In-Memory Computing of Akers Logic Array

Eitan Yaakobi
Electrical Engineering
California Institute of Technology
Pasadena, CA 91125
yaakobi@caltech.edu

Anxiao (Andrew) Jiang
Computer Science and Engineering
Texas A&M University
College Station, TX 77843
ajiang@cse.tamu.edu

Jehoshua Bruck
Electrical Engineering
California Institute of Technology
Pasadena, CA 91125
bruck@caltech.edu

***Abstract*—This work studies memories from a different perspective, while the goal is to explore the concept of in-memory computing. Our point of departure is an old study of logic arrays by Akers in 1972. We demonstrate how these arrays can simultaneously store information and perform logic operations.**

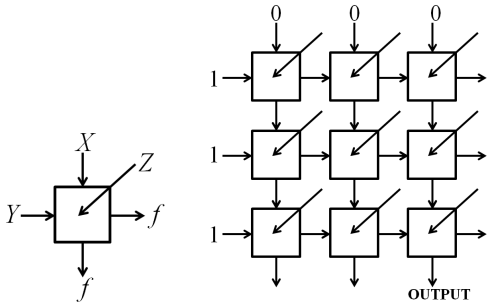
We first extend the structure of these arrays for non-binary alphabets. We then show how a special structure of these arrays can both store elements and output a sorted version of them. We also study other examples of the in-memory computing concept. In this setup, it is shown how information can be stored and computed with, and the array can tolerate or detect errors in the stored data.

I. Introduction

The common and traditional approach to think about a computer memory is as a device used to store information. In many cases this is also the correct perspective, especially when storing digital data. In fact, one can simplify the model of a computer to consist of two main entities: the processor that executes the logic and mathematical operations, and the memory that stores the data. The information is moved back and forth between the memory and the processor in order to execute logic operations and store them.

Let us consider here a different but very common type of memory, namely the human being's memory. We cannot find the equivalent analogy of a computer memory in a human memory. That is, our brain and memory do not apply the structure of two entities: one for storage and the other for executing logic operations. In fact, most studies in this area report a behavior of the brain's memory as a neural network that models an associative memory [2], [4], [13], [14], [17]. The memory is comprised of neurons, which both store information and communicate between them in order to process data. One common model of such a neural network is the Hopfield network [5]. This last model was later analyzed by McEliece *et al.* to show that the storage capacity of such networks is quite limited [15].

In this work, we take another step toward the study of memories that can also process logic operations, which we call *logic memories*. Our point of departure is the old study of *logic arrays* by Akers in 1972 [1], referred to as *Akers logic arrays* (or in short, *Akers arrays*) here. The atomic unit in Akers arrays is a cell with three inputs and two outputs, as illustrated in Fig. 1(a). There are two binary inputs X, Y to the cell and an additional binary input Z called the *control input*. There are two binary outputs from the cell having the same value f, which equals the majority of X, Y, and Z. An Akers array is formed by placing the cells in a two-dimensional array, as shown in Fig. 1(b). The inputs Y to the cells in the left-most column are all 1s, and the inputs X to the cells in the top row are all 0s. The output of the whole array is defined as the output of the cell in the down-right corner, whose value is determined by all the control inputs (given as inputs Z's to the cells).



(a) cell (b) array

Fig. 1. (a) The structure of a basic cell in the Akers logic array. (b) An example of a 3×3 logic array. The array's output is the output of the cell in the down-right corner.

We use an Akers array as an illustration of a memory that can both store information and perform logic computing. We assume the information stored in the array is carried by the control inputs to cells. The computed logic is the output of the array, and is a function of the control inputs. It has been shown by Akers that with the logic of such arrays, it is feasible to represent every boolean function [1]. This exemplifies how data can be stored and computed simultaneously. However, a disadvantage of the scheme is that the number of needed cells can be exponential in the number of variables. Nevertheless, Akers has presented certain types of functions, such as symmetric functions, that can be computed more efficiently.

The work by Akers paved the ground for more work on such types of logic arrays. Kukreja and Chen [12] considered extensions to the model, where again there are three inputs X, Y, Z and two outputs f_1, f_2, but with $f_1 = Y, f_2 = X$ if $Z = 1$ and $f_1 = X, f_2 = Y$ if $Z = 0$. They also considered extensions with time delays. Extensions to 3-dimensional arrays were presented in [3], [16]. Additional extensions to non-binary inputs and for diagnosing erroneous cells were shown in [7]–[10].

