1
The Chaos Computing Paradigm

William L. Ditto, Abraham Miliotis, K. Murali, and Sudeshna Sinha

1.1
Brief History of Computers

The timeline of the history of computing machines can probably be traced back to early calculation aids, varying in sophistication from pebbles or notches carved in sticks to the abacus, which was used as early as 500 B.C.! Throughout the centuries computing machines became more powerful, progressing from Napier's Bones and the slide rule, to mechanical adding machines and on to the modern day computer revolution.

The 'first generation' of modern computers, were based on wired circuits containing vacuum valves and used punched cards as the main storage medium. The next major step in the history of computing was the invention of the transistor, which replaced the inefficient valves with a much smaller and more reliable component. Transistorized (still bulky) computers, normally referred to as 'Second Generation', dominated the late 1950s and early 1960s.

The explosion in the use of computers began with 'Third Generation' computers. These relied on the integrated circuit or microchip. Large-scale integration of circuits led to the development of very small processing units. Fourth generation computers were developed, using a microprocessor to locate much of the computer's processing abilities on a single (small) chip, allowing the computers to be smaller and faster than ever before. Although processing power and storage capacities have increased beyond all recognition since the 1970s the underlying technology of LSI (large-scale integration) or VLSI (very-large-scale integration) microchips has remained basically the same, so it is widely regarded that most of today's computers still belong to the fourth generation.

Reviews of Nonlinear Dynamics and Complexity. Edited by Heinz Georg Schuster

ISBN: 978-3-527-40945-7

One common thread in the history of computers, be it the abacus or Charles Babbage's mechanical 'anlytical engine' or modern microprocessors, is this: *computing machines reflect the physics of the time and are driven by progress in the understanding of the physical world*.

1.2 The Conceptualization, Foundations, Design and Implementation of Current Computer Architectures

Computation can be actually defined as finding a solution to a problem from given inputs by means of an algorithm. This is what the theory of computation, a subfield of computer science and mathematics, deals with. For thousands of years computing was done with pen and paper, or chalk and slate, or mentally, sometimes with the aid of tables.

The theory of computation began early in the twentieth century, before modern electronic computers had been invented. One of the far-reaching ideas in the theory is the concept of a Turing machine, which stores characters on an infinitely long tape, with one square at any given time being scanned by a read/write head. Basically, a Turing machine is a device that can read input strings, write output strings and execute a set of stored instructions at a time. The Turing machine demonstrated both the theoretical limits and potential of computing systems and is a cornerstone of modern day digital computers.

The first computers were hardware-programmable. To change the function computed, one had to reconnect the wires or even build a new computer. John von Neumann suggested using Turing's Universal Algorithm. The function computed can then be specified by just giving its description (program) as part of the input rather than by changing the hardware. This was a radical idea which changed the course of computing.

Modern day computers still largely implement binary digital computing which is based on Boolean algebra; the logic of the true and false. Boolean algebra shows how you can calculate anything (within some epistemological limits) with a system of two discrete values. Boolean logic became a fundamental component of modern computer architecture, and is remarkable for its sheer conceptual simplicity. For instance, it can be rigorously shown that any logic gate can be obtained by adequate connection of NOR or NAND gates (i.e. any boolean circuit can be built using NOR/NAND gates alone). This implies that the

capacity for universal computing can simply be demonstrated by the implementation of the fundamental NOR or NAND gates [1].

1.3 Limits of Binary Computers and Alternative Approaches to Computation: What Lies Beyond Moore's Law?

The operation of any computing machine is necessarily a physical process, and this crucially determines the possibilities and limitations of the computing device. For the past 20 years, the throughput of digital computers has increased at an exponential rate. Fuelled by (seemingly endless) improvements in integrated-circuit technology, the exponential growth predicted by Moore's law has held true. But Moore's Law will come to an end as chipmakers will hit a wall when it comes to shrinking the size of transistors, one of the chief methods of making chips that are smaller, more powerful and cheaper than their predecessors.

As conventional chip manufacturing technology runs into physical limits in the density of circuitry and signal speed, which sets limits to binary logic switch scaling, alternatives to semiconductor-based binary digital computers are emerging. Apart from analogue VLSI, these include bio-chips, which are based on materials found in living creatures; optical computers that live on pure light; and quantum computers that depend on the laws of quantum mechanics in order to perform, in theory, tasks that ordinary computers cannot.

Neurobiologically inspired computing, quantum computing and DNA computing differ in many respects, but they are similar in that their aim, unlike conventional digital computers, is to utilize at the basic level some of the computational capabilities inherent in the basic, analogue, laws of physics. Further, understanding of biological systems, has triggered the question: what lessons do the workings of the human mind offer for computationally hard problems? Thus the attempt is to create machines that benefit from the basic laws of physics and which are not just constrained by them.

Here we review another emerging computing paradigm: one which exploits the richness and complexity inherent in nonlinear dynamics. This endeavour also falls into the above class, as it seeks to extend the possibilities of computing machines by utilizing the physics of the device.

1.4 Exploiting Nonlinear Dynamics for Computations

We would now like to paraphrase the classic question 'What limits do the laws of classical physics place on computation' to read 'What opportunities do the laws of physics offer computation'.

It was proposed in 1998 that chaotic systems might be utilized to design computing devices [2]. In the early years the focus was on proof-of-principle schemes that demonstrated the capability of chaotic elements to do universal computing. The distinctive feature of this alternative computing paradigm was that it exploited the sensitivity and pattern formation features of chaotic systems.

