig. 17 shows the same circumstances as Fig. 15, only using configurations optimized for power usage. From comparing these two figures it can be seen that the results for area and power are quite alike in shape, and also the found Pareto optimal configurations do not differ drastically. The main cause for this is that like in the results for area, multiplier M2 still is the most efficient for power and is therefore used a lot.

The significance of the vector size
In Fig. 18 the results can be seen as computed for different vector sizes. The vector sizes are 496, 496∗4 and 496∗10 elements. It can be clearly seen that with the increase of the vector size, the effectiveness of both the self-healing

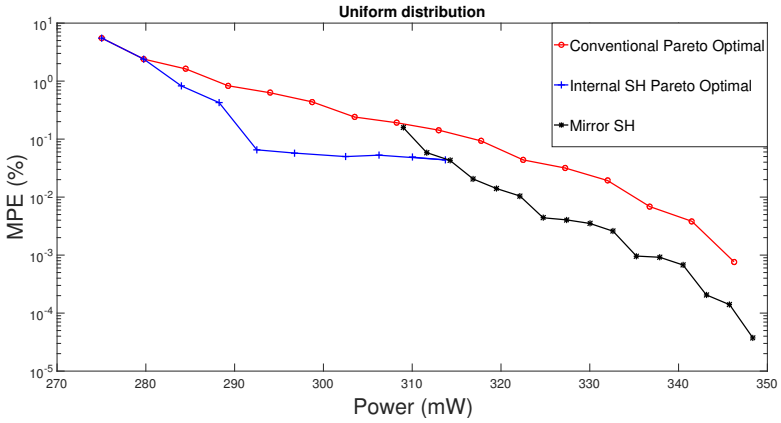
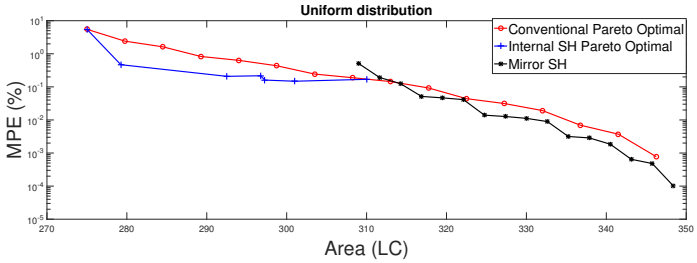
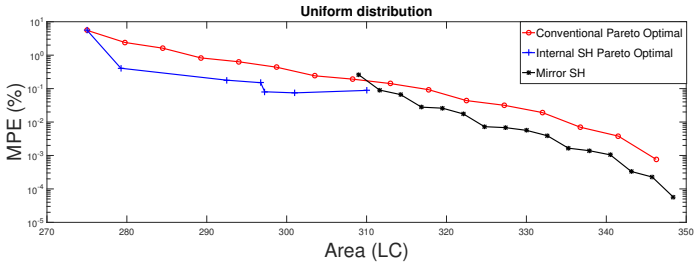


Figure 17: Results optimized for Power and uniform input distribution

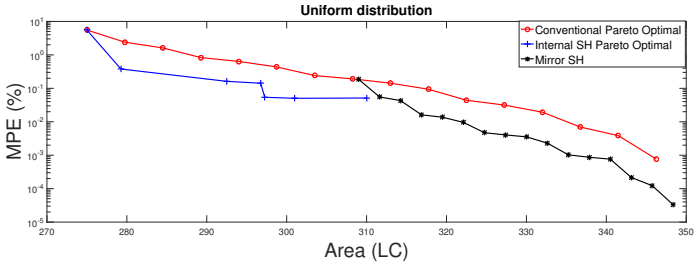
techniques increases. This is logical since the input distribution is becoming more uniform when there are more elements in the vector and when the input is becoming more uniform more errors will cancel each other more evenly.



(a) Vector Size: 496



(b) Vector Size: 1984



(c) Vector Size: 4960

Figure 18: MPE results for different vector sizes with a uniform distribution

5 Conclusion

In this work, two different techniques for self-healing in a MAC applied on an FPGA have been compared. From the results in section 4 can be seen that although both techniques have their advantages, the mirror self-healing technique as described in [4] comes with a lot of restrictions. To start, the technique is only usable in cases where there is already a need for 2 MACs to be used in parallel, otherwise using this technique would already almost double the hardware area needed. Second, the chance of a mirrored multiplier to have the possibility to overflow is quite high, rendering a lot of configurations unusable if applying the exact methodology of mirroring a conventional approximate multiplier. Third, because of this method of always mirroring, the minimal hardware cost that can be achieved by this technique is relatively high. An advantage that the mirror self-healing technique possesses is the ability to balance errors with using only the lowest significant multipliers. However, in general the quality of the results produced by the mirroring technique are not able to compensate for the restrictions it has.

Evaluating different aspects of the MAC gave the conclusion that there is no unexpected behaviour change when evaluating a MAC on either hardware area or power consumption, for either a uniform or normal distribution. All Pareto optimal configurations had no significant difference in what type of multipliers were used and the quality evaluations of the techniques gave very similar results.

6 Discussion

Evaluating the results questions arise that were not addressed in the time permitted. These subjects are treated below.

Cost results for FPGA

In section 4.1 the hardware costs of a MAC on an FPGA are discussed. From these results it became apparent that the costs for multipliers on FPGA do not correspond that well with those for ASIC as discussed in [4] and [5]. For example, the newly created multiplier M6 has a lower area cost than that of multiplier M3, but uses far more gates if it would be build in ASIC and therefore use more hardware area.

Overflow in the multipliers

As discussed in section 4.2 a lot of the found configurations using the mirror self-healing technique are able to overflow. This problem arises because the design space exploration used is build for finding Pareto optimal configurations for the conventional and internal self-healing techniques without overflow cases. Using the conventional configurations and mirroring them is the technique as described in [4]. However, this technique is build for a Square Accumulator (SAC) and within a SAC bit overflow is a much smaller problem since a SAC utilizes a number of 2×2 squarers which compensate for overflow. Using this exact same technique on a MAC therefore is much harder. The technique can however be adjusted, as discussed in the **overflow configurations** part of section 4.2.

Comparing the different self-healing techniques

When comparing the internal self-healing technique and the mirror self-healing technique the problem arises that the mirror self-healing methodology is severely limited in area reduction by the fact that every approximate multiplier should be mirrored. This increases the minimal hardware costs when using this technique and can more easily result in configurations with bit overflow.

7 Future Work

In the field of approximate computing there is still a lot of room for future investigations, focusing on the subjects treated in this work, the following items are still worth investigating:

- As mentioned in section 6, optimization for FPGA can still improve a great deal. The conventional methodology uses certain approximate multipliers because they use less gates and therefore make less hardware cost, which makes sense when working in a ASIC environment. For an FPGA environment however, it does not, and further investigation could be done on which multiplier setups are more efficient for FPGA. This was out of scope for this work, since the goal is to compare the self-healing techniques as they were presented.

- The mirror self-healing technique is severely limited in performance since the methodology restricts itself to only exactly mirror a conventional multiplier, creating problems like bit overflow and a high minimal resource cost. In future work, an adjusted methodology could be developed that works around these restrictions by not simply mirroring the conventional multiplier, but adjust both multipliers in a way that the multipliers balance each other but do not overflow.

- Since the design space exploration algorithm used in this work only computed the Pareto optimal configurations for the conventional technique and the internal self-healing technique, not all possible mirror self-healing options are evaluated and overflow cases are not filtered out. In future work, the design space exploration algorithm could be expanded to also consider mirror self-healing configurations, which will allow a better comparison between the different techniques.

- This work focused on comparing the internal and mirror self-healing techniques, but an interesting concept is what the results would be if both techniques were combined, where multipliers are created that are both balanced internally and with a mirror multiplier in a pair.

References

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

[2] L. D'Addario and D. Wang. "An integrated circuit for radio astronomy correlators supporting large arrays of antennas," Journal of Astronomical Instrumentation, 2016.

[3] B. Verstoep. "Approximate multipliers for MAC," Bachelor assignment, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science, April 2018.

[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, 6, 49112-49128. 2018.

[5] G. A. Gillani et al. "MACISH: Designing Approximate MAC Accelerators with Internal-Self-Healing," [Still to be published] 2019.

[6] P. Kulkarni, P. Gupta, M. Ercegovac, "Trading Accuracy for Power with an Underdesigned Multiplier Architecture," 24th International Conference on VLSI Design (VLSI Design), pp. 346 – 351, 2011.

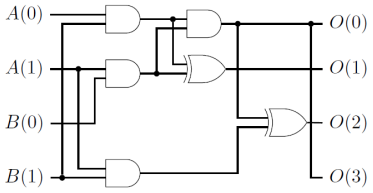
[7] Semeen Rehman, Walaa El-Harouni, Muhammad Shafique, Akash Kumar, and Jorg Henkel, "Architectural-space exploration of approximate multipliers," In Frank Liu, editor, ICCAD, page 80. ACM, 2016.

[8] A. Marcovitz. "The Karnaugh Map," in *Introduction to Logic Design*, McGraw-Hill, 2009, pp. 111-199.

[9] S. Gerez. "Power Analysis with Quartus II and Modelsim," University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science, March 2015.