Our main goal in this paper is to build upon the work by Akers, and demonstrate examples for memory storage that can also carry out logic computing. In Section II, we review the model of Akers arrays, and state their useful properties and special examples. In Section III, we extend these arrays for

non-binary symbols. In particular, we study a special structure of the arrays that can both store and sort variables. In Section IV, we analyze the number of Akers arrays that can tolerate a single cell error. Section V brings another important example for storing data and computing logic functions. Here we show how with Akers arrays, one can store information and detect errors in the stored data. In Section VI, we present concluding remarks.

II. Akers Logic Arrays

In this section, we review the structure of Akers logic arrays. Its basic cell unit is shown in Fig. 1(a). The binary input variables X, Y are information that can come from other cells in the array. The binary *control input* Z is seen as the *value* stored in the cell. The output binary variables of one cell are used as inputs X, Y to adjacent cells. This forms the structure of the Akers logic array, as shown in Fig. 1(b). Let the array have size $m \times n$. The inputs Y to the left-most column are all 1s, and the inputs X to the top row are all 0s. Given the control inputs Z to every cell, the array performs computing, and its output is that of the cell in the down-right corner. According to this definition, it is possible to construct a mapping $F : \{0,1\}^{m\times n} \to \{0,1\}$ that assigns input $A = (a_{i,j})_{m\times n} \in \{0,1\}^{m\times n}$ to an $m \times n$ binary array, and outputs a logic-function value $F(A)$.

Under this array structure, Akers showed that the two inputs X, Y to every cell always satisfy the property that $X \leqslant Y$. Some basic techniques can be used to calculate the output of the array. The key is to observe the cell in the upper-left corner. If its *value* is 1, then the output of each cells in the left-most column is 1; thus the left-most column can be removed (without affecting the array's output). If its value is 0, then the output of each cell in the top row is 0; thus the top row can be removed. This process can be repeated until only a single row or column remains, when computing the final output becomes straightforward.

Akers also gave another useful tool to determine the output of an array. A binary array A of size $m \times n$ is said to have a *zero path* if there exist m positive integers $1 \leqslant j_1 \leqslant j_2 \leqslant \cdots \leqslant j_m \leqslant n$ such that $a_{1,j_1} = a_{2,j_2} = \cdots = a_{m,j_m} = 0$. That is, a zero path exists if there is a zero in every row with non-decreasing column indices. Similarly, A has a *one path* if there exist n positive integers $1 \leqslant i_1 \leqslant i_2 \leqslant \cdots \leqslant i_n \leqslant m$ such that $a_{i_1,1} = a_{i_2,2} = \cdots = a_{i_m,m} = 1$. Akers showed that an array has either a zero path or a one path, but not both. The following lemma gives a tool to calculate the array's output.

Lemma 1. [1] *Given an array A, if A has a zero path; then $F(A) = 0$; if A has a one path, then $F(A) = 1$.*

One of the goals of Akers by representing his logic arrays was to show that every boolean function can be calculated using this method. Even though the answer is affirmative, the number of needed cells for this task can be exponential in the number of variables. However, he showed an efficient implementation of a special family of functions: the *symmetric functions*. A boolean function of n variables $x_1, \ldots, x_n$ is called symmetric if its output is determined solely by the sum of the n variables, $\sum_{i=1}^{n} x_i$. Thus, every symmetric function f can be characterized by $n+1$ variables $w_0, w_1, \ldots, w_n$, where $f(x_1, \ldots, x_n) = w_k$ if $\sum_{i=1}^{n} x_i = k$. Given such a symmetric function f, an array A_f of size $(n+1)\times(n+1)$ is constructed as follows:

$$a_{i,j} = \begin{cases} x_{i+j-1} & \text{if } 2 \leqslant i+j \leqslant n+1 \\ w_{j-1} & \text{if } i+j = n+2 \\ \overline{x}_{i+j-(n+2)} & \text{if } n+3 \leqslant i+j \leqslant 2n+2 \end{cases}$$

An example for an array that calculates a symmetric function of three variables is shown in Fig. 2(a). Akers showed that the array A_f satisfies $F(A_f) \equiv f$.

x_1	x_2	x_3	w_3
x_2	x_3	w_2	$\overline{x}_1$
x_3	w_1	$\overline{x}_1$	$\overline{x}_2$
w_0	$\overline{x}_1$	$\overline{x}_2$	$\overline{x}_3$

(a) Array for symmetric function

x_1	x_2	x_3	x_4
x_2	x_3	x_4	
x_3	x_4		
x_4			

(b) Array for sorting

Fig. 2. (a) An example of an array that calculates a symmetric function with three variables. (b) An example of an array that sorts the variables x_1, x_2, x_3, x_4.

