AlphaRA: An AlphaZero based approach to Redundancy Analysis

Helik Kanti Thacker[1], Atishay Kumar[1], Adrita Barari[1], Damini[1], Ankit Gupta[1], Keerthi Kiran Jagannathachar[1], Deokgu Yoon[2]
DRAM Solutions[1], DRAM Product Engineering Team[2]
Samsung Semiconductor India Research and Development[1], Samsung Electronics[2]
{h.thacker, atishay.1, adrita.b, damini20.d, ankit.g2, keerthi.k, deokgu.yoon}@samsung.com

***Abstract*— Manufacturing flaws in memory devices give rise to faulty cells rendering the chips unusable and consequently reducing the wafer yield. To repair faulty memory cells, redundancies are included in the form of spare rows and columns in the memory. Redundancy Analysis is the process of mapping these spare rows and columns to repair faulty lines in the chip. However, Redundancy Analysis is an NP-complete problem, making it difficult to find a trade-off between repair rate and runtime, especially for large chip sizes. In this paper, we introduce AlphaRA, a first-of-its-kind memory repair algorithm based on the Reinforcement Learning algorithm AlphaZero. We explicate AlphaRA as a single agent problem that learns the strategies of Redundancy Analysis through self-play. Starting tabula rasa, AlphaRA achieves an average normalized repair rate of 99.8% on 16×16 chips with only 32 MCTS simulations. It outperforms the next best heuristic algorithm by 5.42% while utilizing 0.29% lesser spares, making it a suitable Redundancy Analysis algorithm for mass production of memory devices.**

Keywords—reinforcement learning, redundancy analysis, AlphaZero, memory repair.

I. Introduction

The semiconductor manufacturing industry has seen a rapid surge due to advancements in memory technology. In order to meet the increasing demand for memory devices, the leading producers are manufacturing them in huge amounts on a single wafer. Manufacturers have also increased memory densities and decreased the node sizes in these devices due to which the probability of faults in the memory has increased. During the fabrication process, there are several external factors such as temperature, equipment inaccuracies, undesired chemical and airborne particles that can increase the faults in the chip, leading to a reduction in the overall wafer yield.

With increasing fault probabilities in the memory devices, semiconductor manufacturers have incorporated redundancies into the memories in the form of spare rows and columns which can be used to repair the chips. The process of allocating these spare rows and columns to faulty lines in the chip is called Redundancy Analysis (RA) [1]. A chip is considered as repaired only if all the faulty lines are mapped to the spares. As illustrated in Fig. 1 the chip has 5 faulty cells, 1 spare row, and 2 spare columns. The chip is repaired by mapping the spare row to row 3 and the spare columns to columns 2 and 3. These mappings are used during memory repair to substitute the entire faulty rows and columns. During a read or write request to a faulty line, the mapped spare is accessed instead of accessing the faulty line. The decision of allocation of spares to faulty rows and columns has a high impact on the overall yield. Since RA is an NP-complete problem [1], an exponential-time algorithm is required to achieve the maximum possible yield.

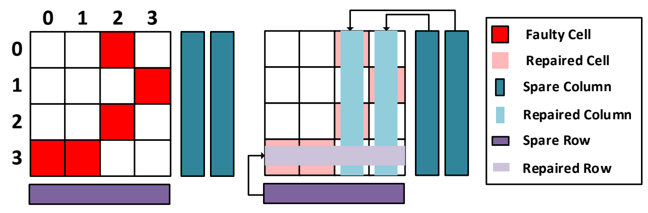

Fig. 1: RA repair process for a 4x4 memory chip with 5 faulty cells having 1 spare row and 2 spare columns

Previous research has focused on heuristic and exhaustive RA algorithms. Heuristic RA algorithms such as Broadside [2], Repair-Most [1], LECA [3] and OSP [4] have lower runtime but also have low repair rates. On the other hand, exhaustive RA algorithms, such as FLCA [3], Branch-and-Bound, and PAGEB [1] achieve optimal repair rates but with exponential time complexities. Section II further discusses the existing algorithms along with their merits and limitations.

Heuristic algorithms currently used for memory repair are designed with certain insights in mind such as patterns of faulty rows and columns. The motivation for exploring a Reinforcement Learning (RL) approach for RA is driven by the fact that RL allows the agent to self-discover unconventional strategies, without requiring insights into fault patterns. The agent learns to repair the chip without any design insights or handcrafted rules.

In this paper, we introduce an AlphaZero [5] based RA algorithm, AlphaRA. It learns from its environment by combining Monte Carlo Tree Search (MCTS) and Deep Neural Networks (DNN) iteratively for policy evaluation and improvement. Through self-play over many episodes, the agent learns a policy to achieve a high repair rate.

The contributions of our paper are as follows:

- We explicate AlphaZero which was originally developed for two-player zero-sum games, as a single-player RA algorithm
- AlphaRA outperforms heuristic algorithms in terms of yield and spare utilization while maintaining a comparable yield with the exhaustive algorithms
- We also demonstrate the scalability of AlphaRA on different chip sizes

Thus, we show that AlphaRA is an effective memory repair algorithm for large scale production of memory devices.

The rest of the paper has been organized as follows. Section II gives a background of the existing memory repair algorithms and the key concepts related to AlphaZero and Memory Fault Simulators. Section III describes the proposed solution AlphaRA. Experimental setup in Section IV is followed by the results and analysis in Section V. Finally, Section VI explains the future scope of our work followed by the conclusion.

