REFERENCES

[1] M. K. M. Ali and F. Kamoun, "Neural networks for shortest path computation and routing in computer networks," *IEEE Trans. Neural Networks,* vol. 4, pp. 931–940, Nov. 1993.

[2] B. Linares Barranco, E. Sanchez-Sinenco, A. Rodriguez-Vazquez, and J. L. Huertas, "A modular T-mode design approach for analog neural network hardware implementations," *IEEE J. Solid State Circuits*, vol. 27, pp. 701–713, May 1992.

[3] D. Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computation.* Englewood Cliffs, NJ: Prentice Hall, 1989.

[4] V. S. Borkar, "Topics in controlled Markov chains," *Pitman Research Notes in Mathematics*, No 240, Longman Scientific and Technical, Harlow, England, 1991.

[5] A. Bouzerdoum and T. R. Pattison, "Neural network for quadratic optimization with bound constraints," *IEEE Trans. Neural Networks*, vol. 4, pp. 293–304, 1993.

[6] R. W. Brockett, "Dynamical systems that sort lists, diagonalize matrices, and solve linear programming problems," in *Linear Algebra and its Applications*, vol. 146, pp. 79–91, 1991.

[7] ——, "Least squares matching problems," Center for Intell. Contr. Syst. Rep. CICS-P-133, Harvard Univ., MA, Apr. 1989.

[8] R. W. Brockett and W. S. Wong, "A gradient flow for the assignment problem," Center for Intell. Contr. Syst. Rep. CICS-P-284, Harvard Univ., MA, Feb. 1991.

[9] L. O. Chua and G. N. Liu, "Nonlinear programming without computation," *IEEE Trans. Circuits Syst.*, vol. 31, pp. 182–188, 1984.

[10] L. O. Chua and L. Yang, "Cellular neural networks: Theory and applications," *IEEE Trans. Circuits. Syst.*, vol. 35, pp. 1257–1290, 1988.

[11] T. Roska and L. O. Chua "The CNN universal machine: An analogic array," *IEEE Trans. Circuits Syst.*, vol. 40, pp. 163–173, Mar. 1993.

[12] C. Chiu, C. Y. Maa, and M. A. Shanblatt, "Energy function analysis of dynamic programming neural networks," *IEEE Trans. Neural Networks*, vol. 2, pp. 418–426, July 1991.

[13] M. T. Chu, "On the continuous realization of iterative processes," *SIAM Rev.*, vol. 30. no. 3, Sept. 1988.

[14] A. Cichocki and R. Unbehauen, "Neural networks for solving systems of linear equations—Part II: Minimax and least absolute value problems," *IEEE Trans. Circuits Syst. II,* vol. 39 pp. 619–633, 1991.

[15] J. Cronin, *Differential Equations: Introduction and Qualitative Theory.* New York: Marcel Dekker, 1994, second ed.

[16] C. B. Garcia and W. I. Zangwill, *Pathways to Solutions, Fixed Points and Equilibria.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[17] D. G. Kelly, "Stability in contractive neural networks," *IEEE Trans. Biomed. Eng.*, vol. 37, pp. 231–242, Mar. 1990.

[18] M. K. Kennedy and L. O. Chua, "Neural networks for nonlinear programming," *IEEE Trans. Ciruits Syst.*, vol. 35, pp. 554–562, 1988.

[19] K. Matsuoka, "Stability conditions for nonlinear continuous neural networks with asymmetric connection weights," *Neural Networks,* vol. 5, pp. 495–500, 1992.

[20] S. Smale, "A convergent process of price adjustment and global newton methods," *J. Math. Economics*, vol. 3, pp. 107–120, 1976.

[21] G. R. Sell, *Topological Dynamics and Ordinary Differential Equations.* London, England: Van Nostrand Reinhold, 1971.

[22] L. Zhang and S. C. A. Thomopoulos, "Neural network implementation of the shortest path algorithm for traffic routing in communication networks," in *Proc. Int. Joint Conf. Neural Networks*, June 1989, p. II 591.

[23] M. Vidyasagar, *Nonlinear System Analysis.* Englewood Cliffs, NJ: Prentice Hall, 1992.

Efficient Implementation of Neighborhood Logic for Cellular Automata via the Cellular Neural Network Universal Machine

Kenneth R. Crounse, Eula L. Fung, and Leon O. Chua

***Abstract*— The main difficulty in implementing cellular automata on the Cellular Neural Network Universal Machine (CNNUM) is the need to perform arbitrary logic functions of the input neighborhood. Since the architecture computes weighted sums of this neighborhood, by using a "B-template," it is limited to threshold logic, i.e., a logical operation to be computed by a single transient must be in the class of linearly separable Boolean functions. It was shown previously how a general logic function can be implemented on the CNNUM by cascading component functions from this class—namely by the direct implementation of the minterm or maxterm formulation of the desired function. However, for functions of a 3 × 3 input neighborhood this method may require up to 256 stages. We propose a more efficient method for implementing general logic functions on the CNNUM and other hardwares capable of performing a threshold logic function of an input neighborhood. The class of considered component functions is a superset of the minterms and maxterms but, for purposes of searchability, ease of implementation, and robustness, a subset of the general linearly separable Boolean functions. We have formulated an algorithm that will find a sequence of *weight-restricted* threshold logic functions (B-templates with weights from {-1, 0, +1} and a bias) that, when cascaded together using two-input logical operations, will result in the desired Boolean function. Two examples are given to exhibit the algorithm.**

I. INTRODUCTION

Cellular automata are an important class of array dynamical systems which are discrete in space, state, and time. They have been shown to be useful in modeling physical systems [1], morphological image processing [2], and random number generation [3].