In particular, it is feasible to calculate the XOR of n variables $x_1, \ldots, x_n$. And in this special case, there can be another small improvement that reduces the array size to $n \times n$, because the XOR of n variables may be considered as a symmetric function of the first $n-1$ variables, where the values $w_0, w_1, \ldots, w_{n-1}$ satisfy $w_i = x_n$ if i is even and $w_i = \overline{x}_n$ if i is odd (for $0 \leqslant i \leqslant n-1$).

Another special type of array has a slightly different structure. This time the array does not have a rectangular shape, but the upper half array that is formed by the minor diagonal, as shown in Fig. 2(b). It was shown by Akers that for the n outputs of the last cell in each row, they are a sorted version of the input variables $x_1, \ldots, x_n$. We call such an array A a *sorting array* of order n. Its cells are (i, j) where $i + j \leqslant n + 1$, and the control input of the cell (i, j) is x_{i+j-1}.

III. Non-Binary Akers Arrays

In this section, we show how to generalize Akers arrays for non-binary alphabets. Assume that all variables can have q different values: $0, 1, \ldots, q-1$. We define the basic cell's output to be the median of its three inputs. That is, in Fig. 1(a)

$$f = \text{med}\{X, Y, Z\}.$$

The structure of the array remains the same, except that we let the input Y to every cell in the left-most column be $q-1$.

As a first step, we are interested in finding conditions that are equivalent to the zero and one paths in order to calculate the array's output. Let $A = (a_{i,j})_{m\times n}$ be an array in $\{0, 1, \ldots, q-1\}^{m\times n}$. $\forall\ 1 \leqslant i \leqslant m$, let u_i be the column index of the minimum element in the i-th row. That is for all $1 \leqslant j \leqslant n$, $a_{i,u_i} \leqslant a_{i,j}$. Similarly, for $1 \leqslant i \leqslant n$, let v_i be the row index of the maximum element in the i-th column. We say

that there is a *minimum path* if $1 \leqslant u_1 \leqslant u_2 \leqslant \cdots \leqslant u_m \leqslant n$ and there is a *maximum path* if $1 \leqslant v_1 \leqslant v_2 \leqslant \cdots \leqslant v_n \leqslant m$. Note that a zero path and a one path are special cases of a minimum path and a maximum path, respectively. For notational convenience, in this section, for a cell in position (i, j), we denote its inputs by $X_{i,j}, Y_{i,j}$ and $Z_{i,j} = a_{i,j}$, respectively, and denote its output by $f_{i,j}$.

Lemma 2. *An array cannot have both a minimum path and a maximum path.*

Theorem 3. *If an array A has a minimum path, then its output is $F(A) = \max_{1\leqslant i\leqslant m}\{a_{i,u_i}\}$. If it has a maximum path, then its output is $F(A) = \min_{1\leqslant j\leqslant n}\{a_{v_j,j}\}$.*

Proof: We prove this property for minimum paths by induction on the number of rows. The proof for maximum paths will be similar by induction on the number of columns.

We prove by induction that for an array A of size $m \times n$ with minimum path $1 \leqslant u_1 \leqslant u_2 \leqslant \cdots \leqslant u_m \leqslant n$, the output of every cell in the bottom row satisfies $f_{m,j} = \max_{1\leqslant i\leqslant m}\{a_{i,u_i}\}$ for $j \geqslant u_m$, and satisfies $f_{m,j} \geqslant \max_{1\leqslant i\leqslant m}\{a_{i,u_i}\}$ for $1 \leqslant j < u_m$. If there is only a single row, then it is immediate to see that if u_1 is the index of the minimum element in the top row, the output of the u_1-th cell as well as all consecutive cells is a_{1,u_1} and the output of the preceding cells is at least a_{1,u_1}.

Now assume the claim is true for arrays of size $m \times n$, and we will prove the correctness for arrays of size $(m + 1) \times n$. Let A be such an array with a minimum path $1 \leqslant u_1 \leqslant u_2 \leqslant \cdots \leqslant u_m \leqslant u_{m+1} \leqslant n$. According to the induction assumption, $f_{m,j} = \max_{1\leqslant i\leqslant m}\{a_{i,u_i}\} = M$ for $j \geqslant u_m$, and $f_{m,j} \geqslant M$ for $1 \leqslant j < u_m$. We consider the bottom row, and distinguish between two cases: 1) $a_{m+1,u_{m+1}} \leqslant M$; 2) $a_{m+1,u_{m+1}} > M$.