In subsequent years there has been much research activity to develop this paradigm [3–17]. It was realized that one of the most promising directions of this computing paradigm was its ability to exploit a single chaotic element to reconfigure into different logic gates through a threshold-based morphing mechanism [3, 4]. In contrast to a conventional field programmable gate array element [18], where reconfiguration is achieved through switching between multiple single-purpose gates, reconfigurable chaotic logic gates (RCLGs) are comprised of chaotic elements that morph (or reconfigure) logic gates through the control of the pattern inherent in their nonlinear element. Two input RCLGs have recently been realized and shown to be capable of reconfiguring between all logic gates in discrete circuits [5–7]. Additionally, such RCLGs have been realized in prototype VLSI circuits (0.13 μm CMOS, 30 MHz clock cycles). Further, reconfigurable chaotic logic gates arrays (RCGA) which morph between higher-order functions such as those found in a typical arithmetic logic unit (ALU), have also been designed [17].

In this review we first recall the theoretical concept underlying the reconfigurable implementation of all fundamental logical operations utilizing nonlinear dynamics [3]. We also describe specific realizations of the theory in chaotic electrical circuits. Then we present recent results of a method for obtaining logic output from a nonlinear system using the time evolution of the state of the system. Finally we discuss a method for storing and processing information by exploiting nonlinear dynamics. We conclude with a brief discussion of some ongoing technological implementations of these ideas.

1.5 General Concept

We outline below a theoretical method for obtaining all basic logic gates with a single nonlinear system. The broad aim here is to use the rich temporal patterns embedded in a nonlinear time series in a controlled manner to obtain a computing device that is flexible and reconfigurable.

Consider a chaotic element (our *chaotic chip* or *chaotic processor*) whose state is represented by a value x. In our scheme all the basic logic gate operations (NAND, NOR, XOR, AND, OR, XNOR and NOT) involve the following steps:

1) Inputs:

   $x \rightarrow x_0 + X_1 + X_2$ for 2-input logic operations, such as the NAND, NOR, XOR, AND, OR and XNOR operations,

   and

   $x \rightarrow x_0 + X$ for 1-input operations, such as the NOT operation.

   Here x_0 is the initial state of the system, and

   $X = 0$ when $I = 0$

   and

   $X = V_{\text{in}}$ when $I = 1$

   where V_{in} is a positive constant.

2) Dynamical update, i.e. $x \rightarrow f(x)$

   where $f(x)$ is a nonlinear function.

3) Threshold mechanism to obtain output Z:

   $Z = 0 \quad \text{if } f(x) \leq E, \quad \text{and}$

   $Z = f(x) - E \quad \text{if } f(x) > E$

   where E is a monitoring threshold.

This is interpreted as logic output 0 if $Z = 0$ and logic ouput 1 if $Z > 0$ (with $Z \sim V_{\text{in}}$).

Since the system is strongly nonlinear, in order to specify the inital x_0 accurately one needs a controlling mechanism. Here we will employ a threshold controller [19,20] to set the inital x_0. Namely, we will use the clipping action of the threshold controller to achieve the initialization and subsequently to obtain the output as well.

Note that in our implementation we demand that the *input and output have equivalent definitions* (i.e. one unit is the same quantity for input and output), as well as among various logical operations. This requires that constant V_{in} assumes the same value throughout a network, and this will allow the output of one gate element to couple easily to another gate element as input, so that gates can be wired directly into gate arrays implementing compounded logic operations.

In order to obtain all the desired input-output responses of the different gates, we need to satisfy the conditions enumerated in Table 1.1 simultaneously. So given a dynamics $f(x)$ corresponding to the physical device in actual implementation, one must find values of the threshold and initial state which satisfy the conditions derived from the Truth Tables to be implemented (see Table 1.2).

Table 1.1 Truth table of the basic logic operations for a pair of inputs: I_1, I_2 [1]. The 1-input NOT gate is given by: NOT(0) is 1; NOT(1) is 0.

I_1	I_2	NAND	NOR	XOR	AND	OR	XNOR
0	0	1	1	0	0	0	1
0	1	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	1	0	0	0	1	1	1

A representative example is given in Table 1.3, which shows the exact solutions of the initial x_0 and threshold E which satisfy the conditions in Table 1.2 when the dynamical evolution is governed by the prototypical logistic equation:

$$f(x) = 4x(1 - x)$$

The constant $V_{\text{in}} = \frac{1}{4}$ is common to both input and output and to all logical gates.

Table 1.2 Necessary and sufficient conditions, derived from the logic truth tables, to be satisfied simultaneously by the nonlinear dynamical element, in order to have the capacity to implement the logical operations AND, OR, XOR, NAND, NOR and NOT (cf. Table 1.1) with the same computing module.

Logic Operation	Input Set (I_1, I_2)	Output	Necessary and Sufficient Condition
AND	(0,0)	0	$f(x_0) < E$
	(0,1)/(1,0)	0	$f(x_0 + V_{\text{in}}) < E$
	(1,1)	1	$f(x_0 + 2V_{\text{in}}) - E = V_{\text{in}}$
OR	(0,0)	0	$f(x_0) < E$
	(0,1)/(1,0)	1	$f(x_0 + V_{\text{in}}) - E = V_{\text{in}}$
	(1,1)	1	$f(x_0 + 2V_{\text{in}}) - E = V_{\text{in}}$
XOR	(0,0)	0	$f(x_0) < E$
	(0,1)/(1,0)	1	$f(x_0 + V_{\text{in}}) - E = V_{\text{in}}$
	(1,1)	0	$f(x_0 + 2V_{\text{in}}) < E$
NOR	(0,0)	1	$f(x_0) - E = V_{\text{in}}$
	(0,1)/(1,0)	0	$f(x_0 + V_{\text{in}}) < E$
	(1,1)	0	$f(x_0 + 2V_{\text{in}}) < E$
NAND	(0,0)	1	$f(x_0) - E = V_{\text{in}}$
	(0,1)/(1,0)	1	$f(x_0 + V_{\text{in}}) - E = V_{\text{in}}$
	(1,1)	0	$f(x_0 + 2V_{\text{in}}) < E$
NOT	0	1	$f(x_0) - E = V_{\text{in}}$
	1	0	$f(x_0 + V_{\text{in}}) < E$