The main task in evolving a particular cellular automaton rule is, in a space-invariant and synchronous manner, to evaluate a given Boolean function of the state of a cell and its immediate neighbors. The result of this Boolean function is then stored as the next state of the cell and the whole procedure is iterated. When a different cellular automaton state transition rule is to be implemented, a different Boolean function must be specified.

The cellular automaton evolution is inherently a highly parallel operation, and many specialized hardwares have been developed exploiting this fact ([4], for instance). When implementing an arbitrary cellular automaton in circuitry, there is a tradeoff between the space required and processing time. At one extreme is the bit-serial processor approach, by which the whole array is updated serially. At the other extreme is the array approach, by which simple processors are assigned one-per-cell and consult a local look-up table in order to update the states.

We propose a method for implementing an arbitrary cellular automaton rule on an architecture which rests somewhere between these extremes. The processing units are placed one-per-cell, but, due to implementation concerns, a unit is not given enough computational capability to implement an arbitrary logic function in a single time

Manuscript received March 10, 1996. This work was supported by the Joint Services Electronics Program, Contract F49620-94-C-0038. The work of E. L. Fung was supported by the National Science Foundation. This paper was recommended by Associate Editor B. Sheu.

The authors are with the Electronics Research Laboratory, Electrical Engineering and Computer Sciences Department, University of California, Berkeley, CA 94720 USA.

Publisher Item Identifier S 1057-7122(97)02068-0.

step. Instead, the processors are time multiplexed to perform a single cellular automaton iteration by means of a short sequence of simple operations.

Architectures capable of implementing the proposed method in a massively parallel fashion are the Cellular Neural Network Universal Machine (CNNUM) and the Discrete-Time CNN [5], [6]. Both of these processors are more general than necessitated by our approach and allow for many other image processing computations. However, the robust implementation of cellular automata and neighborhood logic on such machines is an important question since these operations are often used as stages of more complicated algorithms. Alternatively, larger arrays to run programmable general cellular automata might be built by tailoring the hardware to specialize in the proposed method.

It has long been known that threshold logic can be used to implement the class of linearly separable Boolean functions. And, that by cascading the outputs of threshold logic units, arbitrary Boolean functions can be built [7], [8]. The idea of applying such cascading approaches to neighborhood logic on the CNN and CNNUM was first developed in the works and ideas of Venetianer [9], [10], (and personal communications) and in the multilayer CNN context by Shi [11], [12]. These concepts were later used and fomalized to show that any neighborhood logic funciton could, in fact, be implemented by a CNNUM algorithm [13].

Our approach follows the threshold logic methodology but restricts the weights to be taken from $\{-1, 0, +1\}$. Under this restriction, only N thresholds need to be implemented for Boolean functions of N-inputs—all others are redundant. To minimize the chance of errors due to noise it is wise to choose threshold values which are furthest from the possible sums formed by the unit, which in this case can more easily be done by using $2N - 1$ different thresholds. The advantage of such an approach (over general threshold logic) is threefold: The architecture need not implement an arbitrary analog multiplication and threshold, the weight space can be exhaustively searched by present-day computers, and the hardware would be more robust to noise. The main disadvantage is that, presumably, longer template sequences will be needed than would be required by using general threshold logic. However, in practice, finding such sequences for general threshold logic is a very difficult task [8] and the algorithms for doing so are necessarily sub-optimal. Since the number of Boolean functions in the weight-restricted class is explicitly searchable, it is expected that the possibility of more efficient algorithms for finding template sequences will make up some of the difference. In addition, allowing for *arbitrary* two-input Boolean functions for composing outputs of the threshold units provides further degrees of freedom.

II. Background

A. Cellular Automata Rules

The states of a cellular automaton (CA) evolve according to a state transition rule—a function which determines the next state of each cell as a function of the current state of the cell and its neighbors.

In this brief we will discuss two-dimensional first-order binary-state cellular automata. The states of a binary CA can take on one of two values which we will designate as {0,1} where 0 can be thought of as logical FALSE and 1 as logical TRUE. The state transition rule is a Boolean function of the current states of the cell and its 8-nearest neighbors. The state evolution of the automaton to the next time step can be written as

$$s_{i,j}(n+1) = f[s_{\mathcal{N}_{i,j}}(n)]$$

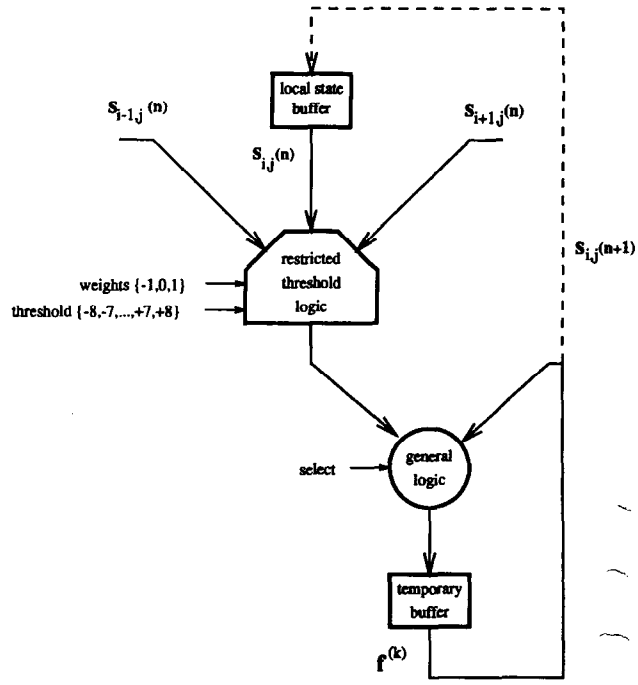

Fig. 1. Required architecture for one cell of a restricted threshold-logic machine. Only two of the lines from the eight neighboring state buffers are shown.