Appendices

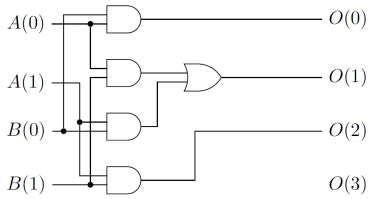
A Multiplier Circuits and Truth Tables



(a) Circuit M1

B \ A	00	01	10	11
00	0000	0000	0000	0000
01	0000	0000	0010	0010
10	0000	0010	0100	0110
11	0000	0010	0110	1001

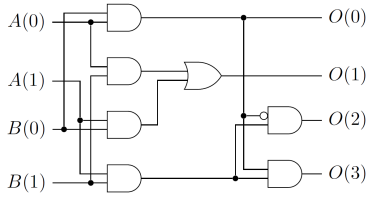
(b) Truth table M1



(c) Circuit M2

B \ A	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	0111

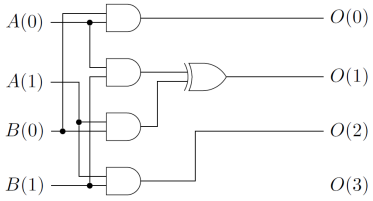
(d) Truth table M2



(e) Circuit M3

B \ A	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	1011

(f) Truth table M3

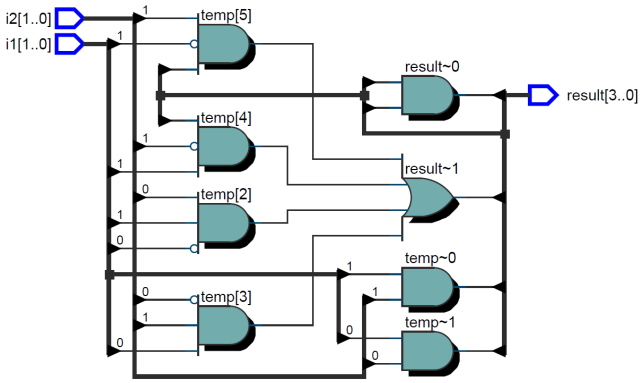


(g) Circuit M4

B \ A	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	0101

(h) Truth table M4

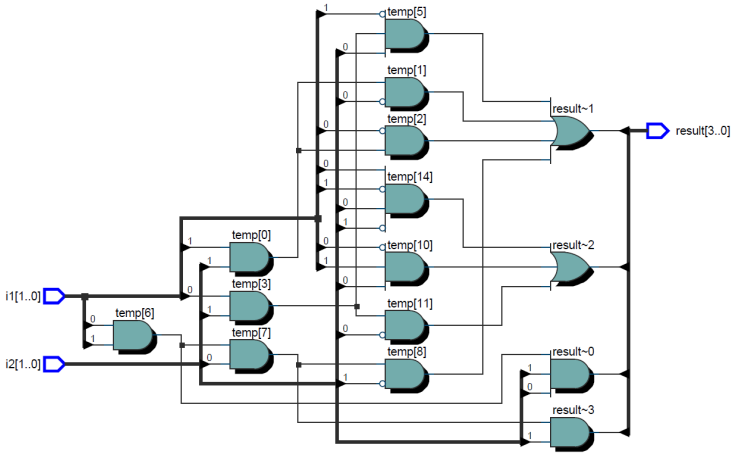
Figure 19: Circuit for M1-M4 and the corresponding truth tables[3]



(a) Circuit M5

B \ A	00	01	10	11
00	0000	0000	0000	0000
01	0000	0001	0010	0011
10	0000	0010	0100	0110
11	0000	0011	0110	1011

(b) Truth table M5



(c) Circuit M6

B \ A	00	01	10	11
00	0000	0000	0000	0000
01	0000	0010	0010	0100
10	0000	0010	0100	0110
11	0000	0100	0110	1001

(d) Truth table M6

Figure 20: Created approximate multipliers M5 and M6 as synthesized by Quartus and the corresponding truth tables

B VHDL Code

The VHDL code is based on that used by [3], with additions made so it describes a MAC with 2 8-bit multipliers and the new multipliers are added.

B.1 MAC Using 2 8-bit Multipliers

```
1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.numeric_std.ALL;
4  USE work.ALL;
5
6  entity AcceightParallelMAC is
7    port( i1, i2, i3, i4:   in std_logic_vector(7 downto 0);
8        CLK:    in std_logic;
9        result:   out unsigned(24 downto 0) --Max output when using a
       vector size of 496 needs 25 bits to be stored
10       );
11   end AcceightParallelMAC;
12
13 architecture bhv of AcceightParallelMAC is
14   COMPONENT eightbitmultiplier is              --Multiplier 1
15   port( i1, i2: in std_logic_vector(7 downto 0);
16       result: out std_logic_vector(15 downto 0)
17       );
18   end COMPONENT;
19
20 signal Ri1,Ri2,Ri3,Ri4: std_logic_vector(7 downto 0);
21 signal mul1, mul2: std_logic_vector(15 DOWNTO 0);
22 signal total, Rtotal: unsigned(24 downto 0) := (others => '0');
23
24 begin
25 --eight bit multiplier:
26 Mult1: eightbitmultiplier PORT MAP(i1 => std_logic_vector(Ri1) , i2
       => std_logic_vector(Ri2) , result => mul1);
27 Mult2: eightbitmultiplier PORT MAP(i1 => std_logic_vector(Ri3) , i2
       => std_logic_vector(Ri4) , result => mul2);
28 --sum up all inputs:
29 total <= Rtotal + resize(unsigned(mul1), 25) + resize(unsigned(mul2
     ), 25);
30
31 PROCESS(CLK)
32
33 BEGIN
34 IF rising_edge(CLK) THEN
35 Ri1 <= i1; --update input registers
36 Ri2 <= i2;
37 Ri3 <= i3;
38 Ri4 <= i4;
39 Rtotal <= total; --next clock cycle update output
40 END IF;
41 END PROCESS;
42 result <= Rtotal;
43 end architecture;
```

B.2 8-bit multiplier

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
USE work.ALL;

entity eightbitmultiplier is
  port( i1, i2: in std_logic_vector(7 downto 0);
  result: out std_logic_vector(15 downto 0)
    );
end eightbitmultiplier;

architecture eighttofour of eightbitmultiplier is
  COMPONENT fourbitmultiplier is
  port( i1, i2: in std_logic_vector(3 downto 0);
  result: out std_logic_vector(7 downto 0)
    );
end COMPONENT;

signal temp1, temp2, temp3, temp4: std_logic_vector(7 downto 0);

begin
mul1: fourbitmultiplier PORT MAP(i1 => i1(3 downto 0) , i2 => i2(3
    downto 0) ,
result => temp1); --LSB
mul2: fourbitmultiplier PORT MAP(i1 => i1(3 downto 0) , i2 => i2(7
    downto 4) ,
result => temp2); --MidSB
mul3: fourbitmultiplier PORT MAP(i1 => i1(7 downto 4) , i2 => i2(3
    downto 0) ,
result => temp3); --MidSB
mul4: fourbitmultiplier PORT MAP(i1 => i1(7 downto 4) , i2 => i2(7
    downto 4) ,
result => temp4); --MSB
result <= std_logic_vector(resize(unsigned(temp1), 16) + shift_left
    (resize(unsigned(temp2), 16),4) + shift_left(resize(unsigned(
    temp3), 16),4) +
shift_left(resize(unsigned(temp4), 16),8)); --shift the results and
     add up
end architecture;
```

B.3 4-bit multiplier

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
USE work.ALL;

entity fourbitmultiplier is
  port( i1, i2: in std_logic_vector(3 downto 0);
    result: out std_logic_vector(7 downto 0)
      );
end fourbitmultiplier;

architecture fourtotwo of fourbitmultiplier is
  COMPONENT testmultiplier is
```

```
  port( i1, i2: in std_logic_vector(1 downto 0);
  result: out std_logic_vector(3 downto 0)
  );
  end COMPONENT;

signal temp1, temp2, temp3, temp4: std_logic_vector(3 downto 0);

begin
  mul1: M?multiplier PORT MAP(i1 => i1(1 downto 0) , i2 => i2(1
    downto 0) , result => temp1); --LSB
  mul2: M?multiplier PORT MAP(i1 => i1(1 downto 0) , i2 => i2(3
    downto 2) , result => temp2); --MidSB
  mul3: M?multiplier PORT MAP(i1 => i1(3 downto 2) , i2 => i2(1
    downto 0) , result => temp3); --MidSB
  mul4: M?multiplier PORT MAP(i1 => i1(3 downto 2) , i2 => i2(3
    downto 2) , result => temp4); --MSB

result <= std_logic_vector(resize(unsigned(temp1), 8) + shift_left(
    resize(unsigned(temp2), 8) ,2) + shift_left(resize(unsigned(
    temp3), 8), 2) + shift_left(resize(unsigned(temp4), 8),4));

end architecture;
```

B.4 Approximate Multipliers

M1

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

entity M1multiplier is
  port( i1, i2: in std_logic_vector(1 downto 0);
    result: out std_logic_vector(3 downto 0)
  );

end M1multiplier;

architecture approximate of M1multiplier is
  signal temp: std_logic_vector(3 downto 0);

begin
temp(0) <= i1(0) and i2(1);
temp(1) <= i1(1) and i2(0);
temp(2) <= temp(0) and temp(1);
temp(3) <= i1(1) and i2(1);

result <= temp(2) & (temp(2) xor temp(3)) & (temp(0) xor temp(1)) &
     temp(2);

end architecture;
```

M2

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

entity M2multiplier is
```

```
  port( i1, i2: in std_logic_vector(1 downto 0);
    result: out std_logic_vector(3 downto 0)
  );

end M2multiplier;

architecture approximate of M2multiplier is
  signal temp: std_logic_vector(1 downto 0);

begin
temp(0) <= i1(0) and i2(1);
temp(1) <= i1(1) and i2(0);
result <= '0' & (i1(1) and i2(1)) & (temp(0) or temp(1)) & (i1(0)
    and i2(0));
end architecture;
```

