Copyright © 1995, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

HARDWARE IMPLEMENTATION OF CELLULAR AUTOMATA VIA THE CELLULAR NEURAL NETWORK UNIVERSAL MACHINE

by

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

Memorandum No. UCB/ERL M95/111

27 November 1995

COVER PAGE

HARDWARE IMPLEMENTATION OF CELLULAR AUTOMATA VIA THE CELLULAR NEURAL NETWORK UNIVERSAL MACHINE

by

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

Memorandum No. UCB/ERL M95/111

27 November 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Hardware Implementation of Cellular Automata via the Cellular Neural Network Universal Machine

Kenneth R. Crounse * Eula L. Fung † Leon O. Chua ‡

November 27, 1995

Abstract

The main difficulty in implementing cellular automata on the Cellular Neural Network Universal Machine (CNNUM) [1, 2] is the need to perform arbitrary logic functions of the input neighborhood. Since the architecture computes weighted sums of the input neighborhood (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[3] how a general logic function can be implemented 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 may require up to 256 stages.

We propose a more efficient method of performing general logic functions on the CNNUM and other hardwares capable of performing a threshold logic function. The class of considered component functions is larger than the minterms and maxterms but, for purposes of searchability, ease of implementation, and robustness, smaller than 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 $\{-1, 0, +1\}$ weights and I 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.

1 Introduction

Cellular automata are an important class of array dynamical systems which are discrete in space, state, and time. They have shown to be useful in modeling physical systems [4], morphological image processing [5], and random number generation [6].

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 [7, for instance].

*Sponsored under the Joint Services Electronics Program, Contract Number F49620-94-C-0038. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation herein.

†Sponsored by the National Science Foundation.

‡The authors are with the Electronics Research Laboratory, University of California at Berkeley, 94720.

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, the computational capability is not enough to implement an arbitrary logic function in a single time 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 and the Discrete-time CNN. 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 could 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. By cascading the outputs of threshold logic units, arbitrary boolean functions can be built [8].

Our approach follows the threshold logic methodology, but further restricts the class to those with $\{-1, 0, +1\}$ weights. 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 be easily done by using $2N - 1$ thresholds. The advantage of such an approach is threefold: The architecture needs not to implement an arbitrary analog multiplication and threshold, the weight space can be exhaustively searched by present-day computers, and the hardware is more robust to noise. The disadvantage is an algorithm with greater time complexity, and some additional control circuitry. In order to reduce the increase in time, we are allowing arbitrary boolean combination of binary-weighted threshold logic function, which requires only a simple two-input universal logic unit at each cell.

2 Background

2.1 Cellular Automata Rules

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

In this paper we will discuss two-dimensional first-order cellular automata. The states of a first-order 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; and the state transition rule is a 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

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

where $\mathcal{N}_{i,j}$ are the indices for the neighborhood of cell (i, j) and f, in our case a boolean function of 9 variables, is the state transition rule. A total of $2^9 = 512$ neighborhood configurations are