where $\mathcal{N}_{i,j}$ are the indices for the neighborhood of cell (i,j) and f, in our case a Boolean function of nine variables, is the state transition rule. A total of $2^9 = 512$ neighborhood configurations are possible. The Boolean function which defines the transition rule must specify a next state for each of these configurations. Since a cell can have two possible next states there are $2^{512} \approx 10^{154}$ possible cellular automata rules in the considered class.

B. Restricted Threshold Logic Machine

We are proposing a technique for implementing any of these cellular automata in array hardware. The minimal architecture required for the proposed method is shown in Fig. 1. The machine has a massively parallel cellular structure, of which one cell is shown.

Each cell has a binary buffer which can be accessed by neighboring cells. For our purposes here, it is convenient to let -1 denote logical FALSE and $+1$ denote logical TRUE. These buffers must be able to be supplied to the input of a *restricted-parameter threshold logic unit*. Mathematically, the unit accomplishes the function

$$\operatorname{sgn}\left(\sum_{k,l=-1}^{1} w(k,l)s_{i+k,j+l}(n) - I_0\right)$$

where the weights $w(k,l)$ must be chosen from $\{-1, 0, 1\}$. Note that since the weighted sums formed by the unit must be integral and will jump in steps of two (although they will be either even or odd depending on the number of weights which are zero), choosing $I_0 \in \{-8, -7, \cdots, +7, +8\}$ is a well-spaced choice for thresholds. The restricted parameter set is one of the unique aspects of this hardware which allows for ease of implementability as well as robustness.

The output of the threshold unit must be able to be combined with the contents of a temporary storage binary buffer through a general selectable two-input logic function. Finally, the contents of the temporary buffer must be able to be transferred back to the state buffer.

These requirements can be satisfied by some more complex architectures such as the CNNUM and some implementations of the Discrete-Time CNN. For instance, a standard CNN is controlled by three parameters: an A-template, a B-template, and a bias term I. To use the CNN to perform the sign function, all values of the A-template are set to zero except the center element which is set to one. The B-template and bias together define the particular threshold logic function of the CNN input [11]. For an 8-neighbor configuration, using

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} w(-1,-1) & w(-1,0) & w(-1,1) \\ w(0,-1) & w(0,0) & w(0,1) \\ w(1,-1) & w(1,0) & w(1,1) \end{bmatrix}$$

$$\mathbf{I} = -I_0.$$

will give the corresponding threshold logic computation at the steady-state CNN output. The Universal Machine is used to provide buffering and general two-input local logic.

III. Weight Generation Algorithm

Fig. 1 hints at how we intend to use the architecture to implement a cellular automaton. The current states $s_{i,j}(n)$ are stored in the state registers. The restricted threshold logic unit is used to perform a series of logic functions $b^{(k)}$ on the current state. Each of these is sequentially combined with the previously composed result through a two-input logic function $\odot^{(k)}$. After M iterations, the desired logic function will have been performed and the result is fed-back to the state register as the next state. Namely,

$$s_{i,j}(n+1) = f^{(M)}[s_{\mathcal{N}_{i,j}}(n)] \tag{1}$$

where

$$\begin{aligned} f^{(0)}[s_{\mathcal{N}_{i,j}}(n)] &= b^{(0)}[s_{\mathcal{N}_{i,j}}(n)] \\ f^{(1)}[s_{\mathcal{N}_{i,j}}(n)] &= f^{(0)}[s_{\mathcal{N}_{i,j}}(n)] \odot^{(1)} b^{(1)}[s_{\mathcal{N}_{i,j}}(n)] \\ f^{(2)}[s_{\mathcal{N}_{i,j}}(n)] &= f^{(1)}[s_{\mathcal{N}_{i,j}}(n)] \odot^{(2)} b^{(2)}[s_{\mathcal{N}_{i,j}}(n)] \\ &\cdots \\ f^{(M)}[s_{\mathcal{N}_{i,j}}(n)] &= f^{(M-1)}[s_{\mathcal{N}_{i,j}}(n)] \odot^{(M)} b^{(M)}[s_{\mathcal{N}_{i,j}}(n)]. \end{aligned}$$

Given an arbitrary state transition rule f, we want to be able to efficiently implement it by such a decomposition. Thus, the problem we would like to solve is: for an arbitrary given state transition rule (neighborhood Boolean function), find the shortest sequence (smallest M) of $b^{(k)}$ from this restricted-weight threshold logic class and corresponding two-input logic operations such that the resulting function $f^{(M)}$ is the given transition rule.

A. Characterization of Restricted-Weight Threshold Logic Class

We will express an input u to the Boolean function f as the N-tuple $u = (u_0, u_1, \cdots, u_{N-1}) \in U = \{0,1\}^N$. (For our examples, $N = 9$.) Then, the Boolean function can be expressed as a 2^N-tuple listed in the natural order when interpreting the input N-tuple as the binary representation of an integer. That is, we can write the Boolean function $f(u) = (f_0, f_1, \cdots, f_{2^N-1})$ such that $f(u) \in F = \{0,1\}^{2^N}$.