M3

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

entity M3multiplier is
  port( i1, i2: in std_logic_vector(1 downto 0);
    result: out std_logic_vector(3 downto 0)
  );

end M3multiplier;

architecture approximate of M3multiplier is
  signal temp: std_logic_vector(3 downto 0);

begin
temp(0) <= i1(0) and i2(0);
temp(1) <= i1(0) and i2(1);
temp(2) <= i1(1) and i2(0);
temp(3) <= i1(1) and i2(1);
result <= (temp(3) and temp(0)) & (temp(3) and (not temp(0))) & (
    temp(1) or temp(2)) & temp(0);
end architecture;
```

M4

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

entity M4multiplier is
  port( i1, i2: in std_logic_vector(1 downto 0);
    result: out std_logic_vector(3 downto 0)
  );

end M4multiplier;

architecture accurate of M4multiplier is
  signal temp: std_logic_vector(9 downto 0);

begin

```

```
  temp(0) <= i1(1) and i2(1); --AC -> O2
  temp(1) <= i1(0) and i2(0); --DB -> O0
  temp(6) <= i2(0) and not i1(0);
  temp(2) <= temp(6) and i1(1); --DB'A
  temp(7) <= i2(1) and not i1(1);
  temp(3) <= temp(7) and i1(0); --CA'B
  temp(8) <= temp(1) and i1(1);
  temp(4) <= temp(8) and not i2(1); --DBAC'
  temp(9) <= temp(1) and not i1(1);
  temp(5) <= temp(9) and i2(1); --DBA'C

  result <= (temp(0) and temp(1)) & (temp(0)) & (temp(2) or temp(3)
     or temp(4) or temp(5)) & (temp(1));


end architecture;
```

M5

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

entity M5multiplier is
  port( i1, i2: in std_logic_vector(1 downto 0);
    result: out std_logic_vector(3 downto 0)
  );

end M5multiplier;

architecture accurate of M5multiplier is
  signal temp: std_logic_vector(9 downto 0);

begin

  temp(0) <= i1(1) and i2(1); --AC -> O2
  temp(1) <= i1(0) and i2(0); --DB -> O0
  temp(6) <= i2(0) and not i1(0); --DB'
  temp(2) <= temp(6) and i1(1); --DB'A
  temp(7) <= i2(1) and i1(0);  --CB
  temp(3) <= temp(7) and not i2(0); --BCD'
  temp(8) <= temp(1) and i1(1); --DBA
  temp(4) <= temp(8) and not i2(1); --DBAC'
  temp(9) <= temp(1) and not i1(1); --DBA'
  temp(5) <= temp(9) and i2(1); --DBA'C

  result <= (temp(0) and temp(1)) & (temp(0)) & (temp(2) or temp(3)
     or temp(4) or temp(5)) & (temp(1));


end architecture;
```

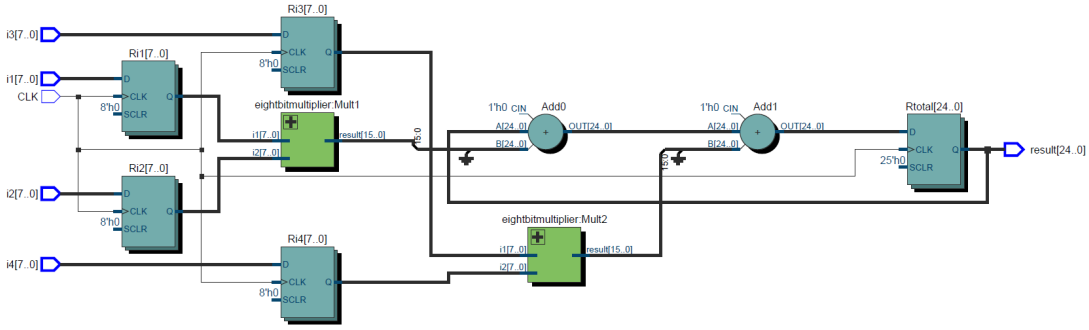
M6

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;

entity M6multiplier is
```

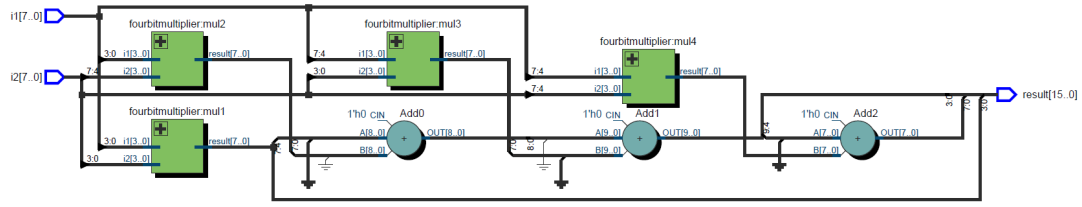
```
  port( i1, i2: in std_logic_vector(1 downto 0);
    result: out std_logic_vector(3 downto 0)
  );

end M6multiplier;

architecture accurate of M6multiplier is
  signal temp: std_logic_vector(14 downto 0);

begin
  temp(0) <= i1(1) and i2(1); --AC
  temp(1) <= temp(0) and not i2(0); --ACD~
  temp(2) <= temp(1) and not i1(0); --ACB~
  temp(3) <= i1(0)  and i2(1); --BC
  temp(4) <= temp(3) and i2(0); --BCD
  temp(5) <= temp(4) and not i1(1); --BCDA~
  temp(6) <= i1(0) and i1(1); --AB
  temp(7) <= temp(6) and i2(0); --ABD
  temp(8) <= temp(7) and not i2(1); --ABC~D
  temp(9) <= i1(1) and i2(0); --AD
  temp(10) <= temp(9) and not i1(0); --AB~D
  temp(11) <= temp(3) and not i2(0); --BCD~
  temp(12) <= i1(0) and not i1(1); --A~B
  temp(13) <= i2(0) and not i2(1); --C~D
  temp(14) <= temp(12) and temp(13); -- A~BC~D

  result <= (i1(0) and i1(1) and i2(0) and i2(1)) & (temp(1) or
    temp(2) or temp(5) or temp(8)) & (temp(11) or temp(14) or temp
    (10)) & (i1(0) and i1(1) and i2(0) and i2(1));


end architecture;
```