Case 1: In this case, we have $M = \max_{1\leqslant i\leqslant m+1}\{a_{i,u_i}\}$. For the first cell in the bottom row, we have $Y_{m+1,1} = q - 1$; and according to the induction assumption $X_{m+1,1} \geqslant M$. Hence $f_{m+1,1} \geqslant M$. The argument follows to all cell $a_{m+1,j}$ for $1 \leqslant j < u_{m+1}$. Since $X_{m+1,j} \geqslant M$ and $Y_{m+1,j} \geqslant M$, we also have $f_{m+1,j} \geqslant M$. Now we consider the cell $a_{m+1,u_{m+1}}$. We have that $Y_{m+1,u_{m+1}} \geqslant M$, $X_{m+1,u_{m+1}} = M$ and $Z_{m+1,u_{m+1}} \leqslant M$. Therefore, $f_{m+1,u_{m+1}} = M$. For all other cells in this row we have for $j > u_{m+1}$, $Y_{m+1,j} = X_{m+1,j} = M$ and this $f_{m+1,j} = M$.

Case 2: Now we have $a_{m+1,u_{m+1}} = \max_{1\leqslant i\leqslant m+1}\{a_{i,u_i}\} = M'$. Similarly to the proof of the first case, $Y_{m+1,1} = q-1$ and $Z_{m+1,1} \geqslant M'$ so $f_{m+1,1} \geqslant M'$. For every $1 \leqslant j < u_{m+1}$, $Y_{m+1,j} \geqslant M'$ and $Z_{m+1,1} \geqslant M'$ and thus $f_{m+1,j} \geqslant M'$. As for the u_{m+1}-th cell in this row, $Y_{m+1,u_{m+1}} \geqslant M'$, $X_{m+1,u_{m+1}} = f_{m,u_{m+1}} < M'$ and $Z_{m+1,u_{m+1}} = M'$. Therefore, $f_{m+1,u_{m+1}} = M'$. For the consecutive cells, for every $j > u_{m+1}$, we get $Y_{m+1,j} = M'$, $X_{m+1,j} = f_{m,j} < M'$ and $Z_{m+1,u_{m+1}} \geqslant M'$ and therefore $f_{m+1,j} = M'$. ∎

Our next task is to generalize the sorting property for non-binary symbols, for the sorting arrays defined in Section II.

Theorem 4. *Let $x_1, x_2, \ldots, x_n$ be n non-binary variables, and let A be a non-binary sorting array constructed with these variables. The outputs $f_{1,n}, f_{2,n-1}, \ldots, f_{n,1}$ satisfy:*

1) *$\{x_1, x_2, \ldots, x_n\} = \{f_{1,n}, f_{2,n-1}, \ldots, f_{n,1}\}$ with the same repetitions, i.e., the two multi-sets are equivalent.*
2) *$f_{1,n} \leqslant f_{2,n-1} \leqslant \cdots \leqslant f_{n,1}$.*

Proof: We prove this property by induction on n. It is simple to verify the conclusion for $n = 1$ and 2. Now suppose that the claim is true for sorting arrays with $n - 1$ variables; and we will prove the correctness for n variables.

Let $i_1, i_2, \ldots, i_{n-1}$ be a permutation of the indices $1, 2, \ldots, n - 1$ such that $x_{i_1} \leqslant x_{i_2} \leqslant \cdots \leqslant x_{i_{n-1}}$. According to the induction assumption, $f_{\ell,n-\ell} = x_{i_\ell}$ for $1 \leqslant \ell \leqslant n - 1$. Let us add two more variables $x_{i_0} = 0$ and $x_{i_n} = q - 1$. Now assume that for some $0 \leqslant k \leqslant n - 1$, $x_k \leqslant x_n \leqslant x_{k+1}$. For $1 \leqslant j \leqslant k$, $Y_{j,n+1-j} = x_{i_j} \leqslant x_n$, $X_{j,n+1-j} = x_{i_{j-1}} \leqslant x_n$ and $Z_{j,n+1-j} = x_n$. Thus, $f_{j,n+1-j} = \max\{x_{i_j}, x_{i_{j-1}}\} = x_{i_j}$. For $k + 2 \leqslant j \leqslant n$, $Y_{j,n+1-j} = x_{i_j} \geqslant x_k$, $X_{j,n+1-j} = x_{i_{j-1}} \geqslant x_k$ and $Z_{j,n+1-j} = x_k$. Thus, $f_{j,n+1-j} = \min\{x_{i_j}, x_{i_{j-1}}\} = x_{i_{j-1}}$. Similarly we also get that $f_{k+1,n-k} = x_n$. And we conclude that the array's outputs are sorted. ∎

IV. Analysis of Akers Arrays

In this section, we characterize some properties of Akers arrays. For simplicity, consider arrays of size $n \times n$. We say that an array $A \in \{0, 1\}^{n\times n}$ is *robust* (or more specifically, can tolerate one cell error) if its output does not change its value even if one of its cells changes its value. Namely, if we use $E_{i,j}$ to denote an $n \times n$ binary matrix that contains only one 1 in the (i, j) entry and all 0s elsewhere, then for every $E_{i,j} \in \{0, 1\}^{n\times n}$, a robust array A satisfies $F(A) = F(A \oplus E_{i,j})$. (Here "$\oplus$" is the XOR operation.)