Some special Boolean functions should be noted. Minterms are Boolean expressions that are true for only one input. Specifically, there are 2^N such minterms, indexed by $i \in U$ defined by

$$m_i(u) = \begin{cases} 1, & \text{if } u = i \\ 0, & \text{otherwise.} \end{cases}$$

The maxterms are the dual to minterms in that they evaluate to 0 if a single given input occurs and 1 otherwise. Then, similarly, the maxterms can be defined as

$$M_i(u) = \begin{cases} 0, & \text{if } u = i \\ 1, & \text{otherwise.} \end{cases}$$

It is an elementary observation that any Boolean function can be expressed as a sum (OR) of minterms or a product (AND) of maxterms. To express a Boolean function in minterms, simply add all minterms corresponding to inputs that make the function evaluate to 1. For maxterms, the Boolean function is equivalent to multiplying all maxterms that correspond to inputs that make the function evaluate to 0.

We will now attempt to quantify the behavior of the restricted-weight threshold logic unit. Let us define the distance between input N-tuples as

$$\text{dist}(u,v) = \sum_{i=0}^{N-1} u_i \oplus v_i \quad \text{for } u, v \in U$$

where $\oplus$ denotes the XOR operation that returns 1 if $u_i \neq v_i$ and 0 if $u_i = v_i$. Let the threshold logic unit have weights w, of which z are zero, and possible thresholds $I_0 = (N - z - 1 - 2k)$ for $k = 0, \cdots, N - z - 1$.

First, consider the case when there are no zero weights, i.e., $z = 0$ Then, when $k = 0$ it is easily shown that the unit implements the minterm m_w, where we have allowed a slight abuse of notation by considering the $\{-1, 1\}$-valued weights w to be a $\{0, 1\}$-valued vector in U. Decreasing the threshold, by increasing k, will allow more inputs to be accepted by the unit. Namely, the unit will return TRUE for the set of inputs $\{u : u \in U, \text{dist}(u,w) \leq k\}$. A Boolean function of this form will be called a *ballterm* and can be written

$$b_{w,k} = \text{OR } m_v \quad \text{for } v \in U \text{ s.t. } \text{dist}(v,w) \leq k.$$

Intuitively, this function can be visualized as detecting all inputs within a ball of radius k centered at the input w.

When $z \neq 0$ the unit will ignore some inputs. Then, the unit with $k = 0$, giving a threshold of $(N - z - 1)$, is equivalent to an ORing of minterms in the same way that a circled implicant on a Karnaugh map is equivalent to the sum of the minterms circled. For example, a set of weights $w = (w_0, w_1, \cdots, w_{N-1})$ with $w_0 = 0$ and $w_i \in \{-1, +1\}$ for $i = 1, \cdots, N-1$ acts the same as the following sum of minterms

$$(-1, w_1, \cdots, w_{N-1}) \text{ OR } (+1, w_1, \cdots, w_{N-1}).$$

For $k \neq 0$ the template will detect all minterms that are at a distance less than or equal to k from the implicant that would be detected when $k = 0$. For example, suppose $N = 4$ and our set of weights is $(+1, 0, -1, +1)$. A threshold of 2 (from $k = 0, z = 1$) will give the implicant equivalent to the minterm template combination $(+1, +1, -1, +1)$ OR $(+1, -1, -1, +1)$, both with thresholds of 3. Now if we consider a threshold of 0 (from $k = 1, z = 1$), the resulting template will be equivalent to the combination

$$\begin{aligned} &(+1,+1,-1,+1) \text{ OR } (+1,-1,-1,+1) \text{ OR} \\ &(-1,+1,-1,+1) \text{ OR } (-1,-1,-1,+1) \text{ OR} \\ &(+1,+1,+1,+1) \text{ OR } (+1,-1,+1,+1) \text{ OR} \\ &(+1,+1,-1,-1) \text{ OR } (+1,-1,-1,-1) \end{aligned}$$

with each set of weights thresholded at $(N - 1) = 3$. The last six sets of weights describe templates that detect minterms a distance 1 away from the implicant detected with $k = 0$.