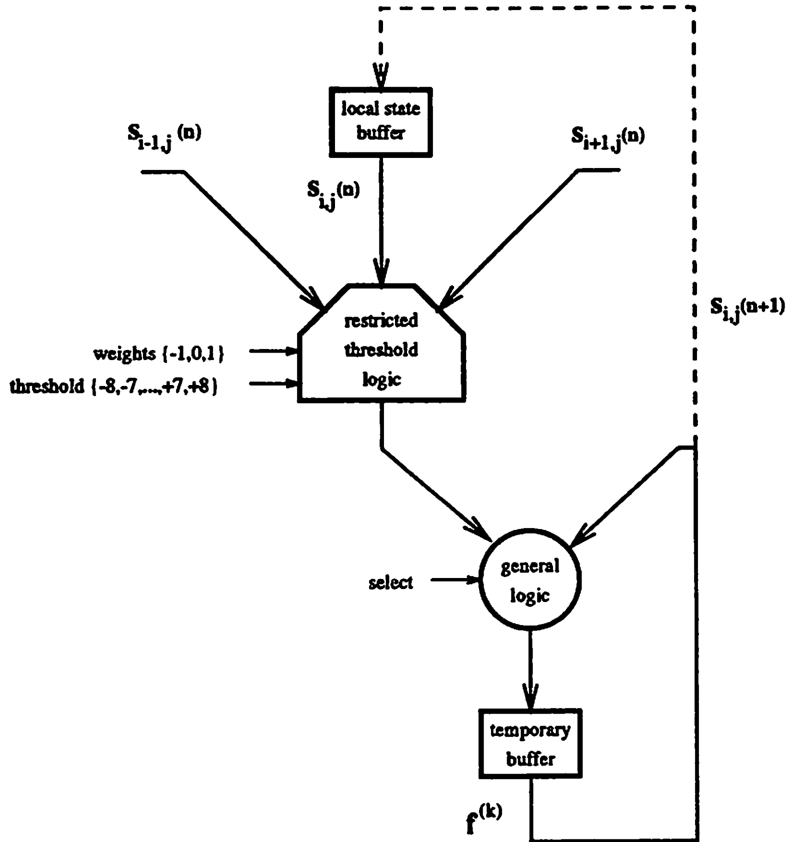

Figure 1: Minimal architecture for one cell of a restricted threshold logic machine. Only two of the lines from the eight neighboring state buffers are shown.

possible. The boolean function which defines the transition rule must specify a next state for each of these neighborhoods. Since each can have two possible next states there are therefore $2^{512} \approx 10^{154}$ possible cellular automata rules in the considered class.

2.2 Restricted Threshold Logic Machine

We are proposing a technique for implementing any of these cellular automata in array hardware. The minimal architecture necessary for the proposed method is shown in Figure 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 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

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

where the weights $w(k,l)$ can 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 $I_0 \in \{-8,-7,\ldots,+7,+8\}$ 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 are minimal and can be satisfied by some more complex architectures, such as the CNN Universal Machine and some implementations of the Discrete-time CNN. For instance, the evolution of states in the CNN is controlled by three parameters: an A-template, a B-template, and a threshold value I_0. To use the CNN to perform linear thresholding [9], we set all values of the A-template to zero except the center element which is set to one. The B-template defines the linearly separable boolean function. Thus for an 8-neighborhood configuration we have

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} w(-1,-1) & w(-1,0) & w(-1,1) \\ w(0,-1) & w(0,0) & w(0,1) \\ w(1,-1) & w(1,0) & w(1,1) \end{bmatrix} \qquad \mathbf{I} = -I_0$$

3 Weight Generation Algorithm

Figure 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 previous 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,

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

$$\begin{aligned} f_0[s_{\mathcal{N}_{i,j}}(n)] &= b^{(0)}[s_{\mathcal{N}_{i,j}}(n)] \\ f_1[s_{\mathcal{N}_{i,j}}(n)] &= f_0[s_{\mathcal{N}_{i,j}}(n)] \odot^{(1)} b^{(1)}[s_{\mathcal{N}_{i,j}}(n)] \\ f_2[s_{\mathcal{N}_{i,j}}(n)] &= f_1[s_{\mathcal{N}_{i,j}}(n)] \odot^{(2)} b^{(2)}[s_{\mathcal{N}_{i,j}}(n)] \\ &\dots \\ f_M[s_{\mathcal{N}_{i,j}}(n)] &= f_{M-1}[s_{\mathcal{N}_{i,j}}(n)] \odot^{(M)} b^{(M)}[s_{\mathcal{N}_{i,j}}(n)] \end{aligned}$$

Given an arbitrary state transition rule f, we want to be able to implement by such a decomposition. Thus our problem is, for an arbitrary given state transition rule (which is a boolean function), to find a sequence of $b^{(k)}$ in this restricted-weight threshold logic class and corresponding two-input logic operations such that the resulting function f_M is the given transition rule.

3.1 Characterization of Restricted-Weight Threshold Logic Class

We will express an input, u, to the boolean function, f, as the N-tuple $u = (u_0, u_1, \dots, u_{N-1}) \in U = \{0,1\}^N$. (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 $f(u) = (f_0, f_1, \dots, f_{2^N-1})$ 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 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 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.

Having made these observations we can address the first question: does a solution of the form of Equation 3 exist? The answer is yes, and a constructive argument can be found in [3]. 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 logical operations OR and AND and the min/maxterm class of templates to construct a solution to our problem. A solution of this form would require at most 2^{N-1} templates.

In general, we 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 templates. We have an additional weight value, 0, which allows a template to ignore the value of some inputs. A template with some zero weights and a threshold of $-(N-z-1)$ where z is the number of zero weights in the template is equivalent to an ORing of minterm templates in the same way that a circled implicant on a Karnaugh map is equivalent to the sum of the minterms circled. (Since any sum of minterms can be written as a product of maxterms, what is true for minterms has a dual realization using maxterms.) For example, a set of weights $w = (w_0, w_1, \ldots, w_{N-1})$ with $w_0 = 0$ and $w_i \in \{-1, +1\}$ for $i = 1 \ldots N-1$ acts the same as the following two minterm template expression:

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

The ability to assign threshold values other than $\pm(N-z-1)$ used for the min/maxterms allows even more flexibility.