Above, we have explicitly shown how one can select temporal responses, corresponding to different logic gate patterns, from a nonlinear system, and this ability allows us to construct flexible hardware. Contrast our use of nonlinear elements here with the possible use of linear systems on one hand and stochastic systems on the other. It is not possible to extract all the *different* logic responses from the *same* element in the case of linear components, as the temporal patterns are inherently very limited. So linear elements do not offer much flexibility or versatility. Stochastic elements on the other hand have many different temporal sequences. However, they are *not deterministic* and so one cannot use them to *design* components. Only nonlinear dynamics enjoys both richness of temporal behavior as well as determinism.

Table 1.3 One specific set of solutions of the conditions in Table 1.2 which yield the logical operations AND, OR, XOR, NAND and NOT, with $V_{\text{in}} = \frac{1}{4}$. Note that these theoretical solutions have been fully verified in a discrete electrical circuit emulating a logistic map [5].

Operation	AND	OR	XOR	NAND	NOT
x_0	0	1/8	1/4	3/8	1/2
E	3/4	11/16	3/4	11/16	3/4

Also note that, while *nonlinearity* is absolutely necessary for implementing all the logic gates, chaos may not always be necessary. In the representative example of the logistic map presented in Table 1.3, solutions for all the gates exist only at the fully chaotic limit of the logistic map but the degree of nonlinearity necessary for obtaining all the desired logic responses will depend on the system at hand and on the specific scheme employed to obtain the input-output mapping. It may happen that certain nonlinear systems will allow a wide range of logic responses without actually being chaotic.

1.6 Continuous-Time Nonlinear System

We now present a somewhat different method for obtaining logic responses from a continuous-time nonlinear system. Our processor is now a continuous-time system described by the evolution equation $d\,\mathbf{x}/dt = \mathbf{F}(\mathbf{x}, t)$, where $\mathbf{x} = (x_1, x_2, \ldots x_N)$ are the state variables and $\mathbf{F}$ is a nonlinear function. In this system we choose a variable, say x_1, to be thresholded. Whenever the value of this variable exceeds a threshold E it resets to E, i.e. when $x_1 > E$ then (and only then) $x_1 = E$.

Now the basic 2-input 1-output logic operation on a pair of inputs I_1, I_2 in this method simply involves the setting of an inputs-dependent threshold, namely the threshold is:

$$E = V_{\text{C}} + I_1 + I_2$$

where V_C is the dynamic control signal determining the functionality of the processor. By switching the value of V_C one can switch the logic operation being performed.

Again I_1/I_2 has the value 0 when the logic input is 0 and has the value V_{in} when the logic input is 1. So the threshold E is equal to V_C when the logic inputs are $(0,0)$, $V_C + V_{in}$ when the logic inputs are $(0,1)$ or $(1,0)$ and $V_C + 2V_{in}$ when the logic inputs are $(1,1)$.

The output is again interpreted as a logic output 0 if $x_1 < E$, i.e. the excess above threshold $V_0 = 0$. The logic output is 1 if $x_1 > E$, and the excess above threshold $V_0 = (x_1 - E) \sim V_{in}$. The schematic diagram of this method is displayed in Figure 1.1.

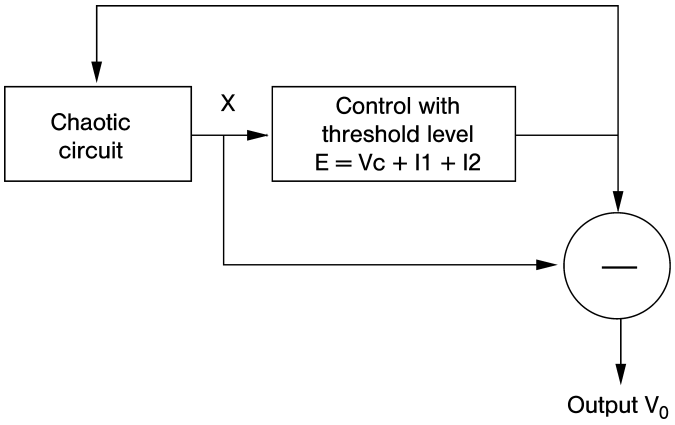

Figure 1.1 Schematic diagram for implementing a morphing 2 input logic cell with a continuous time dynamical system. Here V_C determines the nature of the logic response, and the 2 inputs are $I1, I2$.

Now, for a NOR gate implementation ($V_C = V_{NOR}$) the following must hold true (cf. truth table in Table 1.1):

- when input set is $(0,0)$, output is 1, which implies that for threshold $E = V_{NOR}$, output $V_0 = (x_1 - E) \sim V_{in}$;
- when input set is $(0,1)$ or $(1,0)$, output is 0, which implies that for threshold $E = V_{NOR} + V_{in}$, $x_1 < E$ so that output $V_0 = 0$;
- when input set is $(1,1)$, output is 0, which implies that for threshold $E = V_{NOR} + 2V_{in}$, $x_1 < E$ so that output $V_0 = 0$.