C RTL View

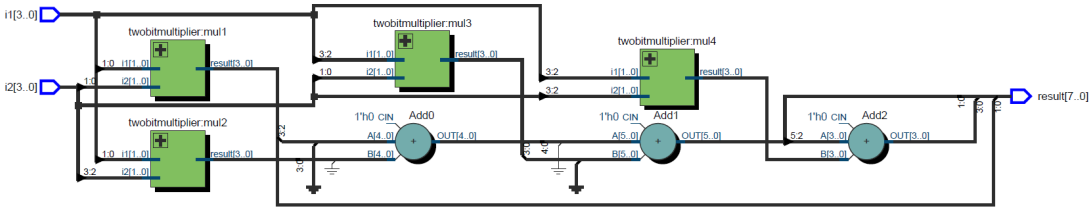
C.1 MAC Using 2 8-bit Multipliers



C.2 8-bit Multiplier



C.3 4-bit Multiplier



D MATLAB Code

D.1 Main

```
%Script to collect results of approximate multipliers using the functions
%EvaluateConfig, EvaluateMult, CreateInputs, CalculateArea, CalculatePower, ParallelMACMir and SequentialMAC, which in turn uses the
%(very slightly modified) lpACLib (Low Power Approximate Computing Library) by Vanshika Baoni and
%Muhammad Shafique.

%-------------------------------------------------------------------------

% Input arguments
% Mean: Specified mean when normal distribution is used
% Dev:  Specified standard deviation when normal distribution is used
% Inputamount: Amount of input vectors, so how many times the MAC runs
%              before the ME, MSE, and MPE are calculated.
%
% VecSize: amount of elements in input vectors, often set to 496 to
%          correspond with radio astronomy purposes. Since there are
%          always 2 N x N multipliers the amount of elements gets split
%          between the 2 sets of input vectors, so the size should be
%          divided by 2.
%
% ChosenMult: An array of chosen multipliers that will be used by the MAC.
%             Every row contains the type of multipliers for 1 MAC.
%             The multipliers are used from LSB to MSB, so the first item
%             in the row is used in the multiplier for the least
%             significant bits.




mean8bit = 128;
dev8bit = 40;
Inputamount = 100;
VecSize = 496*5;

mean4bit = 7.5;
dev4bit = 2.2;




MultPareConUniArea = ConvertMult(P_16x16_unif_config_convent_Area3);
```

```
MultPareISHUniArea = ConvertMult(P_16x16_unif_config_ish_Area3);
MirMultPareConUniArea = MirMult(MultPareConUniArea);


MultPareConNormArea = ConvertMult(P_16x16_norm_config_convent_Area3
    );
MultPareISHNormArea = ConvertMult(P_16x16_norm_config_ish_Area3);
MirMultPareConNormArea = MirMult(MultPareConNormArea);

MultPareConUniPow = ConvertMult(P_16x16_unif_config_convent_pow3);
MultPareISHUniPow = ConvertMult(P_16x16_unif_config_ish_pow3);
MirMultPareConUniPow = MirMult(MultPareConUniPow);

MultPareConNormPow =  ConvertMult(P_16x16_norm_config_convent_pow3)
    ;
MultPareISHNormPow = ConvertMult(P_16x16_norm_config_ish_pow3);
MirMultPareConNormPow = MirMult(MultPareConNormPow);


% The chosen multipliers can be defined by using a matrix. Every
    row
% inludes the multipliers for one full MAC.

%Test Multiplier configuration which increases in approximation (ME
    = 0) for 8-bit MAC
Test8bit = [7,2,3,2, 3,2,3,7, 7,2,3,2, 3,2,3,7;
                     7,2,3,2, 3,2,3,7, 7,2,3,2, 3,7,7,7;
                     7,2,3,2, 3,2,3,7, 7,2,3,7, 7,7,7,7;
                     7,2,3,2, 3,2,3,7, 7,7,7,7, 7,7,7,7;
                     7,2,3,2, 3,7,7,7, 7,7,7,7, 7,7,7,7;
                     7,2,3,7, 7,7,7,7, 7,7,7,7, 7,7,7,7;
                     7,7,7,7, 7,7,7,7, 7,7,7,7, 7,7,7,7];


%The model also works for a 4-bit MAC and a 16-bit MAC
%Test Multiplier configuration which increases in approximation (ME
    = 0) for 4-bit MAC
Test4bit = [2,2,2,3;
                      2,2,2,7;
                      2,2,7,7;
                      2,7,7,7;
                      7,7,7,7];


% Computing the results for configurations from the test
    configurations
% Uniform Input distribution
[METest8bit,MSETest8bit,MPETest8bit, AreaTest8bit, PowerTest8bit] =
    EvaluateConfig(1,mean8bit,dev8bit,VecSize,Inputamount,
    Test8bit, 1);
[METest4bit,MSETest4bit,MPETest4bit, AreaTest4bit, PowerTest4bit] =
    EvaluateConfig(1,mean4bit,dev4bit,VecSize,Inputamount,
    Test4bit, 1);

% Normal Input distribution
```

```
[METest8bitNorm,MSETest8bitNorm,MPETest8bitNorm, AreaTest8bitNorm,
    PowerTest8bitNorm] = EvaluateConfig(2,mean8bit,dev8bit,VecSize,
    Inputamount, Test8bit, 1);
[METest4bitNorm,MSETest4bitNorm,MPETest4bitNorm, AreaTest4bitNorm,
    PowerTest4bitNorm] = EvaluateConfig(2,mean4bit,dev4bit,VecSize,
    Inputamount, Test4bit, 1);


%-------------------------------------------------------------------------

%Example Plot plotting ME against area
figure(1);
ResultPlot1 = subplot(3,2,1);
plot(ResultPlot1, AreaTest8bit,METest8bit,'-or',AreaTest4bit,
    METest4bit,'-+b');
title(ResultPlot1,'Uniform distribution')
xlabel(ResultPlot1,'Area (LC)')
ylabel(ResultPlot1,'ME')
set(gca, 'YScale', 'log')
legend('8-bit multiplier', '4-bit multiplier');

ResultPlot2 = subplot(3,2,2);
plot(ResultPlot2, AreaTest8bitNorm,METest8bitNorm,'-or',
    AreaTest4bitNorm,METest4bitNorm,'-+b');
title(ResultPlot2,'Normal distribution')
xlabel(ResultPlot2,'Area (LC)')
ylabel(ResultPlot2,'ME')
set(gca, 'YScale', 'log')
legend('8-bit multiplier', '4-bit multiplier');
```

D.2 EvaluateConfig

```
function [MEConfig, MSEConfig, MPEConfig, AreaConfig, PowerConfig]
    = EvaluateConfig(distr, mean, dev, VecSize, Inputamount,
    ChosenMult, type)
%Allocating memory for the vectors
AreaConfig = zeros(1,size(ChosenMult,1));
PowerConfig = zeros(1,size(ChosenMult,1));
[MEConfig, MSEConfig, MPEConfig] = deal(zeros(1,size(ChosenMult,1))
    );

%For every configuration given in the ChosenMult matrix
for i = 1:size(ChosenMult,1)

%Calculate estimated hardware costs
AreaConfig(i) = CalculateArea(ChosenMult(i,:), type);
PowerConfig(i) = CalculatePower(ChosenMult(i,:), type);

% Calculate results for given distribution (can also output file
    with
% used inputs so verification in ModelSim can be done)
[MEConfig(i),MSEConfig(i),MPEConfig(i)] = EvaluateMult(distr,mean,
    dev,VecSize,Inputamount, ChosenMult(i,:), type);
end
```

D.3 CalculateArea

```
function AreaSize = CalculateArea(ChosenMult, type)
%Calculating dimension of the configuration under evaluation
dim = 2*sqrt(size(ChosenMult,2));

%Find the amounts used of each multiplier type
acc = find(ChosenMult == 7);
approx1 = find(ChosenMult == 1); %Multiplier that has 3 error cases of 1 magnitude
approx2 = find(ChosenMult == 2); %Multiplier that has 1 error case of 2 magnitude (3*3 = 7)
approx3 = find(ChosenMult == 3); %Mirror pair of approx2 so 3*3 = 11
approx4 = find(ChosenMult == 4); %Multiplier that has 1 error case of 4 magnitude (3*3 = 5)
approx5 = find(ChosenMult == 5);
approx6 = find(ChosenMult == 6);

%%Areas of different multiplier types
M1 = 8.9375;
M2 = 7.0625;
M3 = 9.1875;
M4 = 7.0625;
M5 = 9.1875;
M6 = 8.4375;
MAcc = 9.4375;

%Calculate the multiplier section area for the different techniques
MultiplierArea = size(acc, 2)*MAcc + size(approx1, 2)*M1 + size(approx2, 2)*M2 + size(approx3, 2)*M3 + size(approx4, 2)*M4 + size(approx5,2)*M5 + size(approx6,2)*M6;
MirMultiplierArea = size(acc, 2)*MAcc + size(approx1, 2)*M6 + size(approx2, 2)*M3 + size(approx3, 2)*M2 + size(approx4, 2)*M5 + size(approx5,2)*M4 + size(approx6,2)*M1;
%MirMultiplierArea assumes every approximate multiplier gets mirrored with
%another approximate multiplier.

OverheadArea = 0;
switch(dim)
%Checks how big the multiplier is (eg. 2x2, 4x4, 8x8) and gives the
%corresponding overhead area for that MAC.
    case 2
        OverheadArea = 16;
    case 4
        OverheadArea = 32;
    case 8
        OverheadArea = 49;
    otherwise
        OverheadArea = 0;
        error('size of multiplier is irregular/out of bounds - OverheadArea is set to 0')
end
%Calculate the total area of the MAC
switch(type)
    case 1 %If the same set of multipliers is used for both of the
```

```
    parallel multipliers
46          AreaSize = MultiplierArea*2 + OverheadArea;
47      case 2 %If the mirror self healing type is used, so the 2nd
        multiplier uses a set of mirrored multipliers
48          AreaSize = OverheadArea + MultiplierArea +
        MirMultiplierArea;
49  end
50
51
52  end
```

D.4 CalculatePower

```
1   function PowerUsage = CalculatePower(ChosenMult, type)
2
3   %Functions mostly the same as the CalculateArea function, but with
        the parameters for power
4   acc = find(ChosenMult == 7);
5   approx1 = find(ChosenMult == 1); %Multiplier that has 3 error cases
         of 1 magnitude
6   approx2 = find(ChosenMult == 2); %Multiplier that has 1 error case
        of 2 magnitude (3*3 = 7)
7   approx3 = find(ChosenMult == 3); %Mirror pair of approx2 so 3*3 =
        11
8   approx4 = find(ChosenMult == 4); %Multiplier that has 1 error case
        of 4 magnitude (3*3 = 5)
9   approx5 = find(ChosenMult == 5);
10  approx6 = find(ChosenMult == 6);
11
12  %%Power consumption of different multiplier types
13  M1 = 1.33/32;
14  M2 = 1.27/32;
15  M3 = 1.52/32;
16  M4 = 1.46/32;
17  M5 = 1.52/32;
18  M6 = 1.38/32;
19  MAcc = 1.52/32;
20
21
22  MultiplierPower = size(acc, 2)*MAcc + size(approx1, 2)*M1 + size(
        approx2, 2)*M2 + size(approx3, 2)*M3 + size(approx4, 2)*M4 +
        size(approx5,2)*M5 + size(approx6,2)*M6;
23  MirMultiplierPower = size(acc, 2)*MAcc + size(approx1, 2)*M6 + size
        (approx2, 2)*M3 + size(approx3, 2)*M2 + size(approx4, 2)*M5 +
        size(approx5,2)*M4 + size(approx6,2)*M1;
24  %MirMultiplierPower assumes every approximate multiplier gets
        mirrored with
25  %another approximate multiplier.
26
27  switch(type)
28      case 1 %If the same set of multipliers is used for both of the
        parallel multipliers
29          PowerUsage = MultiplierPower*2;
30      case 2 %If the mirror self healing type is used, so the 2nd
        multiplier uses a set of mirrored multipliers
31          PowerUsage = MultiplierPower + MirMultiplierPower;
32  end
```

```

end
```