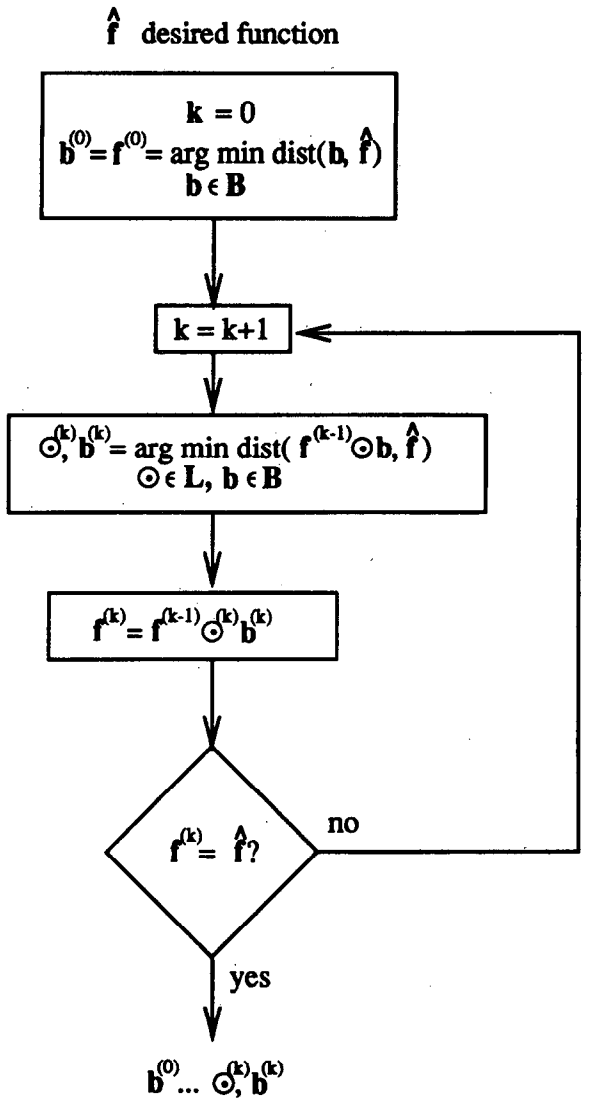

Fig. 2. Flowchart for the template-finding algorithm. The input is the desired Boolean function. The algorithm finds a sequence of functions from the restricted-weight threshold logic class and associated two-input combining logic to implement the desired function.

B. Description of Decomposition Algorithm

Having made these observations we can address the first question: does a solution of the form of (1) exist? The answer is yes, and a constructive argument can be found in [13]. The important point of the argument is: by using weights in $\{-1,+1\}$ and a threshold of $+(N-1)$ one can implement any minterm with the restricted threshold logic unit. Similarly, with weights in $\{-1,+1\}$ and a threshold of $-(N-1)$ any maxterm can be implemented. Thus we only need the local two-input logical operations OR and AND and the min/maxterm class of templates to construct a solution to implementing a desired Boolean function. A solution of this form would require at most 2^{N-1} templates.

By using ballterms, one would expect to be able to find solutions which use fewer templates than required in a purely minterm or maxterm formulation because we are considering a larger class of component functions. We have an additional weight value, 0, which allows a template to ignore some inputs. In addition, the ability to assign threshold values other than the $\pm(N-z-1)$ used for the min/maxterms allows even more flexibility.

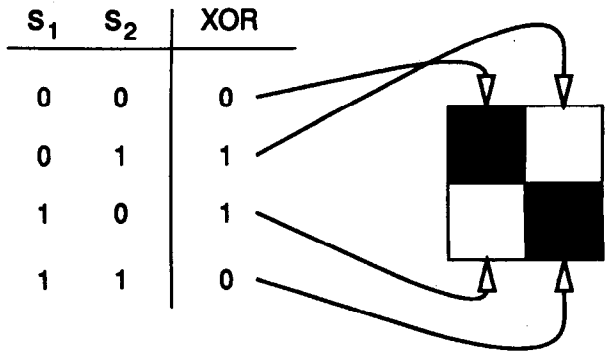

Fig. 3. A demonstration of the pictorial Boolean function notation used in this brief. The truthtable of a Boolean function (in this case the exclusive-OR (XOR)) is mapped to a graph by converting successive regions of the truth table to rows of the graph, as shown, giving a compact representation.

With this in mind, we would like to create an algorithm to automatically find a "good" solution for us. Of course, to find the best solution an exhaustive search of longer and longer sequences of templates and combining logic could be performed until the desired function is found, but such a thought is untenable with current computing technology. At the other extreme, an algorithm could simply generate all the minterms of the given Boolean function and OR them together. Clearly there is a tradeoff between the running time of the algorithm and the length of the resulting template sequence. We propose the following algorithm to demonstrate the existence of a practical approach to design template sequences which takes advantage of the restricted-weight threshold class with arbitrary two-input combining logic. Many such algorithms are possible since, in general, more than one solution exists.

The following greedy algorithm is proposed to decompose a Boolean function into the form of (1). Generate the set B of all the Boolean functions that can be implemented with weights in $\{-1, 0, +1\}$ and appropriate biases. Let L designate the set of two-input Boolean functions used for combining elements in B. Fig. 2 shows a flowchart of the algorithm. First, the function $b^{(0)} \in B$ of minimum distance to the desired function is found. Our notion of distance is similar to the one defined above except that we apply it to Boolean functions rather than inputs, that is

$$\text{dist}_F(f,g) = \sum_{u=0}^{2^N-1} f(u) \oplus g(u) \text{ for any } f, g \in F.$$

As long as the distance between the currently composed function $f^{(k-1)}$ and the desired funciton is greater than zero, the algorithm iteratively searches B and L for a function $b^{(k)}$ and logic operation $\odot^{(k)}$ that will modify the previous composed function (in the manner of (1)) to result in one of minimal distance to the desired function.

The algorithm is guaranteed to converge to a solution since in every iteration prior to convergence the error can decrease by at least one. This can be seen by observing that even in the worst case the current function $f^{(k)}$ can be brought closer to the desired one by either ORing with a minterm or ANDing with a maxterm.

C. Examples

In this section we will show some examples of algorithm results. For readability, we are going to display the 2^N-digit Boolean functions pictorially as a rectangular array of 2^N squares colored either black or white. Starting at the top left with $f(0)$, we will fill in each square row by row, coloring it black if $f(i) = 0$ or white if $f(i) = 1$. For instance, the XOR function, $f_{\text{XOR}} = (0, 1, 1, 0)$, could be depicted as shown in Fig. 3. In the following examples $N = 9$. Each Boolean function will be depicted with $f(0) - f(31)$ displayed in the first row; the last entry, $f(511)$, is in the bottom right corner.