For a NAND gate ($V_C = V_{NAND}$) the following must hold true (cf. truth table in Table 1.1):

- when input set is $(0,0)$, output is 1, which implies that for threshold $E = V_{\text{NAND}}$, output $V_0 = (x_1 - E) \sim V_{\text{in}}$;
- when input set is $(0,1)$ or $(1,0)$, output is 1, which implies that for threshold $E = V_{\text{in}} + V_{\text{NAND}}$, output $V_0 = (x_1 - E) \sim V_{\text{in}}$;
- when input set is $(1,1)$, output is 0, which implies that for threshold $E = V_{\text{NAND}} + 2V_{\text{in}}$, $x_1 < E$ so that output $V_0 = 0$.

In order to design a dynamic NOR/NAND gate one has to find values of V_C that will satisfy all the above input-output associations in a robust and consistent manner.

1.7 Proof-of-Principle Experiments

1.7.1 Discrete-Time Nonlinear System

In this section, we describe an iterated map whose nonlinearity has a simple (i.e. minimal) electronic implementation. We then demonstrate explicitly how all the different fundamental logic gates can be implemented and morphed using this nonlinearity. These gates provide the full set of gates necessary to construct a general-purpose, reconfigurable computing device.

Consider an iterated map governed by the following equation:

$$x_{n+1} = \frac{\alpha x_n}{1 + x_n^{\beta}} \tag{1.1}$$

where α and β are system parameters. Here we will consider $\alpha = 2$ and $\beta = 10$ where the system displays chaos.

In order to realize the chaotic map above in circuitry, one needs two sample-and-hold circuits (S/H): the first S/H circuit holds an input signal (x_n) in response to a clock signal CK1. The output from this sample-and-hold circuit is fed as input to the nonlinear device for subsequent mapping, $f(x_n)$. A second sample-and-hold (S/H) circuit takes the output from the nonlinear device in response to a clock signal CK2. In lieu of control, the output from the second S/H circuit (x_{n+1}) closes the loop as the input to first S/H circuit. The main purpose of the two sample-

and-hold circuits is to introduce discreteness into the system and, additionally, to set the iteration speed.

To implement a control for nonlinear dynamical computing, the output from the second sample-and-hold circuit is input to a threshold controller, described by:

$$\begin{aligned} x_{n+1} &= f(x_n) \quad \text{if} \quad x_{n+1} < E \\ x_{n+1} &= x^* \quad\quad \text{if} \quad x_{n+1} \geq E \end{aligned} \tag{1.2}$$

where E is a prescribed threshold. The output from this threshold controller then becomes the input to the first sample-and-hold circuit.

In the circuit, the notations x_n and x_{n+1} denote voltages. A simple nonlinear device is produced by coupling two complementary (n-channel and p-channel) junction field-effect transistors (JFETs) [13] mimicking the nonlinear characteristic curve $f(x) = 2x/(1 + x^{10})$. The circuit diagram is shown in Figure 1.2. The voltage across resistor R1 is amplified by a factor of five using operational amplifier U1 in order to scale the output voltage back into the range of the input voltage, a necessary condition for a circuit based on a map.

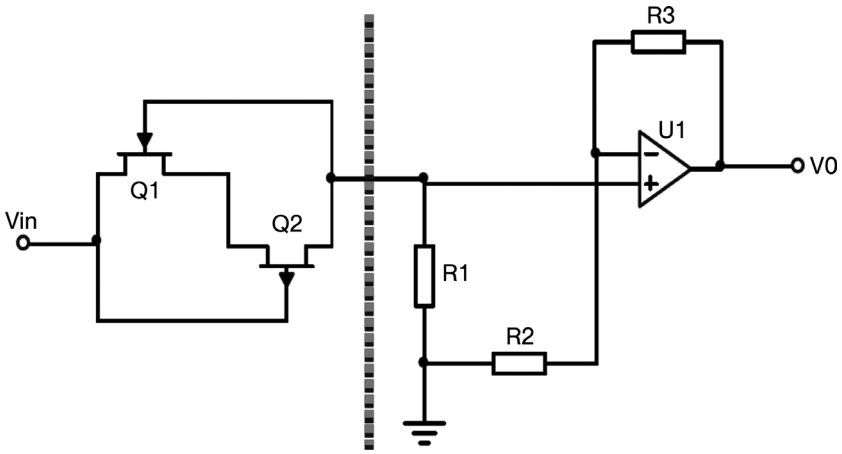

Figure 1.2 Circuit diagram of the nonlinear device. Left: Intrinsic (resistorless), complementary device made of two (n-type and p-type) JFETs. Q1: 2N5457, Q2: 2N5460. Right: Amplifier circuitry to scale the output voltage back into the range of the input voltage. R1: $535\ \Omega$, U1: AD712 op-amp, R2: 100 kΩ and R3: 450 kΩ. Here $V_{\text{in}} = x_n$ and $V_0 = x_{n+1}$.

The resulting transfer characteristics of the nonlinear device are depicted in Figure 1.3 In Figure 1.2, the sample-and-hold circuits are realized with National Semiconductor's sample-and-hold IC LF398, triggered by delayed timing clock pulses CK1 and CK2 [13]. Here a clock rate of either 10 or 20 kHz may be used. The threshold controller circuit as shown in Figure 1.4 is realized with an AD712 operational amplifier, a 1N4148 diode, a 1 kΩ series resistor and the threshold control voltage.