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

II. Background

A. Redundancy Analysis Algorithms

Effective RA algorithms should provide a good mapping of faulty rows and columns to the spares in the memory device within a reasonable time. Such an algorithm should have a high repair rate, which can be measured by the number of chips repaired by the algorithm. The repair rate and normalized repair rate [1] are defined as:

$$RR = C_{Repaired}/C_{Total} \qquad (1)$$

$$NRR = C_{Repaired}/C_{Repairable} \qquad (2)$$

where *RR* and *NRR* are the repair rate and the normalized repair rate respectively. $C_{Repaired}$ is the number of chips repaired. C_{Total} and $C_{Repairable}$ are the total number of chips and the number of theoretically repairable chips respectively. C_{Total} includes theoretically unrepairable chips as well which may downplay the efficiency of an RA algorithm. However, *NRR* is calculated independent of these unrepairable chips, so it is better suited for estimating the yield of an RA algorithm. If an algorithm is not able to repair the chip, it is deemed unusable and discarded, thereby making it crucial for an RA algorithm to have a high repair rate. To make the repair viable, an RA algorithm should be selected which has a high repair rate along with a feasible runtime.

Exhaustive algorithms like Branch-and-Bound [1], PAGEB [1], Faulty Line Covering Algorithm (FLCA) [3], and Fault Driven Comprehensive algorithm [2] construct a search tree and are able to find a repair solution whenever one exists, i.e. these algorithms have a NRR of 100%. While the Fault Driven Comprehensive algorithm [2] branches for each faulty cell, FLCA [3] only branches for each faulty line. FLCA is based on the principle that a faulty row with *k* faults can be covered either by a spare row or *k* spare columns. A similar repair approach is followed for a faulty column. The single faults are also repaired separately in the algorithm, reducing the number of branches significantly. However, with a large number of faults, the space and time complexity of the algorithm increases exponentially which makes it infeasible in the manufacturing line.

Heuristic algorithms like Broadside [2], Repair-Most [1], LECA [3] and OSP [4] are capable of finding the repair solution quickly, but they are not able to achieve an optimal repair rate. Broadside Algorithm [2] is a greedy heuristic algorithm that assigns a spare row or column, whichever is in excess when it repairs a fault. In the case of same number of spare rows and columns, the assignment is based on the algorithm design. Compared to the other algorithms, Broadside has a low runtime but also a low repair rate.

The Largest Effective Coefficient Algorithm (LECA) [3] uses Effective Coefficients (EC) to rank the rows and columns of a chip in the order of repair. The EC considers both fault counters and complements of a faulty line. LECA is not very effective with random faulty bits distribution as effective coefficients have less significance. Its performance varies a lot with the variation in the number of single faults.

One Side Pivot algorithm (OSP) [4] uses pivot fault properties to find repair priorities reducing the analysis time even when the fault rate is high. Faults are classified into pivot faults, intersection faults, and OSP faults. Pivot fault is a fault that is not included in any other faulty line. An intersection fault is included in both a faulty column and row. One side pivot fault is a pivot fault, which is not included in a faulty line that does not have an intersection fault. If a fault is a pivot in its row, it is solved using a spare column and vice versa. Thus, the time taken to find a solution by this algorithm is low but it does not achieve an optimal repair rate. We have tabulated the time complexities of RA algorithms used for comparison with the AlphaRA algorithm in Table 1.

Table 1: Time Complexity of the algorithms

Algorithm	Time complexity	Remarks
Broadside	$O(n)$	*n* is number of faults
FLCA	$O\left(2^{\left(\frac{n-S_F}{m}+1\right)}-1\right)$	*n* is number of total faults, S_F is number of single faults, *m* is the least faulty cell / line
LECA	$O(\max\{R_A, C_A\}^2 \cdot log max\{R_A, C_A\})$	R_A is redundant rows C_A is redundant columns
OSP	$O(max(n, n_p.n))$	*n* is number of faults n_p is number of pivot fault

B. Must-Repair

A row having more faulty cells than the available spare columns is said to be in must-repair condition [1]. Similarly, a column with more faulty cells than the available spare rows is also in must-repair condition. A row (column) in must-repair condition should always be repaired by a spare row (column). It is defined as

$$\sum\nolimits_{ri\in\{R\}} n_{ri} > SR, \qquad or \sum\nolimits_{ci\in\{C\}} n_{ci} > SC \qquad (3)$$

where rows r_i and columns c_i belong to the set of all faulty rows $\{R\}$ and columns $\{C\}$ respectively. $\sum n_{ri}$ and $\sum n_{ci}$ are the total number of faults in row r_i and column c_i respectively. *SR* and *SC* are the available spare rows and columns. As illustrated in Fig. 2, column 0 has 2 faults which is more than the 1 available spare row. Thus column 0 is in must-repair condition and needs to be repaired by a spare column.

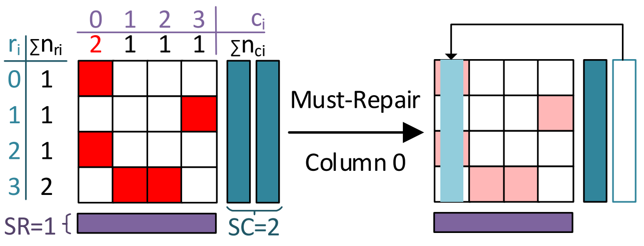

Fig. 2: An example of a 4×4 chip with column 0 in must-repair condition

C. Monte Carlo Tree Search

Tree Search Algorithms are commonly used in games where each tree node represents a game state. They search every possible move that may exist from a particular state to reach a solution. The brute force solution of considering every child node in the tree requires a lot of computational power. Thus, the selection of some moves over others, according to some policy, helps in speeding up the Tree Search.