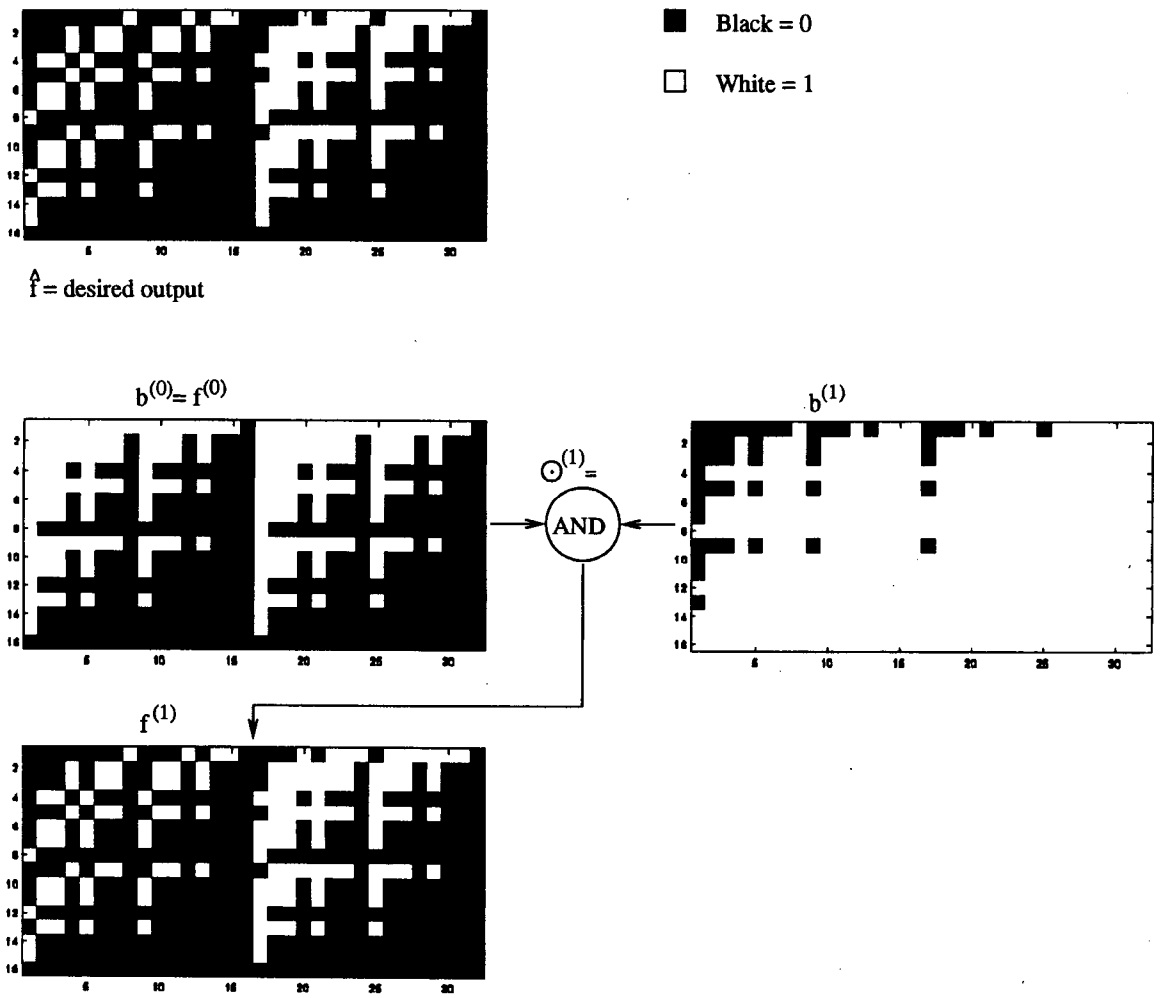

Fig. 4. Template-finding algorithm output for the game of life CA. The 512-entry truth tables for the desired and output Boolean function are shown pictorially.

With nine inputs, there are 118 098 Boolean functions in the set B. However, since each ballterm also has its complement in this set, only half of them need to be generated for searching since the complements are easily generated during the search. Each truth table requires 512 b, requiring a total of less than 4 MB of storage. For these examples, the set L of two-input logic functions was chosen to be {AND, OR, XOR}, a decision which is discussed in Section IV.

Example 1—Game of Life: Fig. 4 shows the algorithm results for implementing the game of life, which is a famous two-dimensional cellular automata rule with many interesting properties [14].

The resulting ballterms (weights and trhreshold) and logic operations are

$$b^{(0)} = b_{(-1,-1,-1,-1,0,-1,-1,-1,-1),+1}$$
$$b^{(1)} = b_{(+1,+1,+1,+1,+1,+1,+1,+1,+1),-4}$$
$$\odot^{(1)} = \text{AND}$$

To implement these results on the CNN, we would create a B_0 template to implement the $b^{(0)}$ ballterm and a B_1 template for the $b^{(1)}$ ballterm as shown below

$$\mathbf{B}_0 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 0 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad \mathbf{I}_1 = -1$$

$$\mathbf{B}_1 = \begin{bmatrix} +1 & +1 & +1 \\ +1 & +1 & +1 \\ +1 & +1 & +1 \end{bmatrix} \quad \mathbf{I}_1 = +4.$$

Each set of templates is applied to the input, and the results are ANDed together to get the final answer. These results are identical to those designed informally in [10].

Example 2: Fig. 5 shows the algorithm's results for another sample function.