Robust arrays are important for fault tolerance in memories. Note that unlike error-correcting codes, where the reliability of a codeword is achieved through its distance to other codewords, here the robustness is achieved through the array's own value. Let $\mathcal{A}_n \subseteq \{0, 1\}^{n\times n}$ be the set of *all* robust arrays of size $n \times n$. Let $\mathcal{C}_\mathcal{A}$ be the capacity of robust arrays, that is, $\mathcal{C}_\mathcal{A} = \lim_{n\to\infty} \frac{\log_2 |\mathcal{A}_n|}{n^2}$.

We observe that if $F(A) = 0$, then the array A has a zero path and in this case, A is robust if and only if it has two non-intersecting zero-paths. The same analogy applies to arrays with one-paths. Now note that if the first two columns of an array are all 0s, then it has two non-intersecting zero paths and thus is robust. So $|\mathcal{A}_n| \geqslant 2^{n^2-2n}$ and $\mathcal{C}_\mathcal{A} = 1$. Therefore, it is more interesting to evaluate the difference between $\log_2 |\mathcal{A}_n|$ and n^2, which we denote by $r_n \triangleq n^2 - \log_2 |\mathcal{A}_n|$.

Lemma 5. *For any array $A \in \{0, 1\}^{n\times n}$, $f(A) = 1 - f(\overline{A}^T)$. And the number of zero-paths in A equals the number of one-paths in $\overline{A}^T$.*

Lemma 5 shows it is sufficient to count the number of robust arrays with output 0. We denote this set by $\mathcal{A}_n^0$. r_n can be computed using next lemma.

Lemma 6.

$$|\mathcal{A}_n^0| = \sum_{q_1=2}^{n} \sum_{q_2=q_1}^{n} \cdots \sum_{q_n=q_{n-1}}^{n} \prod_{i=1}^{n} (q_i - 1) 2^{n^2 - \sum_{i=1}^{n} q_i}.$$

Proof: $\forall\ A \in \{0,1\}^{n\times n}$, $A \in \mathcal{A}_n^0$ if and only if A has two different zero paths. Every zero path can be characterized by a vector $\boldsymbol{p} = (p_1, \ldots, p_n)$ such that every p_i corresponds to the column index in the i-th row where its value is 0, namely $a_{i,p_i} = 0$, and $1 \leqslant p_1 \leqslant p_2 \leqslant \cdots \leqslant p_n \leqslant n$.

For an array $A \in \mathcal{A}_n^0$, let $\boldsymbol{p} = (p_1, \ldots, p_n)$ be the leftmost zero-path and $\boldsymbol{q} = (q_1, \ldots, q_n)$ be the second leftmost zero-path. Hence $\boldsymbol{p} < \boldsymbol{q}$ and in each row, all the entries before the q_i-th entry besides the p_i-th entry are 1s. That is, for all $1 \leqslant i \leqslant n$ and for all $1 \leqslant j < q_i$ and $j \neq p_i$, we have $a_{i,j} = 1$. All other entries in each row can have any value. Therefore, we get $|\mathcal{A}_n^0| = \sum_{q_1=2}^{n} \sum_{q_2=q_1}^{n} \cdots \sum_{q_n=q_{n-1}}^{n} \prod_{i=1}^{n} (q_i - 1) 2^{n^2 - \sum_{i=1}^{n} q_i}$. ∎

Theorem 7. $\lim_{n\to\infty} \frac{r_n}{n} = 2$.

V. Akers Array for In-Memory Error Detection

In this section, we study an application of Akers array: In-memory error detection. The objective is to connect stored data to in-memory computing elements, so that errors that appear in data will be detected immediately (and then corrected). Akers arrays have highly regular structures, which help their fabrication in memories. And as we have shown, they can realize symmetric functions, such as XOR, relatively efficiently, which is useful for error detection.