D.5 EvaluateMult

```
function [MEMult,MSEMult,MPEMult] = EvaluateMult(Distr, mean, var,
    VecSize, Inputamount, ChosenMult, type)


ModelSimInput = 0; % boolean that indicates whether input for
    ModelSim
                   % is created.

%Allocating space for the arrays
SequentialMACResult = zeros(1,Inputamount);
MirrorMACResult = zeros(1,Inputamount);
AccurateResult = zeros(1, Inputamount);

%Calculating the dimension of the NxN matrix.
dim = 2*sqrt(size(ChosenMult,2));

%Creating input vectors
for i = 1:Inputamount


Input1 = CreateInputs(Distr, mean, var, VecSize, dim);
Input2 = CreateInputs(Distr, mean, var, VecSize, dim);
Input3 = CreateInputs(Distr, mean, var, VecSize, dim);
Input4 = CreateInputs(Distr, mean, var, VecSize, dim);

%---------Creating testvector file for ModelSim----------
if ModelSimInput == 1

 FileFormat = 'Tvc_Input%dDistr%dMean%dVar%dVecSize.vhd';
   Filename = sprintf(FileFormat,Distr,mean,var,VecSize);
   fileID = fopen(Filename, 'w');
   fprintf(fileID,'library ieee;\n');
   fprintf(fileID,'use ieee.std_logic_1164.all;\n');
   fprintf(fileID,'use IEEE.Numeric_STD.all;\n');
   fprintf(fileID,'entity tvc_8bitPar is\n');
fprintf(fileID,'port (clk : out std_logic;\n');
fprintf(fileID,'i1 : out std_logic_vector(7 downto 0);\n');
fprintf(fileID,'i2 : out std_logic_vector(7 downto 0);\n');
fprintf(fileID,'i3 : out std_logic_vector(7 downto 0);\n');
fprintf(fileID,'i4 : out std_logic_vector(7 downto 0));\n');
fprintf(fileID,'end tvc_8bitPar;\n');
fprintf(fileID,'architecture behaviour OF tvc_8bitPar is\n');
fprintf(fileID,'constant half_clock_period: time := 10 ns;\n');
fprintf(fileID,'begin\n');
fprintf(fileID,'clock:process\n');
fprintf(fileID,'begin\n');
fprintf(fileID,'clk<= ''1'';\n');
fprintf(fileID,'wait for half_clock_period;\n');
fprintf(fileID,'clk <= ''0'';\n');
fprintf(fileID,'wait for half_clock_period;\n');
fprintf(fileID,'end process clock;\n');
```

```
fprintf(fileID,'inputs:process\n');
fprintf(fileID,'begin\n');
fprintf(fileID,'wait for 0.5*half_clock_period;\n');

for l = 1:size(Input1,2)

   fprintf(fileID, 'i1 <= std_logic_vector(to_unsigned(%d,i1''
    length));\n', Input1(l));
   fprintf(fileID, 'i2 <= std_logic_vector(to_unsigned(%d,i2''
    length));\n', Input2(l));
   fprintf(fileID, 'i3 <= std_logic_vector(to_unsigned(%d,i3''
    length));\n', Input3(l));
   fprintf(fileID, 'i4 <= std_logic_vector(to_unsigned(%d,i4''
    length));\n', Input4(l));
   fprintf(fileID, 'wait for 2*half_clock_period;\n');

end

fprintf(fileID,'end process inputs;\n');
fprintf(fileID,'end behaviour;\n');
fclose(fileID);

end
%-------------------------------------------------------------------------




%Calculate MAC output based on what setup is chosen
switch(type)
    case 1 %Conventional and Internal Self Healing setup
    SequentialMACResult(i) = SequentialMAC(Input1, Input2, Input3,
    Input4, dim, ChosenMult);

    case 2 %Mirror setup
    MirrorMACResult(i) = ParallelMACMir(Input1, Input2, Input3,
    Input4, dim, ChosenMult);

end
%Calculate the accurate result
AccurateResult(i) = sum(Input1.*Input2 + Input3.*Input4);

end


%Calculating the error metrics ME, MPE and MSE
switch(type)
    case 1

MPEMult = 100*sum(abs(AccurateResult-SequentialMACResult)./
    AccurateResult)/Inputamount;
MSEMult = sum((AccurateResult-SequentialMACResult).^2)/Inputamount;
MEMult  = sum(abs(AccurateResult-SequentialMACResult))/Inputamount;


    case 2
MPEMult = 100*sum(abs(AccurateResult-MirrorMACResult)./
```

```
    AccurateResult)/Inputamount;
MSEMult = sum((AccurateResult-MirrorMACResult).^2)/Inputamount;
MEMult  = sum(abs(AccurateResult-MirrorMACResult))/Inputamount;

end
```

D.6 CreateInputs

```
function Input = CreateInputs(distr, mean, dev, VecSize, dim)

%Function creates input vectors based on the given distribution,
    mean and standard deviation
switch(distr)
    case 1
        Input = randi(2^dim-1,1,VecSize);

    case 2
        Input = round(dev.*randn(1,VecSize) + repmat(mean,1,VecSize
    ));
    %If inputs are out of bounds they will be set to have the value
    %of that bound.
        Input(Input > 2^dim-1) = 2^dim-1;
        Input(Input < 0) = 0;

    otherwise
        Input = randi(2^dim-1,1,VecSize);

end
end
```

D.7 MirrorMAC

```
%Parallel MAC calculator
%Calculates the output of a parallel MAC system with a set of
    defined
%multipliers. By specifying the input both uniform and a normal
%distribution can be chosen, with a specified mean and standard
    deviation.
%An input vector specifies the elementary 2x2 multipliers that are
    used,
%and this multiplier vector automatically gets mirrored and used.


function  ParallelMACresult  = ParallelMACMir(i1, i2, i3, i4, dim,
    ChosenMult)

%
    ------------------------------------------------------------------

MirMult = ChosenMult;                   %Creates set of mirrored
    approximate
for k = find(ChosenMult == 2)           %multipliers. Every 3*3 = 7
    case
    MirMult(k) = 3;                     %gets converted to 3*3 =11,
    every
clear k;                  %3*3=5 case to 3*3=13 etc.
for k = find(ChosenMult == 3)
```

```
    MirMult(k) = 2;
end
clear k;
for k = find(ChosenMult == 4)
    MirMult(k) = 5;
end
clear k;
for k = find(ChosenMult == 5)
    MirMult(k) = 4;
end
clear k;
for k = find(ChosenMult == 1)
    MirMult(k) = 6;
end
clear k;
for k = find(ChosenMult == 6)
    MirMult(k) = 1;
end
clear k;
%-------------------------------------------------------------------------


mult = ['genericMultiply',int2str(dim),'x',int2str(dim)];
%Calculates results of multipliers for every input element using the
%lpACLib library
for i=1:size(i1,2);
multiplier1(i) = feval(mult,i1(i),i2(i),ChosenMult,'ACCURATE_ADD',0);
multiplier2(i) = feval(mult,i3(i),i4(i),MirMult,'ACCURATE_ADD',0);
end
mult1and2 = multiplier1 + multiplier2;
ParallelMACresult = sum(mult1and2);      %Accumulates all the results

end
```

D.8 ConventionalMAC

```

%Sequential MAC calculator
%Works in a similar fashion to the parallel version, only it simulates
%a sequential MAC, meaning that there is only 1 set of multipliers that
%do not get mirrored.


function  SequentialMACresult  = SequentialMAC(i1, i2, i3, i4, dim, ChosenMult)

%Calculates results of multipliers for every input element using the
%lpACLib library.
mult = ['genericMultiply',int2str(dim),'x',int2str(dim)];
for i=1:size(i1,2); %calculates results of multipliers for every
```

```
    input element
multiplier1(i) = feval(mult,i1(i),i2(i),ChosenMult,'ACCURATE_ADD'
    ,0);
multiplier2(i) = feval(mult,i3(i),i4(i),ChosenMult,'ACCURATE_ADD'
    ,0);
end
mult1and2 = multiplier1 + multiplier2;
SequentialMACresult = sum(mult1and2);      %Accumulates all the
    results

end
```

E Quartus Power Results

Type	Multiplier Section	Accumulator Section	Total Dynamic Power	Average Power per Multiplier
M1	1.35	1.17	2.52	0.04219
M2	1.30	1.11	2.41	0.04063
M3	1.54	1.26	2.8	0.04813
M4	1.47	1.25	2.72	0.04594
M5	1.52	1.22	2.74	0.04750
M6	1.35	1.17	2.52	0.04219
Macc	1.55	1.21	2.76	0.04844

Vector Size: 496, Uniform distribution, unit: mW

Type	Multiplier Section	Accumulator Section	Total Dynamic Power	Average Power per Multiplier
M1	1.33	1.15	2.48	0.04156
M2	1.26	1.09	2.35	0.03938
M3	1.52	1.10	2.62	0.04750
M4	1.46	1.19	2.65	0.04563
M5	1.52	1.24	2.76	0.04750
M6	1.38	1.18	2.56	0.04313
Macc	1.51	1.23	2.74	0.04719

Vector Size: 4960, Uniform distribution, unit: mW

Type	Multiplier Section	Accumulator Section	Total Dynamic Power	Average Power per Multiplier
M1	1.33	1.14	2.47	0.04156
M2	1.30	1.11	2.41	0.04063
M3	1.54	1.25	2.79	0.04813
M4	1.48	1.25	2.73	0.04625
M5	1.52	1.28	2.8	0.04750
M6	1.35	1.19	2.54	0.04219
Macc	1.54	1.21	2.75	0.04813

Vector Size: 496, Normal distribution, unit: mW

Type	Multiplier Section	Accumulator Section	Total Dynamic Power	Average Power per Multiplier
M1	1.33	1.15	2.48	0.04156
M2	1.28	1.12	2.4	0.04000
M3	1.53	1.20	2.73	0.04781
M4	1.47	1.22	2.69	0.04594
M5	1.53	1.39	2.92	0.04781
M6	1.38	1.18	2.56	0.04313
Macc	1.54	1.26	2.80	0.04813

Vector Size: 4960, Normal distribution, unit: mW

F Pareto Optimal Multiplier Configurations

Design / Location	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D10**	**D11**	**D12**	**D13**	**D14**	**D15**	**D16**	**D17**
$A_{HH}B_{HH}$	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{HH}$	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{HL}$	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{HL}$	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{LH}$	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7
$A_{LH}B_{HH}$	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LH}B_{HL}$	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LL}B_{HH}$	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LL}B_{HL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7
$A_{LH}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7
$A_{LH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7
$A_{LL}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7
$A_{LL}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7