Monte Carlo Tree Search (MCTS) [6] is a tree search algorithm that selects some nodes over others based on node statistics to reach a solution faster. It performs a four-step process of selection, expansion, rollout, and backpropagation to determine the node statistics.

In each MCTS simulation, the game is played to the end by selecting moves according to the Upper Confidence Bound for Trees (UCT) [7] formula shown in (4). When a leaf node is encountered, it is expanded and a random rollout is performed from that newly expanded node. The return value is then backpropagated in the tree to update the statistics. We maintain the following search statistics in each tree edge:

$Q(s,a)$ - expected reward for taking action a from state s
$N(s,a)$ - number of times action a was taken from state s

$$UCT(s,a) = Q(s,a) + C * \sqrt{\frac{ln(N(s))}{N(s,a)}} \qquad (4)$$

where C is an exploration parameter and $N(s)$ is the total number of visits to the parent state s.

Predictor + UCT (PUCT), shown in (5), further improves UCT to prioritize good moves by combining it with a predictor which guides the MCTS search. It removes the need to visit all the child nodes at least once. Thus, only promising moves are explored which reduces the number of MCTS simulations required to arrive at the solution. Additionally, we also maintain $P(s,a)$, which is the probability given by the predictor of taking action a from state s.

$$PUCT(s,a) = Q(s,a) + Cpuct * P(s,a) * \frac{\sqrt{N(s)}}{1 + N(s,a)} \qquad (5)$$

where $Cpuct$ is an exploration parameter. These four steps are iterated for a fixed number of simulations and the action with the best statistic is selected, with ties broken randomly.

D. *AlphaGo and Alpha Zero*

Researchers have developed an algorithm that efficiently combines DNN and MCTS with self-play to predict the best moves for the complex game of Go. AlphaGo [8] is the first paper in the series, which shows that DNNs could play Go by predicting a policy and value estimate. A policy $\boldsymbol{\pi}$ is a mapping from actions to the probabilities of selecting those actions. The value function of a state is the estimate of how good it is for an agent following $\boldsymbol{\pi}$ to be in that state. These estimates are then used to assist a tree-based look ahead search by selecting which actions to take from given states. AlphaGo also incorporates supervised learning on a dataset of expert moves of professional Go players to train the neural network.

AlphaGo Zero [9], improves upon AlphaGo by starting from zero human knowledge. It combines the value and policy network into a single neural network and replaces the rollouts in MCTS with the value returned from the DNN. AlphaZero [5] demonstrates the effectiveness and generality of the AlphaGo Zero further by making a few subtle modifications to it and generalizing it for games like Chess and Shogi. The extension of AlphaZero for single-player games has been discussed in [10].

E. *Memory Fault Simulators*

For the development of AlphaRA, we required a variation of fault patterns and a large number of chips which is why we used a Memory Fault Simulator. We also needed chips of different sizes for checking the scalability of the algorithm. Various simulators such as VLASIC [11], RAISIN [12], and SEARS [13] have been proposed in the literature that aim to simulate the fault patterns found in real memory chips. SEARS is one of the latest simulators that incorporates numerous fault patterns and multiple memory repair algorithms. It also supports generating chips of varying sizes. Thus, in this paper, we use SEARS to generate faulty chips.

III. Proposed Solution - AlphaRA

We describe the proposed RA algorithm, AlphaRA, in the following subsections.

A. *Redundancy Analysis as a Markov Decision Process*

We formulate RA as a Markov Decision Process (MDP). The state $\boldsymbol{s}$ consists of the memory chip and the number of spare rows and columns available. The memory chip is represented by an $N \times N$ 2D Boolean matrix with 1 indicating a faulty cell. The action space is a vector of length $2N$ where the first N elements correspond to selecting rows 0 to $N-1$ and the next N elements correspond to selecting columns 0 to $N-1$ for repair.

For a given state s, we first check if there are any faulty lines in must-repair condition. If so, the action set for s consists only of the faulty lines in must-repair condition. Otherwise, the action set consists of all the faulty lines in the chip. When a faulty row or column is repaired, all the 1s in that line are zeroed out in s.

An 8×8 chip with 2 spare rows and 2 spare columns is shown along with the initial state representation in Fig. 3 (a). The action space is of length 16 with actions 0 to 7 indicative of rows 0 to 7 being selected for repair, whereas actions 8 to 15 represent columns 0 to 7 being selected. Since row 1 and column 4 are in must-repair condition, the only valid actions from s_0 are 1 and 12. The agent selects action 1 in Fig. 3 (a) and hence row 1 is repaired by zeroing it out as depicted in Fig. 3 (b). The agent chooses actions until the episode terminates which occurs if either of the following conditions is satisfied:

i. all faults in the chip are successfully repaired
ii. all spares are exhausted leaving the chip unrepaired

The complete episode for the sample 8×8 chip continues from Fig. 3 (c) through Fig. 3 (e). Here, the agent repairs the chip in 4 time steps, using all 4 spares.

The reward is given to AlphaRA at the end of an episode, thus the reward structure is sparse. The reward is +1 if the agent successfully repairs the chip and -1 if it fails to do so.