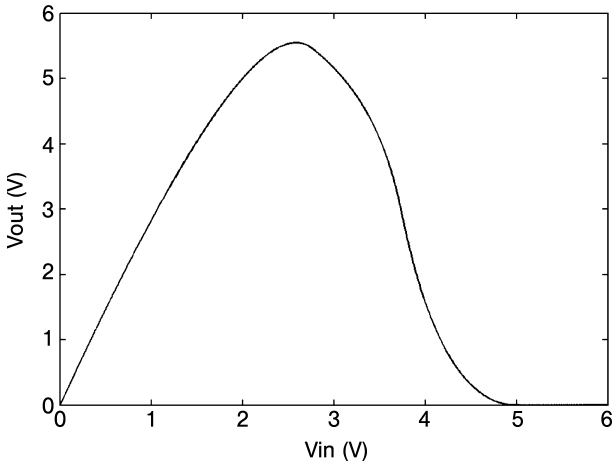

Figure 1.3 Nonlinear device characteristics.

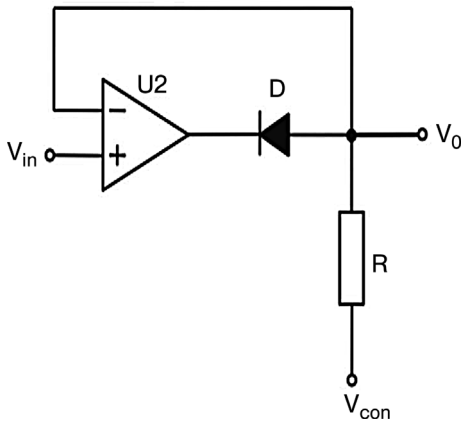

Figure 1.4 Circuit diagram of the threshold controller. V_{in} and V_0 are the input and output, D is a 1N4148 diode, R = 1 kΩ, and U2 is an AD712 op-amp. The threshold level E is given by the controller input voltage V_{con}.

Now in order to implement all the fundamental logic operations, NOR, NAND, AND, OR and XOR with this nonlinear system we have to find a range of parameters for which the necessary and sufficient conditions displayed in Table 1.2 are satisfied. These inequalities have many possible solutions depending on the size of V_{in}. By setting $V_{\text{in}} = 0.3$ we can easily solve the equations for the different x_0 that each gate requires. The specific x_0 values for different logical operations are listed in Table 1.4.

Table 1.4 One specific solution of the conditions in Table 1.2 which yields the logical operations AND, OR, XOR, NAND and NOT, with $V_{\text{in}} = 0.3$ and threshold V_{con} equal to 1 (cf. Figure 1.4). These values are in complete agreement with hardware circuit experiments.

Operation	NOR	NAND	AND	OR	XOR
x_0	0.9138	0.6602	0.0602	0.3602	0.45

Thus we have presented a proof-of-principle device that demonstrates the capability of this nonlinear map to implement all the fundamental computing operations. It does this by exploiting the nonlinear responses of the system. The main benefit is its ability to exploit a single chaotic element to reconfigure into different logic gates through a threshold-based morphing mechanism. Contrast this to a conventional field programmable gate array element, where reconfiguration is achieved through switching between multiple single-purpose gates. This latter type of reconfiguration is both slow and wasteful of space on an integrated circuit.

1.7.2 Continuous-Time Nonlinear System

A proof-of-principle experiment of the method using the continuous time chaotic systems described in Section 1.6 was realized with the dou-

ble scroll chaotic Chua's circuit given by the following set of (rescaled) three coupled ODEs [21]

$$\dot{x_1} = \alpha(x_2 - x_1 - g(x_1)) \tag{1.3}$$

$$\dot{x_2} = x_1 - x_2 + x_3 \tag{1.4}$$

$$\dot{x_3} = -\beta x_2 \tag{1.5}$$

where $\alpha = 10$ and $\beta = 14.87$ and the piecewise linear function $g(x) = bx + \frac{1}{2}(a-b)(|x+1| - |x-1|)$ with $a = -1.27$ and $b = -0.68$. We used the ring structure configuration of the classic Chua's circuit [21].

In the experiment we implemented minimal thresholding on variable x_1 (this is the part in the 'control' box in the schematic figure). We clipped x_1 to E, if it exceeded E, only in (1.4). This has very easy implementation, as it avoids modifying the value of x_1 in the nonlinear element $g(x_1)$, which is harder to do. So then all we need to do is to implement $\dot{x_2} = E - x_2 + x_3$ instead of (1.4), when $x_1 > E$, and there is no controlling action if $x_1 \leq E$.

A representative example of a dynamic NOR/NAND gate can be obtained in this circuit implementation with parameters: $V_{\text{in}} = 2$ V. The NOR gate is realized around $V_{\text{C}} = 0$ V (see Figure 1.6). At this value of control signal, we have the following: for input (0,0) the threshold level is at 0, which yields $V_0 \sim 2$ V; for inputs (1,0) or (0,1) the threshold level is at 0, which yields $V_0 \sim 0$ V; and for input (1,1) the threshold level is at 2 V, which yields $V_0 = 0$ as the threshold is beyond the bounds of the chaotic attractor.

The NAND gate is realized around $V_{\text{C}} = -2$ V. This control signal yields the following: for input (0,0) the threshold level is at -2 V, which yields $V_0 \sim 2$ V; for inputs (1,0) or (0,1) the threshold level is at 2 V, which yields $V_0 \sim 2$ V; and for input (1,1) the threshold level is at 4 V, which yields $V_0 = 0$ [6].

In the example above, the knowledge of the dynamics allowed us to design a control signal that can select out the temporal patterns emulating the NOR and NAND gate [7]. So as the dynamic control signal V_{C} switches between 0 V and -2 V, the module first yields the NOR and then a NAND logic response. Thus one can obtain a dynamic logic gate capable of switching between two fundamental logic responses, namely the NOR and NAND.