Pareto optimal multiplier configurations for area and a uniform input distribution, conventional technique

Design / Location	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D10**	**D11**	**D12**	**D13**	**D14**	**D15**	**D16**	**D17**
$A_{HH}B_{HH}$	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{HH}$	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{HL}$	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{HL}$	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7
$A_{HL}B_{LH}$	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7
$A_{LH}B_{HH}$	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LH}B_{HL}$	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LL}B_{HH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7
$A_{LL}B_{HL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7
$A_{LH}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7
$A_{LL}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7
$A_{LL}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7

Pareto optimal multiplier configurations for area and a normal input distribution, conventional technique

Design / Location	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D10**	**D11**	**D12**	**D13**	**D14**	**D15**	**D16**	**D17**
$A_{HH}B_{HH}$	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{HH}$	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{HL}$	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{HL}$	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{LH}$	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7
$A_{LH}B_{HH}$	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LH}B_{HL}$	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LL}B_{HH}$	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LL}B_{HL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7
$A_{LH}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7
$A_{LH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7
$A_{LL}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7
$A_{LL}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7

Pareto optimal multiplier configurations for power and a uniform input distribution, conventional technique

Design / Location	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D10**	**D11**	**D12**	**D13**	**D14**	**D15**	**D16**	**D17**
$A_{HH}B_{HH}$	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{HH}$	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{HL}$	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{HL}$	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7
$A_{HL}B_{LH}$	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7
$A_{LH}B_{HH}$	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LH}B_{HL}$	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LL}B_{HH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7
$A_{LL}B_{HL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7
$A_{LH}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7
$A_{LL}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7
$A_{LL}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7

Pareto optimal multiplier configurations for power and a normal input distribution, conventional technique

Design / Location	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**
$A_{HH}B_{HH}$	M2	M3	M7	M7	M7	M7	M7
$A_{HL}B_{HH}$	M2	M2	M3	M3	M3	M3	M3
$A_{HH}B_{HL}$	M2	M4	M2	M2	M2	M2	M2
$A_{HL}B_{HL}$	M2	M4	M2	M2	M7	M7	M7
$A_{HH}B_{LH}$	M2	M2	M3	M3	M3	M3	M3
$A_{HH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2
$A_{HL}B_{LH}$	M2	M2	M4	M4	M4	M4	M4
$A_{HL}B_{LL}$	M2	M2	M4	M4	M4	M4	M4
$A_{LH}B_{HH}$	M2	M2	M3	M3	M3	M3	M7
$A_{LH}B_{HL}$	M2	M4	M2	M2	M4	M4	M3
$A_{LL}B_{HH}$	M2	M4	M4	M4	M4	M1	M2
$A_{LL}B_{HL}$	M2	M4	M4	M4	M2	M4	M4
$A_{LH}B_{LH}$	M2	M2	M2	M3	M2	M2	M3
$A_{LH}B_{LL}$	M2	M2	M2	M2	M2	M4	M4
$A_{LL}B_{LH}$	M2	M2	M2	M4	M2	M4	M1
$A_{LL}B_{LL}$	M2	M2	M2	M4	M2	M2	M4

Pareto optimal multiplier configurations for area and a uniform input distribution, internal self-healing technique

Design / Location	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D10**	**D11**
$A_{HH}B_{HH}$	M2	M2	M2	M4	M4	M2	M2	M2	M7	M7	M7
$A_{HL}B_{HH}$	M2	M2	M3	M3	M3	M3	M3	M3	M3	M3	M3
$A_{HH}B_{HL}$	M2	M2	M2	M3	M3	M7	M7	M7	M2	M2	M2
$A_{HL}B_{HL}$	M2	M3	M3	M3	M3	M3	M3	M3	M7	M7	M7
$A_{HH}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M3
$A_{HH}B_{LL}$	M2	M2	M2	M2	M4	M2	M2	M2	M3	M3	M2
$A_{HL}B_{LH}$	M2	M2	M2	M2	M3	M3	M4	M7	M3	M3	M2
$A_{HL}B_{LL}$	M2	M2	M2	M2	M4	M4	M2	M2	M2	M2	M2
$A_{LH}B_{HH}$	M2	M2	M2	M2	M4	M4	M2	M2	M3	M3	M2
$A_{LH}B_{HL}$	M2	M2	M2	M2	M2	M2	M3	M2	M2	M2	M3
$A_{LL}B_{HH}$	M2	M2	M2	M2	M4	M2	M2	M2	M2	M2	M3
$A_{LL}B_{HL}$	M2	M2	M2	M2	M2	M2	M4	M4	M3	M3	M3
$A_{LH}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M3	M3	M7
$A_{LH}B_{LL}$	M2	M2	M2	M2	M2	M4	M2	M2	M1	M4	M3
$A_{LL}B_{LH}$	M2	M2	M2	M2	M4	M4	M1	M1	M4	M4	M2
$A_{LL}B_{LL}$	M2	M2	M2	M2	M4	M4	M2	M2	M4	M7	M7

Pareto optimal multiplier configurations for area and a normal input distribution, internal self-healing technique

Location \ Design	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D10**
$A_{HH}B_{HH}$	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{HH}$	M2	M2	M3	M3	M3	M3	M3	M3	M3	M3
$A_{HH}B_{HL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2
$A_{HL}B_{HL}$	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7
$A_{HH}B_{LH}$	M2	M2	M2	M3	M3	M3	M3	M3	M3	M3
$A_{HH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2
$A_{HL}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2
$A_{HL}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2
$A_{LH}B_{HH}$	M2	M2	M2	M2	M3	M3	M7	M7	M7	M7
$A_{LH}B_{HL}$	M2	M2	M2	M2	M2	M2	M7	M7	M2	M2
$A_{LL}B_{HH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2
$A_{LL}B_{HL}$	M2	M2	M2	M2	M2	M2	M2	M2	M3	M3
$A_{LH}B_{LH}$	M2	M2	M2	M2	M2	M3	M2	M1	M3	M3
$A_{LH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M1	M1
$A_{LL}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M4	M4
$A_{LL}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M1	M4

Pareto optimal multiplier configurations for power and a uniform input distribution, internal self-healing technique

Location \ Design	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D10**	**D11**	**D12**	**D13**	**D14**
$A_{HH}B_{HH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7
$A_{HL}B_{HH}$	M2	M2	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3
$A_{HH}B_{HL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2
$A_{HL}B_{HL}$	M2	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M7	M7
$A_{HH}B_{LH}$	M2	M2	M2	M2	M2	M3	M2	M3	M2	M3	M2	M7	M3	M7
$A_{HH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M4	M4	M4	M2	M2	M3
$A_{HL}B_{LH}$	M2	M2	M2	M2	M3	M2	M3	M2	M3	M7	M3	M3	M2	M7
$A_{HL}B_{LL}$	M2	M2	M2	M2	M2	M3	M2	M2	M1	M2	M2	M4	M3	M2
$A_{LH}B_{HH}$	M2	M2	M2	M2	M2	M2	M2	M3	M3	M2	M3	M7	M2	M7
$A_{LH}B_{HL}$	M2	M2	M2	M3	M3	M3	M3	M7	M7	M3	M7	M7	M3	M7
$A_{LL}B_{HH}$	M2	M2	M2	M2	M2	M2	M4	M2	M2	M2	M2	M4	M3	M2
$A_{LL}B_{HL}$	M2	M2	M2	M2	M2	M2	M3	M4	M1	M4	M4	M2	M2	M3
$A_{LH}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M1	M1	M7	M7
$A_{LH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M3	M3	M3	M3
$A_{LL}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M1	M1	M1	M1	M1	M2	M2
$A_{LL}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7

Pareto optimal multiplier configurations for power and a uniform input distribution, internal self-healing technique