Error detection is an important part of *memory scrubbing*, a key function that ensures data reliability in memories. Conventionally, data are stored using an error-correcting code (ECC). To detect errors, the data need to be read from the memory into the CPU, and decoding is performed there. Since reading causes delay and cannot be performed frequently, the ECC is often designed to tolerate many errors. However, infrequent reads enhance error accumulation, which increases the chance the codeword cannot be correctly decoded, and also makes decoding more time consuming on average. It will be helpful to have a mechanism for the memory to check the codeword frequently, and see if it has errors. More specifically, as soon as one or a few errors appear, we would like the memory to detect them immediately, then trigger the CPU's decoding procedure and have the errors removed from the data. This will prevent error accumulation and keep the codeword error free most of the time. Only in the rare event that many errors appear nearly simultaneously may the error detection mechanism fail (because the number of errors exceeds what the mechanism guarantees to detect). In that rare case, we resort to the conventional approach, namely, to wait for the CPU to read the codeword and correct all the errors. The above approach can significantly increase the time-to-failure for data storage. With in-memory hardware implementation, the error detection can be fast, frequent and power efficient. In the following, we study how to use Akers arrays to efficiently detect errors, given the ECC used to store data.

Let $\mathcal{C} \subseteq \{0,1\}^n$ be an ECC. Let $(x_1, \cdots, x_n) \in \mathcal{C}$ be a generic codeword. Let $P_1, P_2, \cdots, P_r \subseteq \{1, 2, \cdots, n\}$ be its parity-check constraints; specifically, $\forall\ i = 1, \cdots, r$, $\oplus_{j\in P_i} x_j \equiv 0$. We associate P_i with a weight $w_i > 0$, which is the cost of realizing the parity-check function $\oplus_{j\in P_i} x_i$ using an Akers array. If the Akers array's output is 1, errors are detected. Since the Akers array has size $|P_i| \times |P_i|$ (as we showed before), we let $w_i = |P_i|^2$.

Our objective is to choose a subset $S \subseteq \{1, 2, \cdots, r\}$ of parity-check constraints that can detect t errors, where t is a given parameter. (If $i \in S$, then P_i is chosen.) Given this constraint, we need to minimize the total weight $\sum_{i\in S} w_i$.

Let $1 \leqslant i \leqslant t$, and let $\{j_1, j_2, \cdots, j_i\} \subseteq \{1, 2, \cdots, n\}$ with $j_1, j_2, \cdots, j_i$ being i distinct numbers. We call $\{j_1, j_2, \cdots, j_i\}$ an *error pattern of size* i. Let E_i be the set of all error patterns of size i. Clearly, $|E_i| = \binom{n}{i}$.

Theorem 8. *A set of parity-check constraints $S \subseteq \{1, 2, \cdots, r\}$ can detect t errors if and only if $\forall\ i \in \{1, 2, \cdots, t\}$ and $e \in E_i$, there exists $s \in S$ such that $|P_s \cap e|$ is odd.*

We can therefore formulate the error detection problem as the following variation of the weighted set cover problem. Let $E_1 \cup E_2 \cup \cdots \cup E_t$ be a universe of elements. $\forall 1 \leqslant i \leqslant r$, we define $Q_i \subseteq (E_1 \cup E_2 \cup \cdots \cup E_t)$ as a subset that consists of all the error patterns P_i can detect. That is, for any error pattern of size $k \leqslant t$ – say $\{j_1, j_2, \cdots, j_k\} \in E_k$, – it is in Q_i if and only if $|P_i \cap \{j_1, j_2, \cdots, j_k\}|$ is odd. If an error pattern is in Q_i, we call it *covered* by Q_i. It is not hard to see that

$$|Q_i| = \sum_{j=1}^{t} \sum_{k=1,3,5,\cdots,2\cdot\lfloor\frac{\min\{j,|P_i|\}-1}{2}\rfloor+1} \binom{|P_i|}{k} \binom{n-|P_i|}{j-k}.$$

Then, our objective is to choose a set $S \subseteq \{1, 2, \cdots, r\}$ with the minimum value of $\sum_{i\in S} w_i$ such that $\cup_{i\in S} Q_i = \cup_{i\in\{1,\cdots,t\}} E_i$.

We use a well known greedy approximation algorithm [11] to solve the above error detection problem (due to its NP hardness). Let U denote the set of error patterns that are uncovered yet. Initially, let $U = (E_1 \cup E_2 \cup \cdots \cup E_t)$. Repeatedly choose a subset $Q_i \in \{Q_1, \cdots, Q_r\}$ that minimizes $\frac{w_i}{|Q_i \cap U|}$ (i.e., the weight w_i divided by the number of uncovered error patterns in Q_i) and update U as $U - Q_i$, until all the error patterns are covered. The algorithm has time complexity $O(n^{2t})$. For memory scrubbing, t is usually a small constant, so the complexity is $poly(n)$.