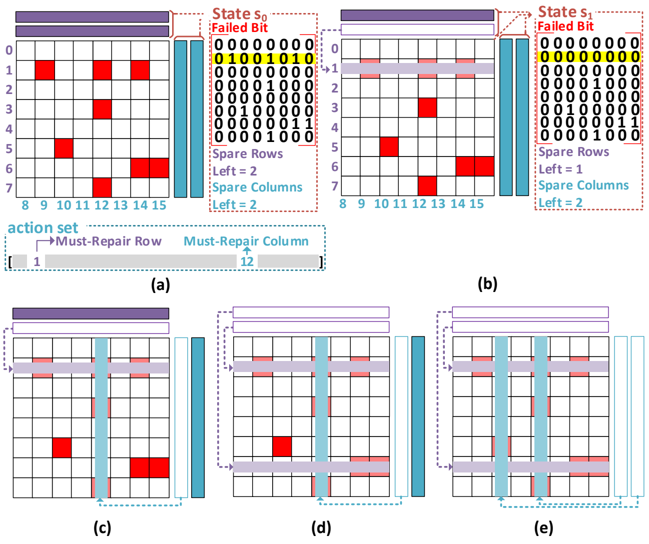

Fig. 3: State Space and Action Set on an 8×8 chip with 2 spare rows and columns each

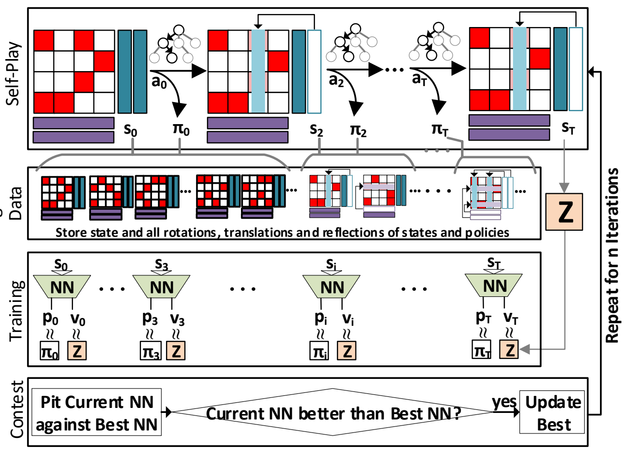

Fig. 4: Training pipeline of AlphaRA

B. *Training through Self-play*

Training of AlphaRA starts with self-play as shown in Fig. 4. A self-play episode consists of the repair process of a single chip. In each iteration, a number of self-play episodes are carried out. For each time step t of repair, a number of MCTS simulations are performed starting from the current state s_t and an improved policy $\boldsymbol{\pi}_t$ is returned. The estimates obtained from MCTS are then used as new targets to subsequently train the DNN and get improved policy and value functions.

The DNN f_θ, parameterized by θ, is trained from the dataset collected from self-play. It takes the state of the chip (s) as input and has two outputs: a value of state $v(s) \in [-1, 1]$ and a vector of action probabilities $\boldsymbol{p}_\theta(s)$. The DNN is initialized to random weights, thus the initial policy $\boldsymbol{p}_\theta$ is also random. For each state s, an MCTS search is executed, guided by the DNN f_θ. The DNN acts as the predictor. The actions in MCTS are selected according to the PUCT formula (5) until a leaf node is encountered. Once a leaf node s' is encountered, it evaluates the node by using the DNN to predict a policy and a value $(\boldsymbol{p}_\theta(s'), v(s'))$ for this node. Instead of performing a rollout from s', it backpropagates $v(s')$ and updates the Q and N values of all nodes along the current simulation path. If instead, we come across a terminal state during the search, we propagate the actual reward z, i.e. +1 if the chip is repaired and -1 otherwise. The MCTS outputs a vector of action probabilities $\boldsymbol{\pi}$, which is generally much stronger than the DNN policy $\boldsymbol{p}_\theta$.

The data generated from each time step t of a self-play episode is stored as a tuple (s_t, π_t, z) where z is the reward at the end of the episode. This data is inserted in a replay buffer queue of a fixed length. Since successive states are strongly correlated, we do a random uniform sampling from the replay buffer in order to train the DNN. The objective is to minimize the Mean Squared Error between the predicted value v and the target z and to maximize the similarity between predicted neural network policy $\boldsymbol{p}_\theta$ and target MCTS policy $\boldsymbol{\pi}$ by minimizing the cross-entropy loss. The combined loss function [9], ℓ is defined as

$$\ell = (z - v)^2 - \boldsymbol{\pi}^{\mathrm{T}} \cdot \log(\boldsymbol{p}) \qquad (6)$$

Thus, the MCTS drives the DNN to output better policies, which in turn guides the MCTS to search the action space more efficiently, and iteratively increases the overall strength of the algorithm.