G Increasing Approximation Multiplier Configurations

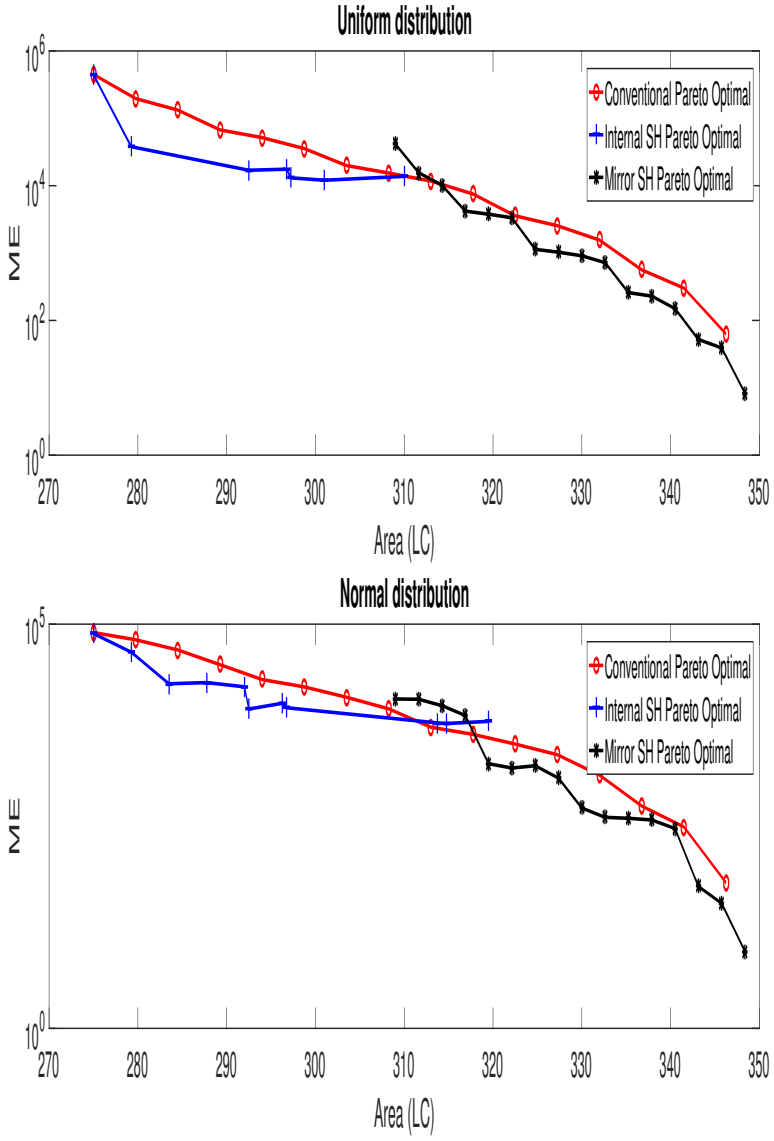
Design / Location	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**
$A_{HH}B_{HH}$	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{HH}$	M3	M7	M7	M7	M7	M7	M7
$A_{HH}B_{HL}$	M2	M7	M7	M7	M7	M7	M7
$A_{HL}B_{HL}$	M3	M3	M7	M7	M7	M7	M7
$A_{HH}B_{LH}$	M2	M2	M7	M7	M7	M7	M7
$A_{HH}B_{LL}$	M3	M3	M3	M7	M7	M7	M7
$A_{HL}B_{LH}$	M2	M2	M2	M7	M7	M7	M7
$A_{HL}B_{LL}$	M7	M7	M7	M7	M7	M7	M7
$A_{LH}B_{HH}$	M7	M7	M7	M7	M7	M7	M7
$A_{LH}B_{HL}$	M3	M3	M3	M3	M7	M7	M7
$A_{LL}B_{HH}$	M2	M2	M2	M2	M7	M7	M7
$A_{LL}B_{HL}$	M3	M3	M3	M3	M3	M7	M7
$A_{LH}B_{LH}$	M2	M2	M2	M2	M2	M7	M7
$A_{LH}B_{LL}$	M3	M3	M3	M3	M3	M3	M7
$A_{LL}B_{LH}$	M2	M2	M2	M2	M2	M2	M7
$A_{LL}B_{LL}$	M7	M7	M7	M7	M7	M7	M7

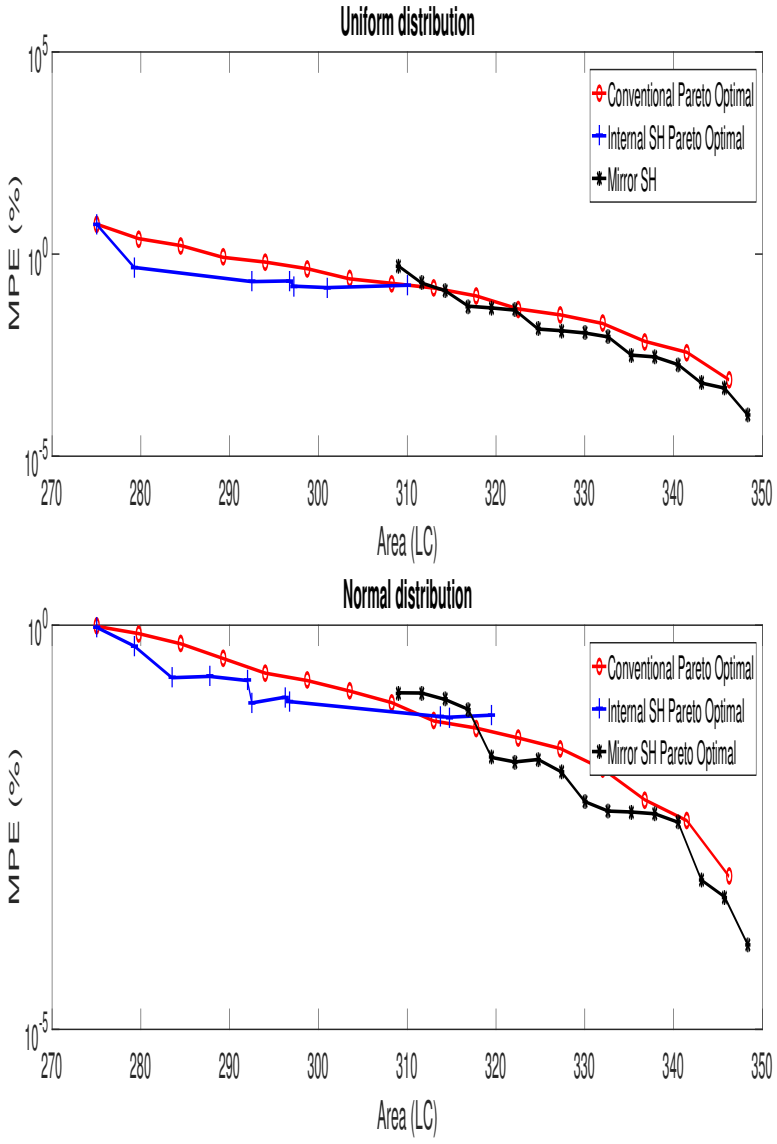
Increasing approximation multiplier configuration for internal self-healing technique

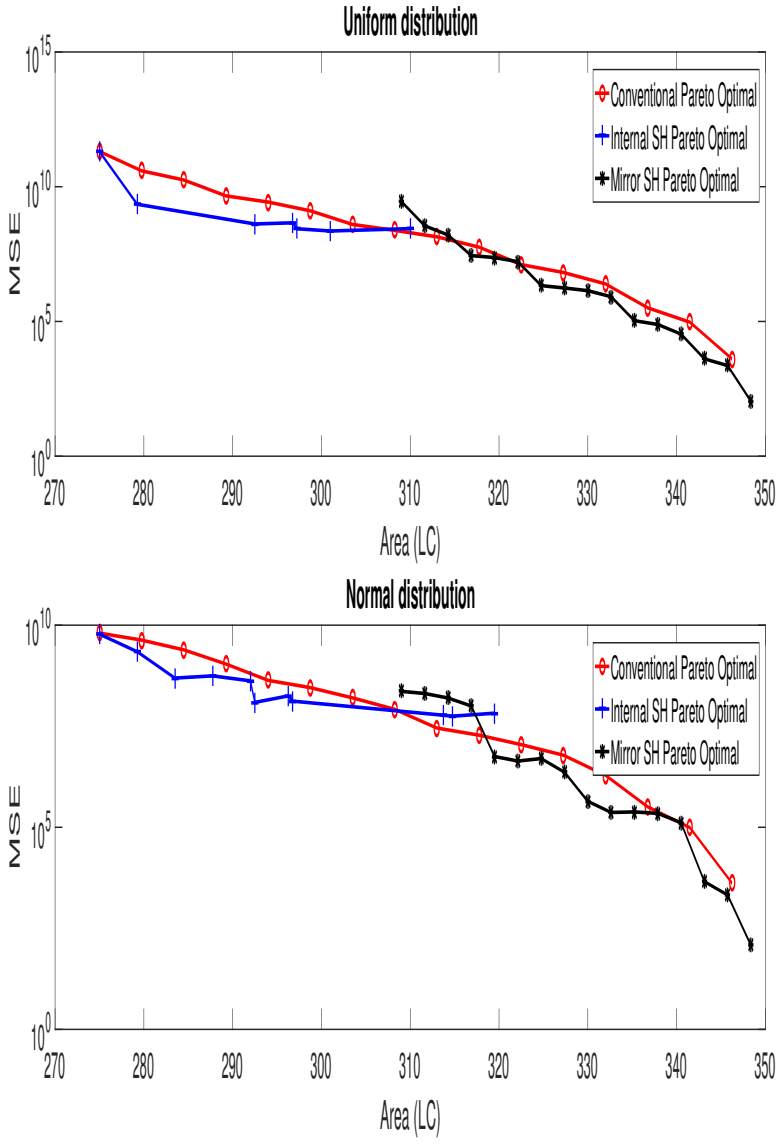
Design / Location	**D1**	**D2**	**D3**	**D4**	**D5**	**D6**	**D7**	**D8**	**D9**	**D10**	**D11**	**D12**	**D13**	**D14**	**D15**	**D16**	**D17**
$A_{HH}B_{HH}$	M3	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{HH}$	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{HL}$	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{HL}$	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{LH}$	M3	M3	M3	M3	M3	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{HL}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7
$A_{LH}B_{HH}$	M3	M3	M3	M3	M3	M3	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LH}B_{HL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7	M7
$A_{LL}B_{HH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7	M7	M7
$A_{LL}B_{HL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7	M7	M7
$A_{LH}B_{LH}$	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M3	M7	M7	M7	M7
$A_{LH}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7	M7
$A_{LL}B_{LH}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7	M7
$A_{LL}B_{LL}$	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M2	M7