We now analyze the performance of the above algorithm. Let S_t^* be an optimal solution, and let $W_t^* = \sum_{i\in S_t^*} w_i$ be its weight. Let S_t be the solution of the above greedy algorithm, and let $W_t = \sum_{i\in S_t} w_i$ be its weight. For any positive integer m, let $H_m = \sum_{i=1}^{m} \frac{1}{i}$ be the m-th harmonic number. By the well known analysis on set covering [11], we see that the approximation ratio $\frac{W_t}{W_t^*}$ is at most $H_{\max_{i\in\{1,\cdots,r\}} |Q_i|}$, which is $O(\log n)$ (with t viewed as a constant).

The practical performance, however, can be substantially better, especially for low-density parity-check (LDPC) codes. Let $d = \max_{i\in\{1,\cdots,r\}} |P_i|$; namely, d is the maximum cardinality of a parity-check constraint.

Theorem 9.

$$\frac{W_t}{W_t^*} \leqslant \frac{W_t}{W_1} \cdot H_d.$$

Proof: We have $W_t/W_t^* \leqslant W_t/W_1^* = (W_t/W_1) \cdot (W_1/W_1^*) \leqslant (W_t/W_1) \cdot H_d$. ∎

Theorem 10. $\frac{W_t}{W_t^*} \leqslant \frac{W_t}{dn}$.

A more refined upper bound than Theorem 10 is presented below. Consider irregular LDPC codes. For $i = 2, 3, \cdots, d$, let ρ_i denote the fraction of parity-check constraints of cardinality i. (So for $i = 2, 3, \cdots, d$, there are $r\rho_i$ parity-check constraints of cardinality i.) Let γ be the smallest integer in $\{2, 3, \cdots, d\}$ such that $\sum_{i=2}^{\gamma} i\rho_i \geqslant \frac{n}{r}$.

Theorem 11.

$$\frac{W_t}{W_t^*} \leqslant \frac{W_t}{r\sum_{i=2}^{\gamma-1} i^2\rho_i + \gamma^2\lceil\frac{n-r\sum_{i=2}^{\gamma-1} i\rho_i}{\gamma}\rceil}.$$

Proof: Let $S_1^* = \{i_1, i_2, \cdots, i_k\}$ be an optimal solution for $t = 1$. For $j = 1, 2, \cdots, k$, define $U_i \triangleq \{1, \cdots, n\} - \cup_{z=1}^{j-1} P_{i_z}$. (By default, let $U_1 = \{1, \cdots, n\}$.) $\forall\, j = 1, 2, \cdots, k$ and $z \in P_{i_j} \cap U_j$, define $p_z \triangleq \frac{|P_{i_j}|^2}{|P_{i_j} \cap U_j|} \geqslant |P_{i_j}|$ and call it the *price* of the zth codeword bit. It is not hard to see that $W_1^* = \sum_{j \in S_1^*} w_j = \sum_{j=1}^{n} p_j$. For $j = 2, 3, \cdots, d$, there are $r\rho_j$ parity-check constraints of cardinality j, and they can make less than or equal to $r\rho_j \cdot j$ codeword bits have price j (where the equality holds only if all the $r\rho_j$ parity-check constraints of cardinality j are selected in the optimal solution and the sets of codeword bits they cover are mutually disjoint). So it is not difficult to see that $W_t^* \geqslant W_1^* = \sum_{j=1}^{n} p_j \geqslant \sum_{j=2}^{\gamma-1}(j \cdot r\rho_j) \cdot j + \gamma^2\lceil\frac{n-r\sum_{j=2}^{\gamma-1} j\rho_j}{\gamma}\rceil$, which leads to the final conclusion. ∎

We verify the above upper bounds in practice through experiments with LDPC codes. Some typical results are shown in Fig. 3, for $t = 2$. The x-axis of the figure represents the rates of LDPC codes, and the y-axis represents the upper bounds to the approximation ratio W_t/W_t^*. Here codes of length $n = 1200$ and 9 different rates are shown (with rates 0.5, 0.67, 0.75, 0.8, 0.83, 0.86, 0.875, 0.889 and 0.9, respectively), using $(3, 6)$-, $(3, 9)$-, $(3, 12)$-, $(3, 15)$-, $(3, 18)$-, $(3, 21)$-, $(3, 24)$-, $(3, 27)$-, $(3, 30)$-regular LDPC codes, respectively. The curve at the top is the worst-case upper bound $H_{max_{i \in \{1,\cdots,r\}}|Q_i|}$. The curve at the bottom corresponds to the two practical upper bounds in Theorem 9 and Theorem 10. (We take the minimum of the two.) For each code rate, 50 codes were constructed, and a 100% confidence interval is shown. (Each interval is very small and seems like a point in the figure.) It can be seen that the actual approximation ratio in practice is much better than the worst-case approximation ratio.