The resulting ballterms and logic operations are

$$b^{(0)} = b_{(-1,-1,0,+1,0,-1,-1,0,-1),3}$$
$$b^{(1)} = b_{(-1,-1,+1,-1,0,+1,-1,+1,-1),5}$$
$$b^{(2)} = b_{(+1,-1,0,0,-1,0,-1,0,+1),-2}$$
$$b^{(3)} = b_{(-1,0,-1,-1,-1,0,+1,-1,0),5}$$
$$\odot^{(1)} = \text{XOR}$$
$$\odot^{(2)} = \text{AND}$$
$$\odot^{(3)} = \text{OR}.$$

Therefore, this chosen neighborhood logic function could be implemented using a sequence of four CNN templates and three local logic operations.

IV. Performance Considerations

Many modifications could be made to improve the performance of the weight-finding algorithm. For instance, the case often occurs during the search that more than one function in the table will result in the same minimal distance. For convenience, the algorithm simply picks the first such function it encountered. This choice of function,

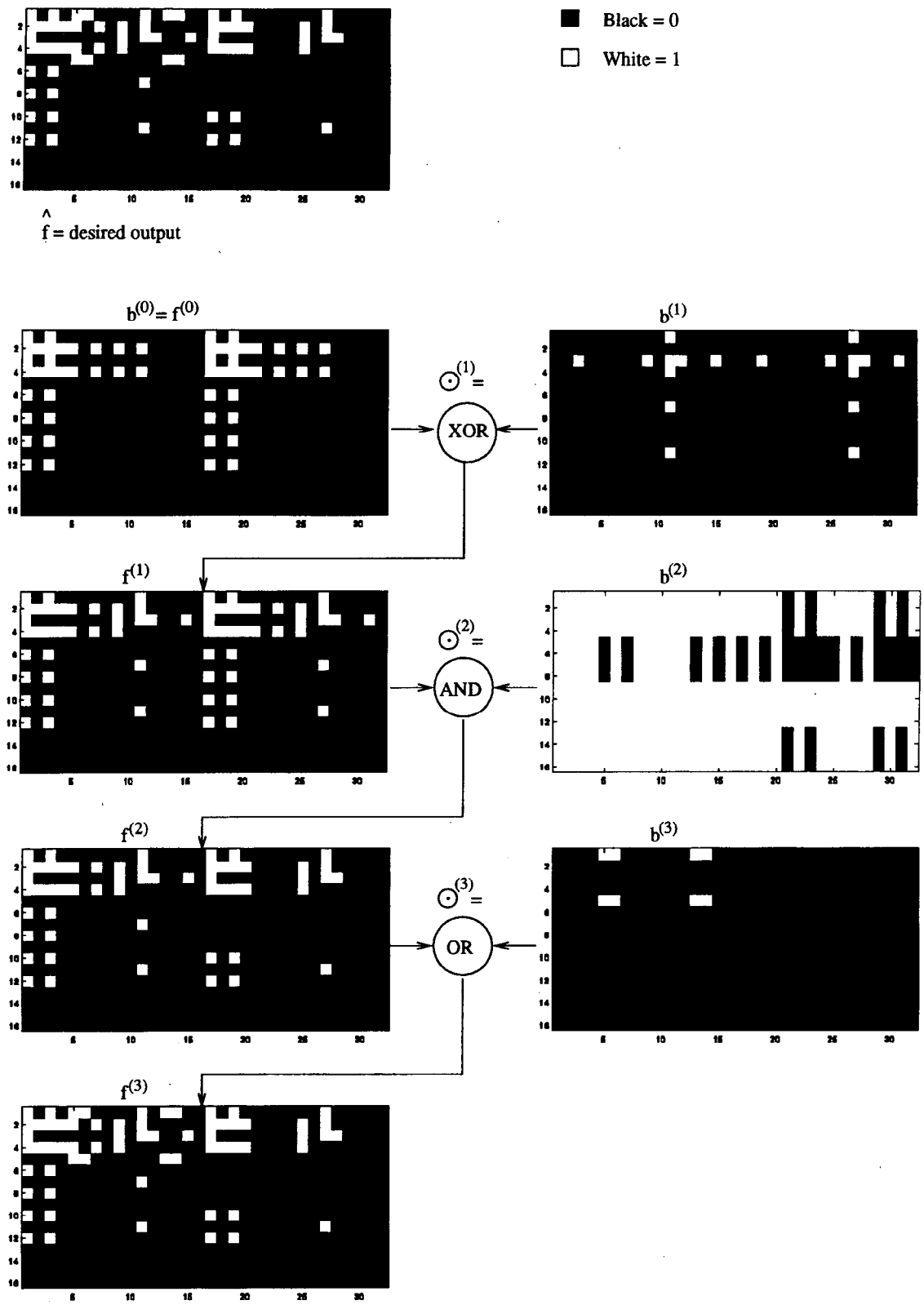

Fig. 5. Template-finding algorithm output for a sample input function. The desired function was found to be implementable in four restricted-weight threshold logic stages. The 512-entry truth tables for the desired logic function, the restricted-weight threshold logic functions, and the intermediate results after each stage are shown here pictorially.

however, can make a difference in the final number of templates needed. Random testing indicates that the improvement gained by searching all paths with same minimal distance is typically not significant (a few templates) and for demonstration purposes did not justify the significant increase in search time. When finding templates for permanent use however, a breadth-first search which keeps all templates giving the same minimal distance at each level may be expedient.