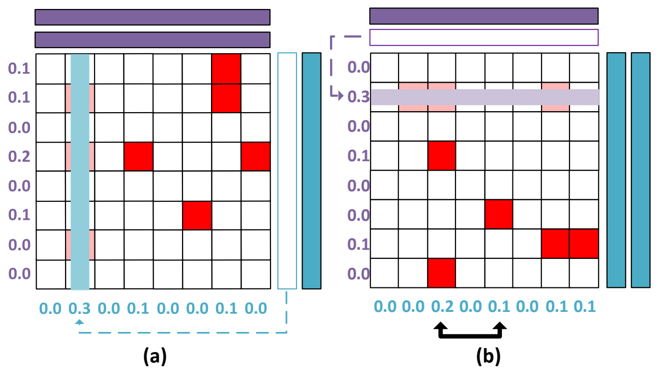

Fig. 5: An example of rotation and swapping

C. *State Rotation , Reflection and Swapping*

Since rotating, reflecting, or swapping operations do not affect the repairability of the chip, we augment the data so that the DNN has a richer dataset to learn from. Fig. 5 (a) shows the state in Fig. 3 (b) rotated by 90° anti-clockwise. It illustrates the state and the corresponding policy rotated. This also changes the availability of spares. Earlier there were 2 spare columns and 1 spare row available, but in the rotated version, there are 2 spare rows and 1 spare column. Fig. 5 (b) shows swapping of columns 2 and 4. For every state, we rotate it by 90°, 180°, and 270° along with their reflections to get 8 states. We also swap 4 pairs of row and column indices each to get 8 more states. Thus, for every state from self-play we obtain a total of 16 states. These states are stored in the replay buffer queue.

D. *Contest*

AlphaZero is designed for two-player adversarial board games, whereas RA can be viewed as a single-player game. In two-player adversarial games, the game result (win/draw/lose) is clear. For RA, we provide the same chips from the validation set to both the best and the current AlphaRA agent and the game result is determined as follows:

i. If both agents repair the chip:

   a. If the spare utilization by both the agents is the same, the game ends in a draw.

   b. Otherwise, the agent using fewer number spares is the winner

ii. If both agents fail to repair the chip:

   a. If both agents repair the same number of faults, the game ends in a draw.

   b. Otherwise, the agent repairing more faults is the winner.

iii. If one agent repairs the chip and the other agent fails to do so, the agent which repairs the chip is the winner.

We include spare utilization as a criterion along with repair rate, to determine the performance of an agent. This is because an algorithm that uses fewer spares is preferred as spares saved can be used further in the manufacturing process. If the current agent beats the best agent by a certain threshold, the best model is updated. The next iteration of AlphaRA self-play starts after the contest phase.

IV. Experimental Setup

Different chips sizes and fault rates were used for performing experiments. We used the SEARS simulator which had Broadside, LECA, OSP, and FLCA algorithms implemented. We elaborate the modifications to the simulator parameters in this section along with the experimental setup used for AlphaRA which includes the DNN architecture and the hyperparameters used.

A. Experimental Setup – SEARS

We generate datasets of chip sizes 8×8, 16×16, and 32×32 with the parameters shown in Table 2. The underlying fault distribution is similar to SEARS [13]. The training dataset is used for each self-play episode, whereas the validation dataset is used in contest phase of AlphaRA. All the results presented in Section V (B to E) are for 16×16 chips on the entire test set.

Table 2: Simulation Parameters for various chip sizes

#	Chip Size	Spares	Fault range	Training/Testing/Validation Data
1	8	2	[9%, 31%]	1×10^4
2	16	4	[4%, 14%]	5×10^4
3	32	6	[2%, 5%]	1×10^4

B. Experimental Setup – AlphaRA

The DNN used in AlphaRA has 4 convolution layers used with batch normalization and ReLU activation function. The features extracted from the DNN are appended with available spares data and is the input to the fully connected layers. There are 2 output heads to predict policy and value. Softmax is applied on the output of the policy head whereas $tanh$ is applied on the output of the value head. The loss function is a summation of the policy loss, which is measured by cross-entropy between $\boldsymbol{p}_\theta$ (DNN policy) and $\boldsymbol{\pi}$ (MCTS policy) and the value loss, which is the mean-squared error between v (DNN value) and z (MCTS value) as shown in (6).

The AlphaRA training hyperparameters are as follows. Each training iteration calls 100 episodes of self-play. This self-play data is stored in a replay buffer queue of length 2×10^5. The number of MCTS simulations used in training is 128 while the exploration factor $Cpuct$ is set to 1. Since RA is invariant to rotation, reflection, and swapping, we augment our data with 4 rotations and their reflections and 8 swaps (4 each for rows and columns). At every iteration, we pit our current agent against the best agent on 200 chips from the validation set. If the current model beats the best model by a threshold of 55%, the current model becomes the best model.

Our implementation is based on [14], which is a synchronous single-thread single-GPU implementation on the Pytorch framework. The results are benchmarked on a system with Intel® i9-9900K (5.00 GHz), Nvidia 2080Ti (11GB), and 32GB RAM on an Ubuntu 18.04.4 LTS OS.

V. Results and Analysis

A. Empirical Analysis of AlphaRA training

In order to determine which training iteration of AlphaRA gives the best result, we compare their performance on the validation dataset. Fig. 6 illustrates the normalized repair rate of AlphaRA as the training progresses. Every time the best neural network is updated, we run AlphaRA with just 2 MCTS simulations. With just 15 training iterations (~5 hours), it starts to outperform LECA and OSP. The best model is found in the 76th iteration (~51 hours) and has ~97% normalized repair rate. All subsequent results for 16×16 chips are obtained using this model.