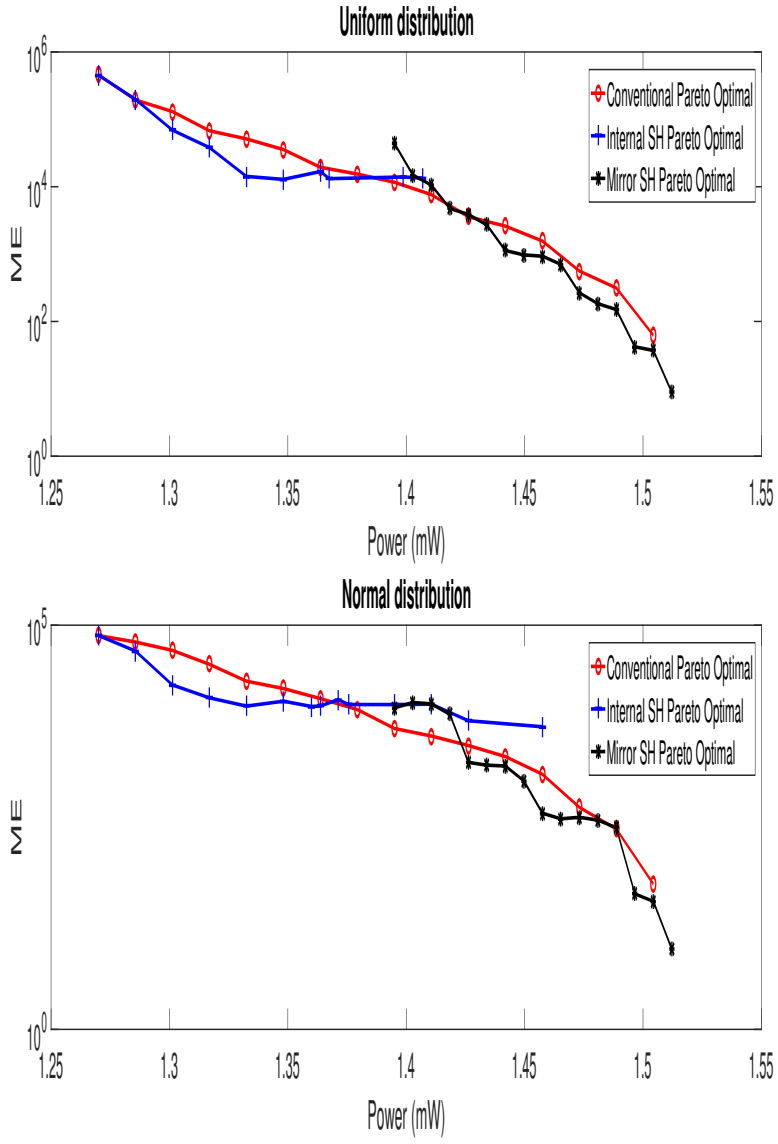
Increasing approximation multiplier configuration for mirror self-healing technique

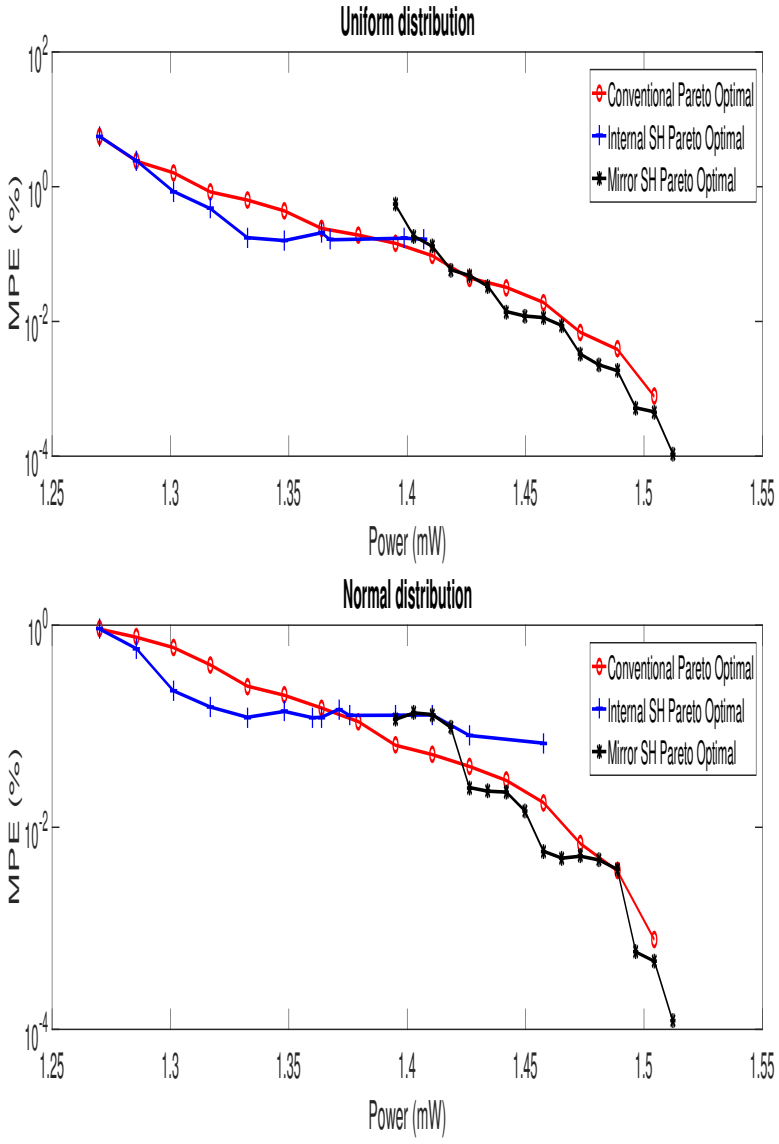
H Quality Evaluation Results

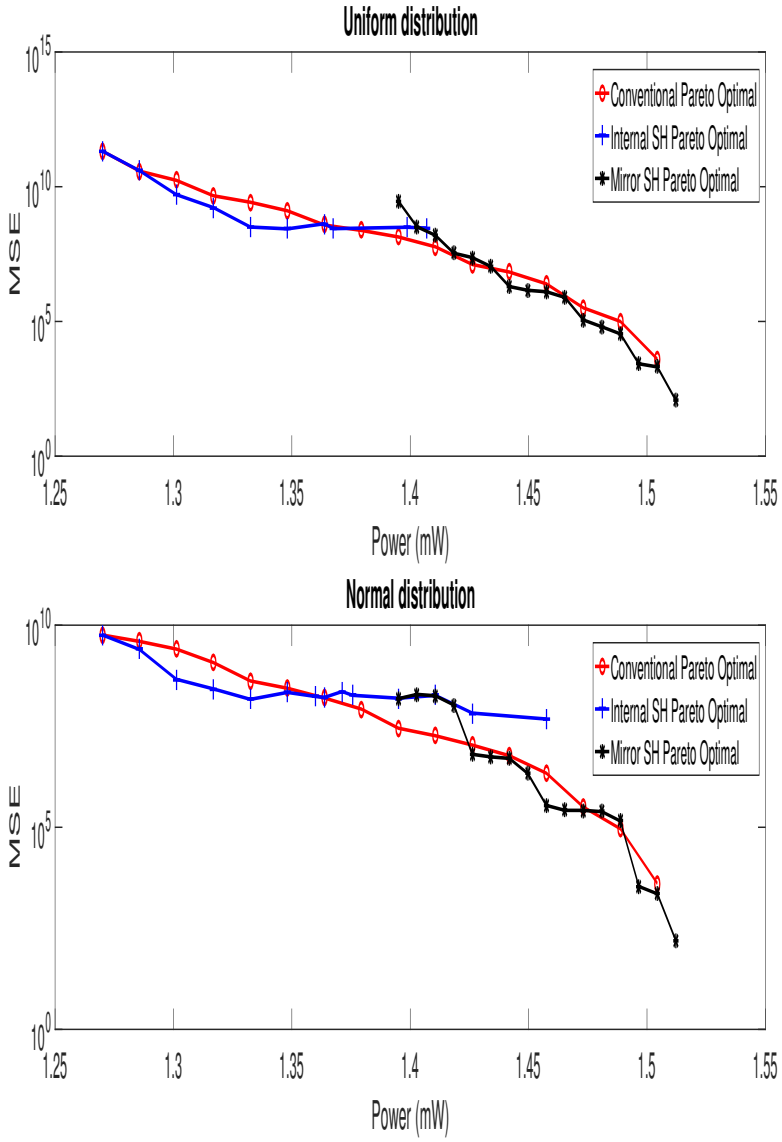



Uniform distribution
MPE (%)
Area (LC)
Conventional Pareto Optimal
Internal SH Pareto Optimal
Mirror SH
Normal distribution
MPE (%)
Area (LC)
Conventional Pareto Optimal
Internal SH Pareto Optimal
Mirror SH Pareto Optimal


Uniform distribution
MSE
Area (LC)
Conventional Pareto Optimal
Internal SH Pareto Optimal
Mirror SH Pareto Optimal
Normal distribution
MSE
Area (LC)
Conventional Pareto Optimal
Internal SH Pareto Optimal
Mirror SH Pareto Optimal


Uniform distribution
ME
Power (mW)
Conventional Pareto Optimal
Internal SH Pareto Optimal
Mirror SH Pareto Optimal
Normal distribution
ME
Power (mW)
Conventional Pareto Optimal
Internal SH Pareto Optimal
Mirror SH Pareto Optimal


Uniform distribution
Conventional Pareto Optimal
Internal SH Pareto Optimal
Mirror SH Pareto Optimal
MPE (%)
Power (mW)
Normal distribution
Conventional Pareto Optimal
Internal SH Pareto Optimal
Mirror SH Pareto Optimal
MPE (%)
Power (mW)


Uniform distribution
MSE
Power (mW)
Conventional Pareto Optimal
Internal SH Pareto Optimal
Mirror SH Pareto Optimal
Normal distribution
MSE
Power (mW)
Conventional Pareto Optimal
Internal SH Pareto Optimal
Mirror SH Pareto Optimal