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

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

where $\oplus$ denotes the XOR operation that returns 1 if $u_i \neq v_i$ and 0 if $u_i = v_i$. Consider a template with weights w and threshold $I_0 = -(N-z-1-2k)$ for $k = 0 \ldots N-z$. For the moment, assume that $z = 0$. Then, as described above, when $k = 0$ the unit will implement the particular minterm m_w. When $k \neq 0$, the unit will return TRUE for the set of inputs $\{u : u \in U, dist(u, w) \leq k\}$. A boolean function of this form will be called a *ballterm* and can be written

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

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

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

3.2 Description of Decomposition Algorithm

Since we know a solution always exists to decompose the desired boolean appropriately, we would like to create an algorithm to automatically find a solution for us. There are many possible algorithms because, in general, more than one solution exists. For example, we could simply generate all the minterms of the given boolean function and OR them together. Such a solution, however, ignores the benefits of our larger class of templates and logical operators.

We take advantage of these additional options by implementing a greedy algorithm. The algorithm generates the set B of all the boolean functions that can be implemented with weights in {-1,0,+1}. Let L designate the set of two-input boolean functions for combining elements in B. Figure 2 gives 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:

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

While this distance 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 functions 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.

3.3 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 $f(i) = 1$. So the XOR function, $f_{XOR} = (0, 1, 1, 0)$, could be depicted as shown in Figure 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.

With nine inputs, there are 118,098 boolean functions in the set B . However, since each 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 bits, requiring a total of less than 4MB of storage.

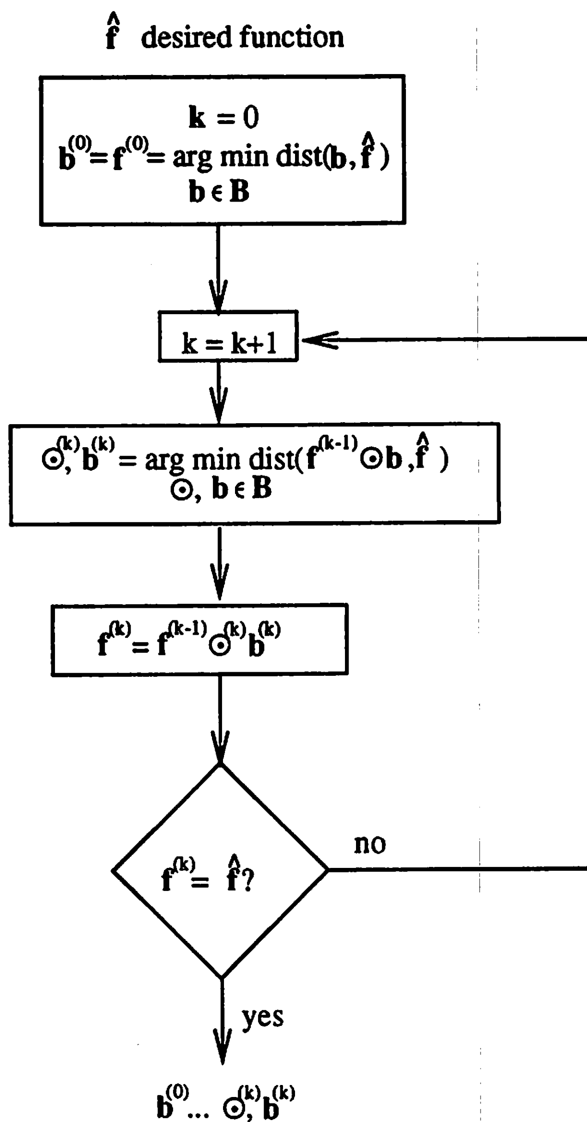

Figure 2: Flowchart for the template-finding algorithm. The input is the desired boolean function. The algorithm finds a sequence of functions from the restricted-weight threshold logic class and associated two-input combining logic to implement the desired function.