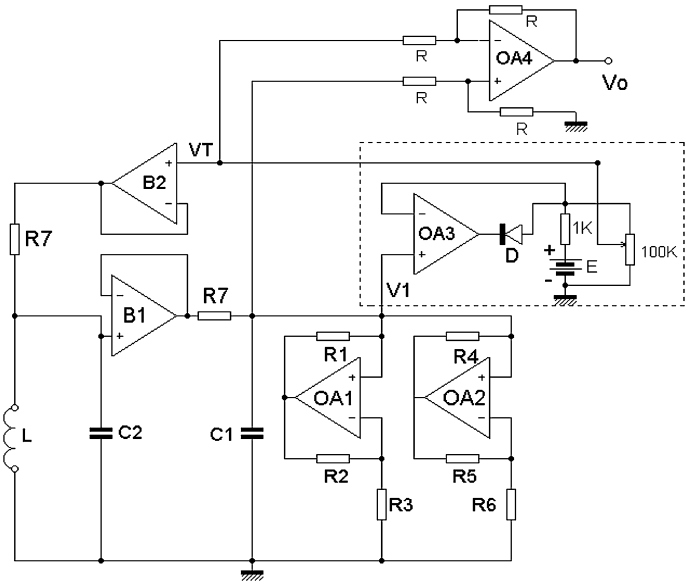

Figure 1.5 Circuit diagram with the threshold control unit in the dotted box.

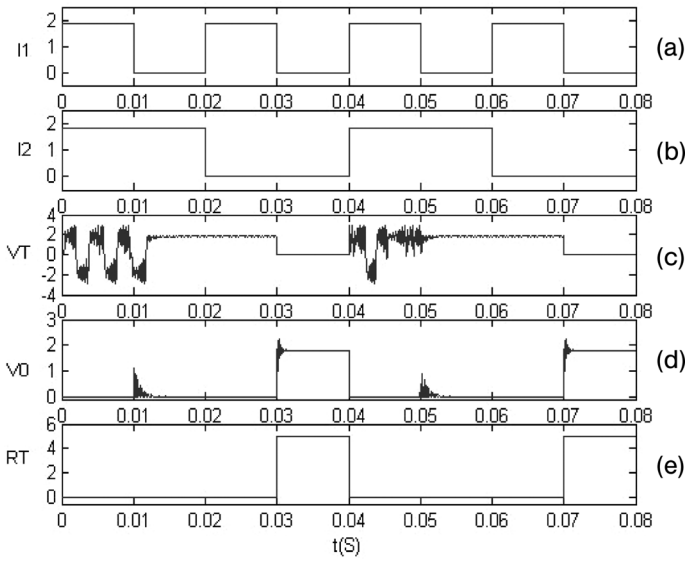

Figure 1.6 Timing sequences from top to bottom: (a) First input I1, (b) Second input I2, (c) Output VT (cf. Figure 1.5), (d) Output V_0 (cf. Figure 1.5) and (e) Recovered Output (RT) obtained by thresholding, corresponding to NOR (I_1, I_2).

1.8 Logic from Nonlinear Evolution: Dynamical Logic Outputs

Now we describe a method for obtaining logic output from a nonlinear system using the time evolution of the state of the system. Namely, our concept uses the nonlinear characteristics of the time dependence of the state of the dynamical system to extract different responses from the system. The highlight of this method is that a single system can yield complicated logic operations, very efficiently.

As before, we have:

1) Inputs:

   $x \rightarrow x_0 + X_1 + X_2$ for 2-input logic operations, such as NOR, NAND, AND, OR, XOR and XNOR operations, and

   $x \rightarrow x_0 + X$ for 1-input logic operations such as NOT operation

   Here x_0 is the initial state of the system, and

   $X = 0$ when $I = 0$, and

   $X = V_{\text{in}}$ when $I = 1$ (where V_{in} is a positive constant)

2) Nonlinear evolution over n time steps, i.e. $x \rightarrow f_n(x)$
   where $f(x)$ is a nonlinear function.

3) Threshold mechanism to obtain the Output:

   If $f_n(x) \leq E$ Logic Output is 0, and

   If $f_n(x) > E$ Logic Output is 1

   where E is the threshold.

So the inputs set up the initial state: $x_0 + I1 + I2$. Then the system evolves over n iterative time steps to updated state x_n. The evolved state is compared to a monitoring threshold E. If the state is greater

Table 1.5 Necessary and sufficient conditions to be satisfied by a chaotic element in order to implement the logical operations NAND, AND, NOR, XOR and OR during different iterations.

LOGIC	NAND	AND	NOR	XOR	OR
Iteration n	1	2	3	4	5
Input $(0,0)$	$x_1 = f(x_0) > E$	$f(x_1) < E$	$f(x_2) > E$	$f(x_3) < E$	$f(x_4) < E$
Input $(0,1)/(1,0)$	$x_1 = f(x_0 + V_{\text{in}}) > E$	$f(x_1) < E$	$f(x_2) < E$	$f(x_3) > E$	$f(x_4) > E$
Input $(1,1)$	$x_1 = f(x_0 + 2V_{\text{in}}) < E$	$f(x_1) > E$	$f(x_2) < E$	$f(x_3) < E$	$f(x_4) > E$

Table 1.6 Updated state of chaotic element satisfying the conditions in Table 1.5 in order to implement the logical operations NAND, AND, NOR, XOR and OR during different iterations with $x_0 = 0.325, V_{\text{in}} = \frac{1}{4}$ and $E = 0.6$.