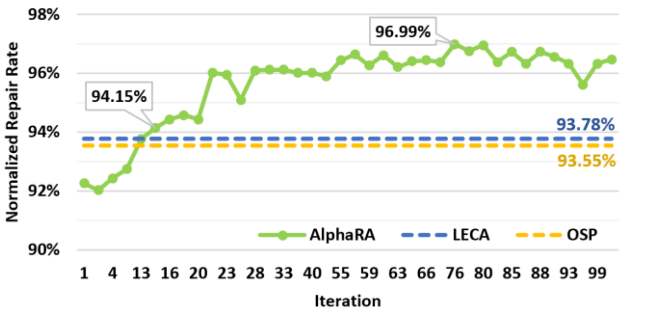

Fig. 6: Normalized repair rate of AlphaRA during training for 100 iterations

B. Yield Comparison

For assessing the strength of AlphaRA, we compare its normalized repair rate against other algorithms. Out of 50000 test dataset chips, 35159 chips were theoretically repairable. Broadside, LECA, and OSP were able to repair 9789, 33063, and 33185 chips respectively while AlphaRA with 32 MCTS simulations was able to repair 35090 chips, achieving an average normalized repair rate of 99.8%, which is 5.4% more than the next best algorithm LECA. In Fig. 7 we compare the normalized repair rate for varying chip faults. We observe that for the number of faults less than 11 (~ 4.29%), all algorithms have a normalized repair rate above 95% but with an increase in the number of faults, it starts decreasing rapidly for the existing algorithms. In comparison, AlphaRA is able to maintain a high normalized repair rate throughout.

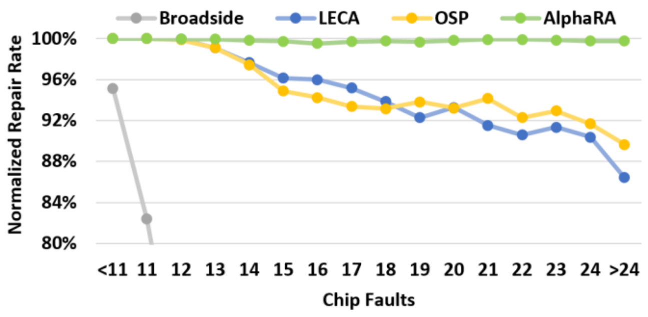

Fig. 7: Normalized repair rate comparison of AlphaRA against heuristic algorithms for varying chip faults. Normalized repair rate of Broadside drops below 65% for more than 11 faults.

C. Spare Utilization

In Table 3, we compare the spare row, spare column, and total spare utilization. With an average total spare utilization of 7.107, AlphaRA has the least spare utilization compared to all the algorithms under consideration. OSP gives a higher preference to spare columns whereas LECA gives a higher preference to spare rows. This leads to these algorithms exhausting either spare rows or columns quickly resulting in a lower normalized repair rate. However, AlphaRA gives approximately equal preference to spare rows and columns (~3.5), achieving a much higher normalized repair rate.

Table 3: Average spare utilization by different algorithms

Algorithm	AlphaRA	LECA	OSP	Broadside
Spare Rows	3.532	3.692	**3.337**	3.963
Spare Columns	3.575	**3.422**	3.791	3.884
Total Spares	**7.107**	7.114	7.128	7.847
AlphaRA Improvement	0.00%	0.10%	0.29%	9.43%

D. *Unique chips repaired by AlphaRA*

AlphaRA repairs many unique chips that are not repaired by other heuristic algorithms. Table 4 illustrates that our algorithm is able to repair 2055 chips (~ 4.11 %) that were not repaired by LECA and 1946 chips (~3.89%) not repaired by OSP. Although there are still 28 chips (~0.056%) repaired by LECA and 41 chips (~0.082%) repaired by OSP that are not repaired by AlphaRA, overall numbers suggest that our algorithm learns a strategy that is qualitatively different from the heuristics of LECA and OSP to repair the chips. We have demonstrated the end-to-end repair process of one such unique chip in the Appendix and compared it with the repair processes of LECA and OSP.

Table 4: Unique chips repaired by different algorithms.

Algorithm		Not Solved By			
		FLCA	LECA	OSP	**AlphaRA**
Solved By	FLCA	0	2096	1974	**69**
	LECA	0	0	1374	**28**
	OSP	0	1496	0	**41**
	AlphaRA	**0**	**2055**	**1946**	**0**

E. *Yield Comparison for Number of MCTS Simulations*

Changing the number of MCTS simulations has an impact on the normalized repair rate and runtime of the proposed algorithm. Fig. 8 (a) shows that as we increase the number of MCTS simulations to 64, AlphaRA performance increases to 99.9% of the FLCA repair rate. Even without any MCTS simulations, the number of chips repaired by AlphaRA is more than the existing heuristic algorithms. Although increasing the MCTS simulations increases the repair rate, the runtime of the algorithm also increases as shown in Fig. 8 (b).

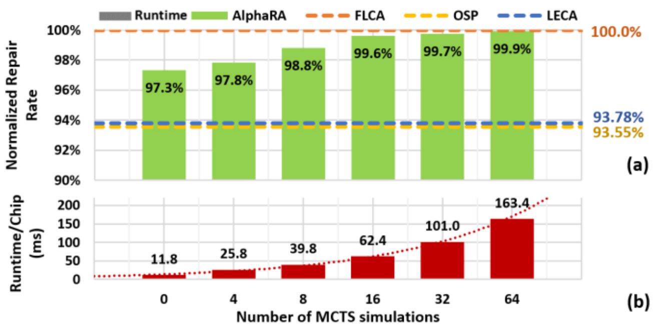

Fig. 8: (a) Normalized repair rate and (b) Algorithm runtime per chip with different number of MCTS simulations

F. *Performance and Scalability*

To demonstrate the scalability of AlphaRA on different chip sizes, we train it on 8×8 chips, as well as 32×32 chips. The results are shown in Fig. 9. The number of MCTS simulations used in these experiments is equal to the size of action space, i.e. 16 for 8×8, 32 for 16×16, and 64 for 32×32 chips. AlphaRA maintains a normalized repair rate above 98% across all the chip sizes under consideration. For 16×16 and 32×32 chip sizes, AlphaRA outperforms the heuristic algorithms by a margin of at least 4%.