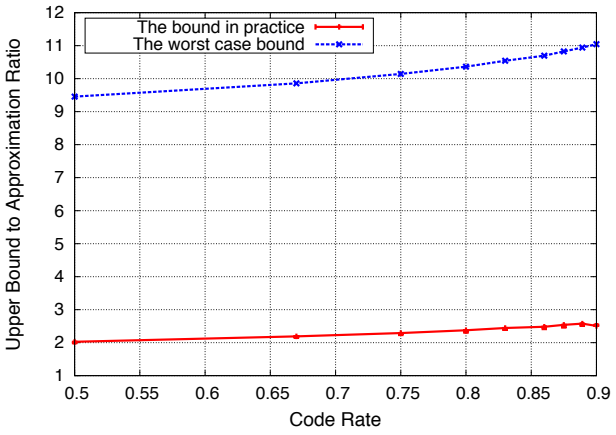


Fig. 3. Comparison between the upper bounds in practice and the worst-case upper bound to the approximation ratio W_t/W_t^*, for $t = 2$.

VI. Conclusions

This paper studies the concept of in-memory computing through the study of Akers arrays. Generalized Akers arrays for sorting high-alphabet symbols are presented. The capacity of robust Akers arrays that tolerate one error is analyzed. And Akers arrays' application to in-memory error detection is studied. A number of important open problems still exist. In particular, it is interesting to understand the information theoretic bounds to the performance of in-memory computing using Akers arrays and similar regular-structured arrays. They remain as our future research directions.

References

[1] S. B., Akers, "A rectangular logic array," *IEEE Transactions on Computers*, vol. C-21, no. 8, pp. 848–857, August 1972.

[2] S. Amari, "Neural theory of association and concept formation," *Biol. Cybern*, vol. 26, pp. 175–185, 1977.

[3] J. B. Bate and J. C. Muzio, "Three cell structure for ternary cellular arrays," *IEEE Trans. Computers*, vol. C-26, no. 12, pp. 1191–1202, 1977.

[4] G. E. Hinton and J. A. Anderson, eds., *Parallel Models of Associative Memory*, Hillsdale, NJ: Erlbaum, 1981.

[5] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proc. Nat. Acad. Sci. USA*, vol. 79, pp. 2554–2558, 1982.

[6] S. N. Kukreja and I. Chen, "Combinational and sequential cellular structures," *IEEE Trans. Computers*, vol. C-22, no. 9, pp. 813–823, 1973.

[7] N. Kamiura, Y. Hata, F. Miyawaki, and K. Yamato, "Easily testable multiple-valued cellular array", *Proc. 22nd Int. Symp. on Multiple-valued Logic*, pp. 36–42, 1992.

[8] N. Kamiura, Y. Hata, and K. Yamato, "Design of repairable cellular array on multiple-valued logic", *IEICE Trans. Inf. & Syst.*, vol. E77-D, no. 8, pp. 877–884, 1994.

[9] N. Kamiura, Y. Hata, and K. Yamato, "Design of fault-tolerant cellular arrays on multiple-valued logic", *Proc. 24th Int. Symp. on Multiple-Valued Logic*, pp. 297–304, 1994.

[10] N. Kamiura, H. Satoh, Y. Hata, and K. Yamato, "Design in Fault Isolating of Ternary Cellular Arrays Using Ternary Decision Diagrams", *Proc. 3rd Asian Test Symposium*, pp. 201–206, 1994.

[11] J. Kleinberg and E. Tardos, *Algorithm Design*, 1st *ed.*, Chapter 11.3, Pearson Education Inc., 2006.

[12] S. N. Kukreja and I. Chen, "Combinational and sequential cellular structures," *IEEE Trans. Computers*, vol. C-22, no. 9, pp. 813–823, 1973.

[13] W. A. Little, "The existence of persistent states in the brain," *Math. Biosci.*, vol. 19, pp. 101–120, 1974.

[14] W. A. Little and G. L. Shaw, "Analytic study of the memory storage capacity of a neural network," *Math. Biosci.*, vol. 39, pp. 281–290, 1978.

[15] R. J. McEliece, E. C. Posner, E. R. Rodemich, and S. S. Venkatesh, "The capacity of the Hopfield associative memory," *IEEE Trans. on Information Theory*, vol. 33, no. 4, pp. 461–482, 1987.

[16] D. M. Miller and J. C. Muzio, "A ternary cellular array," *Proc. Int. Symp. on Multiple-Valued Logic*, pp. 469–482, 1974.

[17] G. Palm, "On associative memory," *Biol. Cybern*, vol. 36, pp. 19–31, 1980.