Another consideration is to restrict the number of two-input combining logic functions in the set L for quicker convergence. It is clear that TRUE, FALSE, identity of either input, and complement of either input, need not be considered. Furthermore, two-input functions which complement $b^{(k)}$ are not needed since the complements are also ballterms. But, in fact, during simulation it was found that combining functions which complement $f^{(k-1)}$ never arose (although we have not proven that they are not needed) leaving only the AND, OR, and XOR functions.

The choice of distance measure also affects the outcome. The distance measure described above looks for the best fit by weighting both TRUE (1) and FALSE (0) equally so that the best fit function may have errors which output 1 when it should output 0 and vice versa. The operation OR can only modify a function by changing 0's to 1's its truth table while the operation AND can only change 1's to 0's; only XOR can add both 1's and 0's. Thus this distance measure shows no preference between OR and AND since both could be used to decrease the distance.

Suppose instead we defined a distance measure that considers only functions that must output TRUE every time the desired function outputs TRUE but may output TRUE or FALSE when the given function outputs FALSE. The best fit function would then be the one that outputs TRUE the fewest times when the desired output is FALSE. Such a distance measure would favor the logical operation AND since in subsequent steps, 0's need to be added. In the same manner, we could also define a distance measure that favors OR. (Due to the difficulties of searching the weight-space of the general threshold logic functions, the method given in [8] is restricted to this last type of distance measure.)

In certain situations, such favoritism would be desirable. For example, for $N = 9$ consider the Boolean function that outputs TRUE every time exactly four input variables are 1. Using the new AND-favoring distance measure, the algorithm finds a solution using only two templates while the original distance measure results in a solution with over 70 templates. A more complex algorithm could perform a search by using each of these measures.

Even if an optimal algorithm could be designed to choose template sequences from this class, we do not know how many steps might be needed in the worst-case to implement a desired function. A simple counting argument gives 28 steps as a lower upper-bound on the minimal number. That is, there is some logic function that requires at least this many steps even when optimally implemented. If the upper bound is in fact much greater than this, it may be necessary to explore techniques which drop the weight restriction.

V. Conclusion

Choosing from among the enormous number of threshold logic functions of nine variables makes composing general Boolean functions with them a daunting task. We have shown how by restricting the threshold logic functions to those with $\{-1, 0, +1\}$ weights, the component functions can be reduced to a manageable number. The concept of *ballterm* was introduced to characterize such Boolean functions. An algorithm was presented for finding a sequence of templates and two-input logical operators that will implement any desired neighborhood Boolean function on the CNNUM by using this weight-restricted class. The algorithm does not necessarily choose the optimal set from the ballterm class and sometimes produces long sequences. However, the generated sequences are shorter than those produced by a direct minterm/maxterm implementation. Therefore, the proposed technique represents a significant step toward the efficient automation of template design for the implementation of cellular automata and general neighborhood logic.

References

[1] T. Toffoli, *Cellular Automata Machines.* Cambridge, MA: The MIT Press, 1987.

[2] K. Preston, Jr. and M. J. B. Duff, *Modern Cellular Automata: Theory and Applications.* New York: Plenum, 1984.

[3] S. Wolfram, "Random sequence generation by cellular automata," *Advances in Applied Mathematics*, vol. 7, pp. 123–169, 1986.

[4] A. P. Marriott, P. Tsalides, and P. J. Hicks, "VLSI implementation of smart imaging system using two-dimensional cellular automata," *Proc. Inst. Elect. Eng. G*, Circuits, Devices and Systems, vol. 138, pp. 582–586, Oct. 1991.

[5] T. Roska and L. O. Chua, "The CNN Universal Machine: An analogic array computer," *IEEE Trans. Circuits Syst. II*, vol.40, pp. 163–173, Mar. 1993.

[6] L. O. Chua and T. Roska, "The CNN paradigm," *IEEE Trans. Circuits Syst. I*, vol. 40, pp. 147–156, Mar. 1993.

[7] P. M. Lewis II and C.L. Coates, *Threshold Logic* . New York: Wiley, 1967.

[8] T. A. M. Kevenaar, "PLANET: A hierarchical network simulator," Ph.D. dissertation, Eindhoven Univ. Technol., The Netherlands, 1992.

[9] P. L. Venetianer, P.Szolgay, K. R. Crounse, T.Roska, and L.O. Chua, "Analogue combinatorics and cellular automata — Key algorithms and layout design," *Int. J. Circuit Theory Applicat.*, vol. 24, pp. 145–164, Jan.-Feb. 1996.

[10] L. O. Chua, T. Roska, and P. L. Venetianer, "The CNN is as universal as the Turing Machine," *IEEE Trans. Circuits Syst. I*, vol. 40, pp. 289–291, 1993.

[11] L. O. Chua and B. E. Shi, "Exploiting Cellular Automata in the design of Cellular Neural Networks for binary image processing," Memo. UCB/ERL M89/130, Univ. Calif., Berkeley, Electron. Res. Lab., Nov. 1989.

[12] ——— "Multiple layer Cellular Neural Networks —A tutorial," in *Algorithms and Parallel VLSI Architectures*, F. Deprettere and A. Van der Veen, Eds. Amsterdam, The Netherlands: Elsevier Sci., 1991, pp. 137–168.

[13] K. R. Crounse and L. O. Chua, "The CNN universal machine is as universal as a turing machine," *IEEE Trans. Circuits Syst. I*, vol. 43, pp. 353–355, Apr. 1996.

[14] E. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for your Mathematical Plays.* New York: Academic, 1982, vol. 2, ch. 25, pp. 817–850.