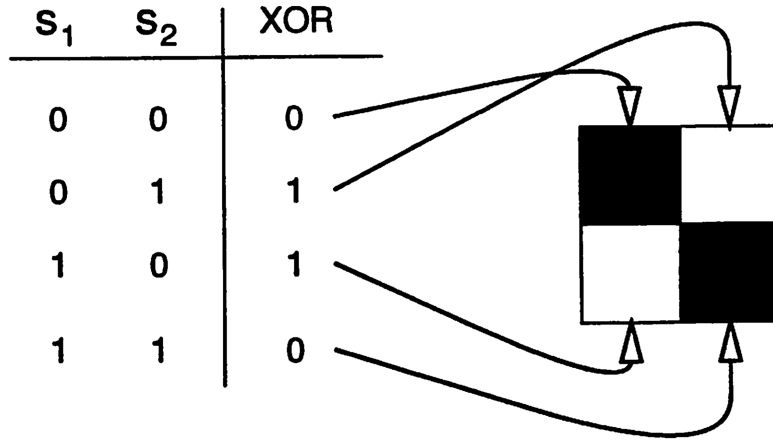

Figure 3: A demonstration of the pictorial boolean function notation used in this paper. The truth table of a boolean function (in this case the exclusive-OR (XOR)) is mapped to a graph by converting successive regions of the truth table to rows of the graph, as shown, giving a compact representation.

Example 1 - Game of Life

Figure 4 shows the algorithm results for implementing the game of life, which is a famous two-dimensional cellular automata rule with many interesting properties [10].

The resulting ballterms and logic operations are:

$b^{(0)} = b_{(-1,-1,-1,-1,0,-1,-1,-1,-1),-1}$

$b^{(1)} = b_{(+1,+1,+1,+1,+1,+1,+1,+1,+1),+4}$

$\odot^{(1)}$ =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:

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

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{B}_1 = \begin{bmatrix} +1 & +1 & +1 \\ +1 & +1 & +1 \\ +1 & +1 & +1 \end{bmatrix} \qquad \mathbf{I}_1 = +4$$

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 [11].

Example 2

Figure 5 shows the algorithm's results for another sample function.

The resulting ballterms and logic operations are:

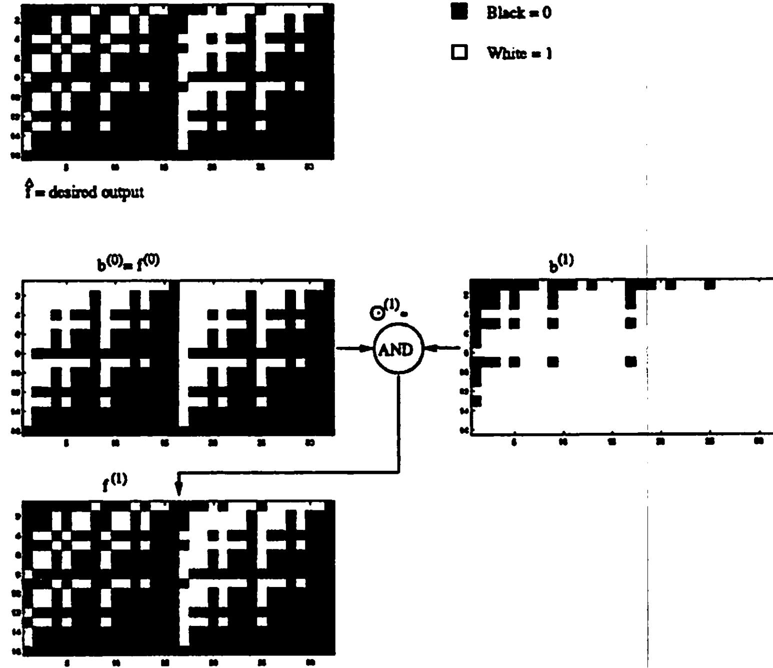

Figure 4: Template-finding algorithm output for the game of life CA. The 512-entry truth tables for the desired and output boolean function are shown pictorially.

$$b^{(0)} = b_{(-1,-1,0,+1,0,-1,-1,0,-1),-3}$$
$$b^{(1)} = b_{(-1,-1,+1,-1,0,+1,-1,+1,-1),-5}$$
$$b^{(2)} = b_{(-1,+1,0,0,+1,0,+1,0,-1),-2}$$
$$b^{(3)} = b_{(-1,0,-1,-1,-1,0,+1,-1,0),-5}$$

$\odot^{(1)}$ =XOR
$\odot^{(2)}$ =AND
$\odot^{(3)}$ =OR

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

4 Performance Considerations

The above algorithm does not guarantee a solution with the minimal number of templates possible. For instance, the case often occurs 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 encounters. This choice of function, however, can make a difference in the final number of templates needed.