Operation	NAND	AND	NOR	XOR	OR
Iteration n	1	2	3	4	5
State of the system (x_n)	x_1	x_2	x_3	x_4	x_5
Logic input(0,0) x_0 =0.325	0.88	0.43	0.98	0.08	0.28
Logic input(0,1)/(1,0) x_0 = 0.575	0.9775	0.088	0.33	0.872	0.45
Logic input(1,1) x_0 =0.825	0.58	0.98	0.1	0.34	0.9

than the threshold, a logical 1 is the output, and if the state is less than the threshold, a logical 0 is the output. This process is repeated for subsequent iterations. (See Figure 1.7 for a representative example.)

1.8.1 Implementation of Half- and Full-Adder Operations

Now the ubiquitous bit-by-bit arithmetic addition (half-adder) involves two logic gate outputs: namely AND (to obtain carry) and XOR (to obtain first digit of sum). Using the scheme above we can obtain this combinational operation in consecutive iterations, with a *single* one-dimensional chaotic element.

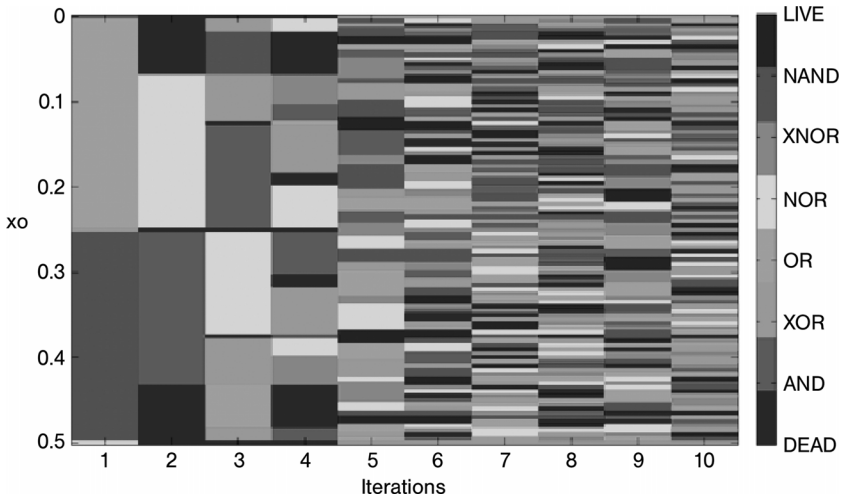

Figure 1.7 Template showing different logic patterns for range of x_0 (0–0.5) versus iteration n (0–10). Here $E = 0.75$ for $1 \leq n \leq 4$ and $E = 0.4$ for $n > 4$. V_{in} is fixed as 0.25.

Further, the typical full-adder requires two half-adder circuits and an extra OR gate. So in total, the implementation of a full-adder requires five different gates (two XOR gates, two AND gates and one OR gate). However, using the dynamical evolution of a *single* logistic map, we require only three iterations to implement the full-adder circuit. So this method allows combinational logic to be obtained very efficiently.

1.9 Exploiting Nonlinear Dynamics to Store and Process Information

Information storage is a fundamental function of computing devices. Computer memory is implemented by computer components that retain data for some interval of time. Storage devices have progressed from punch cards and paper tape to magnetic, semiconductor and optical disc storage by exploiting different natural physical phenomena to achieve information storage. For instance, the most prevalent memory element in electronics and digital circuits is the flip-flop or bistable multivibrator which is a pulsed digital circuit capable of serving as a one-bit memory, namely storing value 0 or 1. More meaningful information is obtained by combining consecutive bits into larger units.

Now we consider a different direction in designing information storage devices. Namely, we will implement data storage schemes based on the wide variety of controlled patterns that can be extracted from nonlinear dynamical systems. Specifically we will demonstrate the use of arrays of nonlinear elements to stably encode and store various items of information (such as patterns and strings) to create a database. Further, we will demonstrate how this storage method also allows one to efficiently determine the number of matches (if any) to specified items of information in the database. So the nonlinear dynamics of the array elements will be utilized for flexible-capacity storage, as well as for preprocessing data for exact (and inexact) pattern matching tasks. We give below, the specific details of our method and demonstrate its efficacy with explicit examples.

1.9.1 Encoding Information

We consider encoding N data elements (labeled as $j = 1, 2, \ldots, N$), each comprised of one of M distinct items (labeled as $i = 1, 2, \ldots, M$). N can be arbitrarily large and M is determined by the kind of data being stored. For instance, for storing English text one can consider the letters of the alphabet to be the natural distinct items building the database, namely $M = 26$. Or, for the case of data stored in decimal representation, $M = 10$, and for databases in bioinformatics comprised typically of symbols A, T, C, G, one has $M = 4$. One can also consider strings and patterns as the items. For instance, for English text one can also consider the keywords as the items, and this will necessitate larger M as the set of keywords is large.

Now we demonstrate a method which utilizes nonlinear dynamical systems, in particular chaotic systems, to store and process data through the natural evolution of the dynamics. The abundance of distinct behaviors of a chaotic system gives it the ability to represent a large set of items. We also demonstrate how one can process data stored in such systems by utilizing specific dynamical patterns.

We start with a database of size N which is stored by N chaotic elements, with state $X_n^i[j]$ ($j = 1, 2, \ldots, N$). Each dynamical element stores one element of the database, encoding any one of the M items comprising our data. Now in order to hold information one must confine the dynamical system to a fixed point behavior, i.e. a state that is stable and constant throughout the dynamical evolution of the system. We

employ the threshold mechanism mentioned above to achieve this. It works as follows. Whenever the value of a state variable of the system, $X_n^i[j]$, exceeds a prescribed threshold $T^i[j]$ (i.e. when $X_n^i[j] > T^i[j]$) the variable $X_n^i[j]$ is reset to $T^i[j]$. This simple mechanism is capable of extracting a wide range of stable regular behaviors from chaotic systems under different threshold values [19,20].