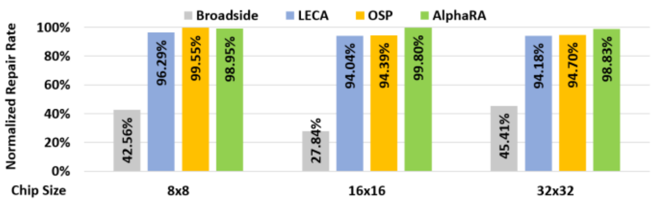

Fig. 9: Scalability of AlphaRA across different chip sizes

Since AlphaRA does not exhaustively search all moves, it scales better than exponential algorithms for larger chip sizes.

VI. Future Scope And Conclusion

Ongoing work aims to scale AlphaRA to larger chip sizes. Although AlphaRA learns through self-play reinforcement learning, we opine that faster convergence can be achieved by using the repair actions of the heuristic algorithms to first train the DNN in a supervised manner rather than initializing it randomly, similar to what has been used in AlphaGo.

In this paper, we have introduced AlphaRA, a first-of-its-kind AlphaZero based RA algorithm. We have compared it with existing heuristic and exhaustive algorithms against different performance metrics. With just 32 MCTS simulations, AlphaRA achieves an average normalized repair rate of 99.8% on 16×16 chips. It surpasses heuristic algorithms such as OSP by 5.42% while having the least spare utilization amongst all. It outperforms FLCA in terms of runtime, thus making it scalable for larger chip sizes. In addition, it maintains its performance with an increasing number of single faulty lines and has the ability to learn new qualitative strategies without any domain knowledge. Due to these merits, AlphaRA has the potential to be used in large-scale memory device manufacturing.

Acknowledgment

The authors would like to thank the reviewers, Arta Seify, Jeremy Salwen, Rajeev Verma, Ramya B T and Shubham Deshmukh for their helpful discussions and insights.

References

[1] S. K. Cho, K. Wooheon, C. Hyungjun, L. Changwook and K. Sungho, A Survey of Repair Analysis Algorithms for Memories, ACM Computer Survey, 2016.

[2] J. R. Day, "A fault-driven comprehensive redundancy algorithm," *IEEE Des. Test Comput.*, vol. 2, no. 3, p. 35–44, Jun. 1985.

[3] F. Lombardi and W. K. Huang, "Approaches for the repair of VLSI/WSI RRAMs by row/column deletion," *International Symposium on Fault-Tolerant Computing.*, pp. 342-347, 1988.

[4] J. Kim, K. Cho, W. Lee and S. Kang, "A new redundancy analysis algorithm using one side pivot," *International SoC Design Conference (ISOCC),* pp. 134-135, Jeju, 2014.

[5] D. Silver et. al., "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *ArXiv,* vol. abs/1712.01815, 2017.

[6] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," *Computers and Games, 5th International Conference,* pp. 72-83, 2006.

[7] Kocsis, Levente and C. Szepesvári, "Bandit based monte-carlo planning," *European conference on machine learning, Springer,* pp. 282-293, 2006.

[8] D. Silver, A. Huang, C. Maddison and e. al., "Mastering the game of Go with deep neural networks and tree search," *Nature 529,* pp. 484-489, 2016.

[9] D. Silver, J. Schrittwieser, K. Simonyan and e. al., "Mastering the game of Go without human knowledge.," *Nature,* vol. 550, p. 354–359, 2017.

[10] A. Seify, "Single-Agent Optimization with Monte-Carlo Tree Search and Deep Reinforcement Learning," *Ph.D dissertation, Dept.of Computing Science, University of Alberta,* 2020.

[11] H. Walker and S. W. Director, "VLASIC: A Catastrophic Fault Yield Simulator for Integrated Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* pp. 541-556, 1986.

[12] R. Huang, J. Yeh, J. Li and C. Wu, "Raisin: Redundancy Analysis Algorithm Simulation," *IEEE Design & Test of Computers,,* vol. 24, no. 4, pp. 386-396, 2007.

[13] Atishay, A. Gupta, R. Sonawat, H. K. Thacker and B. Prasanth, "SEARS: A Statistical Error and Redundancy Analysis Simulator," *27th International Conference on VLSI-SoC,* pp. 117-122, 2019.

[14] S. Nair, "Github - Alpha Zero General," [Online]. Available: https://github.com/suragnair/alpha-zero-general.

Appendix

In this section, we present a 16×16 chip from the test set which was a unique chip repaired by AlphaRA. Fig. 10 illustrates the memory chip, with 4 spare rows and columns along with faulty cells in red. Fig. 11 (a) to (g) show the sequence of actions performed to repair the faulty lines based on the raw action probabilities predicted by the AlphaRA neural network without action masking or MCTS simulations. In Fig. 11 (a), we observe that row 0 has 6 faults which are more than the number of available spare columns. Hence, it is in must-repair condition and has to be repaired by a spare row which is correctly identified by the neural network by giving it a very high probability of 0.993. After that, rows 14, 13, and 15 are repaired followed by columns 8, 7, and 10. This series of actions taken by AlphaRA allow it to repair the chip. In contrast, LECA and OSP leave 2 and 1 faults respectively leaving the chip unrepaired.