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

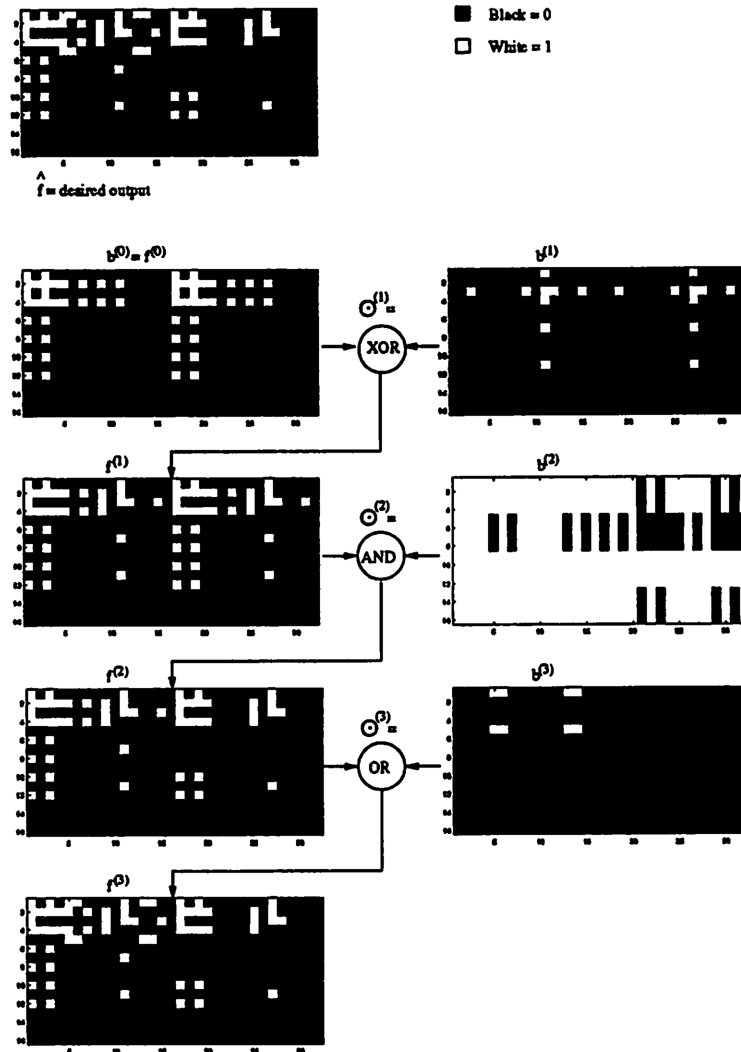

Figure 5: Template-finding algorithm output for a sample input function. The desired function was found to be implementable in four restricted-weight threshold logic stages. The 512-entry truth tables for the desired logic function, the restricted-weight threshold logic functions, and the intermediate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

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 certain situations, such favoritism would be desirable. For example, for $N = 9$ consider the boolean function that outputs 1 every time exactly 4 input variables are 1. Using this new distance measure, the algorithm finds a solution using only two templates while the original distance measure results in a solution with over 70 templates.

In the same manner, we could also define a distance measure that favors OR. Actually, if we restricted our choice of logical operations to include only OR or AND, then we could implement a possibly more time-efficient algorithm that solves for a minimal set of prime implicants plus has some additional procedures to take the ballterm templates into account.

Even if an optimal algorithm could be designed to choose template sequences from this class along with the combining two-input logic, we do not know how many steps might be needed in the worst-case to implement a desired function. A simple counting argument gives 26 steps as a lower upper-bound on the 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.

4.1 Conclusion

Choosing from among the enormous number of threshold logic functions of 9 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. In fact, even if it was optimal it is not known whether the ballterm class is rich enough to implement any boolean function in a reasonable number of iterations. However, the proposed technique represents a significant step towards the efficient automation of template design for the implementation of cellular automata and general neighborhood logic.

References

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

[2] L. O. Chua and T. Roska, "The CNN paradigm," *IEEE Transactions on Circuits and Systems – I*, vol. 40, pp. 147–156, Mar. 1993.

[3] K. R. Crounse and L. O. Chua, "The CNN Universal Machine is as universal as a Turing machine," Memorandum UCB/ERL M95/29, University of California at Berkeley Electronics Research Laboratory, Mar. 1995. Letter accepted for publication in IEEE Transactions on Circuits and Systems – II.

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

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

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

[7] A. P. Marriott, P. Tsalides, and P. J. Hicks, "VLSI implementation of smart imaging system using two-dimensional cellular automata," *IEE Proceedings G (Circuits, Devices and Systems)*, vol. 138, pp. 582–586, Oct. 1991.

[8] P. M. Lewis II and C. L. Coates, *Threshold Logic.* New York: J. Wiley & Sons, Inc., 1967.

[9] L. O. Chua and B. E. Shi, "Exploiting Cellular Automata in the design of Cellular Neural Networks for binary image processing," Memorandum UCB/ERL M89/130, University of California at Berkeley Electronics Research Laboratory, Nov. 1989.

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

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