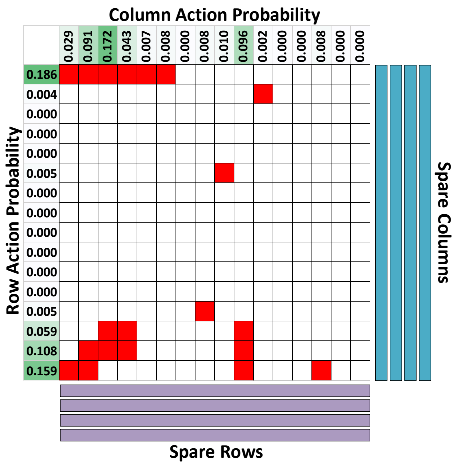

Fig. 10: The selected 16×16 chip with a 20 faults and 4 spare rows and columns. A sample row and column action probability is illustrated

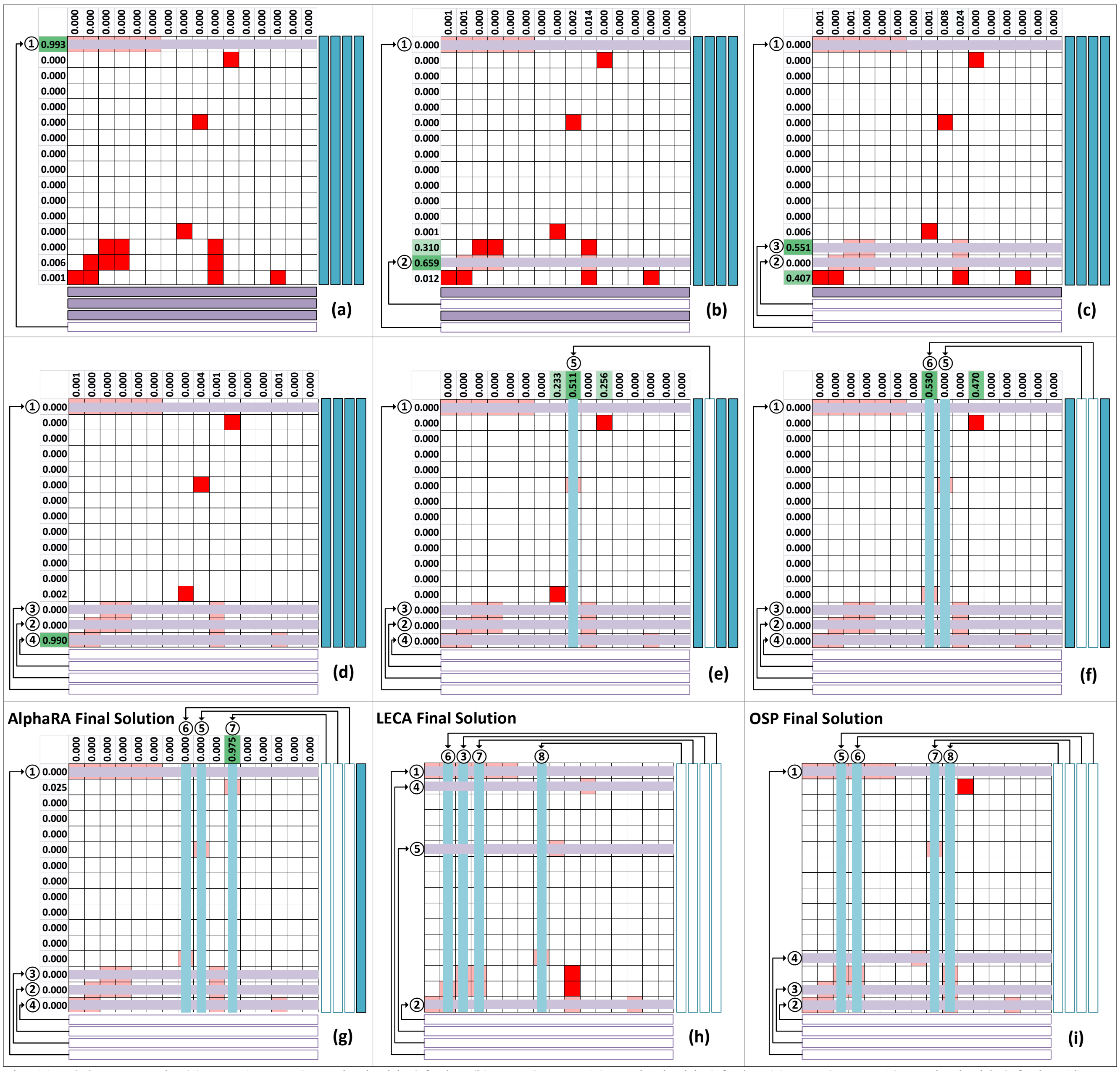

Fig. 11. AlphaRA repair, (a) Step 1: Row 0 repaired with 6 faults, (b) Step 2: Row 14 repaired with 4 faults, (c) Step 3: Row 13 repaired with 3 faults, (d) Step 4: Row 15 repaired with 4 faults, (e) Step 5: Column 8 repaired with 1 fault, (f) Step 6: Column 7 repaired with 1 fault, (g) Step 8: Column 10 repaired with 1 fault and AlphaRA complete repair solution, (h) LECA complete repair solution, (i) OSP complete repair solution.