Typically, a large window of threshold values can be found where the system is confined on fixed points, namely, the state of the chaotic element under thresholding is stable at $T^i[j]$ (i.e. $X_n^i[j] = T^i[j]$ for all times n). So each element is capable of yielding a continuous range of fixed points [19]. As a result it is possible to have a large set of thresholds $T^1, T^2, \ldots, T^M$, each having a one-to-one correspondence with a distinct item of our data. So the number of distinct items that can be stored in a single dynamical element is typically large, with the size of M limited only by the precision of the threshold setting.

In particular, consider a collection of storage elements that evolve in discrete time n according to the tent map,

$$f(X_n^i[j]) = 2\min(X_n^i[j], 1 - X_n^i[j]) \tag{1.6}$$

with each element storing one element of the given database ($j = 1, \ldots N$). Each element can hold any one of the M distinct items indicated by the index i. As described above, a threshold will be applied to each dynamical element to confine it to the fixed point corresponding to the item to be stored. For this map, thresholds ranging from 0 to $2/3$ yield fixed points, namely $X_n^i[j] = T^i[j]$, for all time, when threshold $0 < T^i[j] < 2/3$. This can be obtained exactly from the fact that $f(T^i[j]) > T^i[j]$ for all $T^i[j]$ in the interval $(0, 2/3)$, implying that the subsequent iteration of a state at $T^i[j]$ will always exceed $T^i[j]$, and thus get reset to $T^i[j]$. So $X_n^i[j]$ will always be held at value $T^i[j]$.

In our encoding, the thresholds are chosen from the interval $(0, 1/2)$, namely a subset of the fixed point window $(0, 2/3)$. For specific illustration, with no loss of generality, consider each item to be represented by an integer i, in the range $[1, M]$. Defining a resolution r between each integer as:

$$r = \frac{1}{2}\frac{1}{M+1} \tag{1.7}$$

gives a lookup map from the encoded number to the threshold, namely relating the integers i in the range $[1, M]$, to threshold $T^i[j]$ in the range $[r, \frac{1}{2} - r]$, by:

$$T^i[j] = i.r \tag{1.8}$$

Therefore, we obtain a direct correspondence between a set of integers ranging from 1 to M, where each integer represents an item and a set of M threshold values. So we can store N database elements by setting appropriate thresholds (via (1.8)) on N dynamical elements.

Clearly, from (1.7), if the threshold setting has more resolution, namely smaller r, then a larger range of values can be encoded. Note, however, that precision is not a restrictive issue here, as different representations of data can always be chosen in order to suit a given precision of the threshold mechanism.

1.9.2 Processing Information

Once we have a given database stored by setting appropriate thresholds on N dynamical elements, we can query for the existence of a specific item in the database using one global operational step. This is achieved by *globally* shifting the state of all elements of the database up by the amount that represents the item searched for. Specifically the state $X_n^i[j]$ of all the elements ($j = 1, \ldots N$) is raised to $X_n^i[j] + Q^k$, where Q^k is a search key given by:

$$Q^k = \frac{1}{2} \; - \; T^k \tag{1.9}$$

where k is the number being queried for. So the value of the search key is simply $\frac{1}{2}$ minus the threshold value corresponding to the item being searched for.

This addition shifts the interval that the database elements can span, from $[r, \frac{1}{2} - r]$ to $[r + Q^k, \frac{1}{2} - r + Q^k]$, where Q^k is the globally applied shift. See Figure 1.8, for a schematic of this process.

Notice that the information item being searched for, is coded in a manner 'complimentary' to the encoding of the items in the database (much like a key that fits a particular lock), namely $Q^k + T^k$ adds up to $\frac{1}{2}$. This guarantees that *only* the element matching the item being queried for will have its state shifted to $\frac{1}{2}$. The value of $\frac{1}{2}$ is special

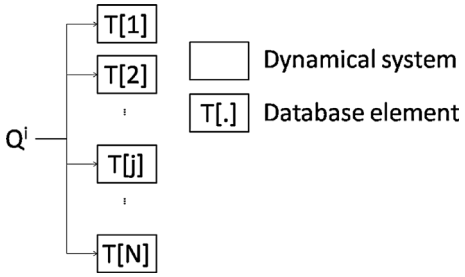

Figure 1.8 Schematic of the database held in an array of dynamical systems and of the parallelized query operation.

in that it is the only state value that, on the subsequent update, will reach the value of 1, which is the maximum state value for this system. So only the elements holding an item matching the queried item will reach extremal value 1 on the dynamical update following a search query. Note that the important feature here is the nonlinear dynamics that maps the state $\frac{1}{2}$ to 1, while all other states (both higher and lower than $\frac{1}{2}$) get mapped to values lower than 1 (see Figure 1.9).

The unique characteristic of the point $\frac{1}{2}$ that makes this work, is the fact that it acts as 'pivot' point for the folding that will occur on the interval $[r + Q^k, \frac{1}{2} - r + Q^k]$ upon the next update. This provides us with a single global monitoring operation to push the state of all the elements matching the queried item to the unique maximal point, in parallel.

The crucial ingredient here is the nonlinear evolution of the state, which results in folding. Chaos is not strictly necessary here. It is evident though that, for unimodal maps, higher nonlinearities allow larger operational ranges for the search operation, and also enhance the resolution in the encoding. For the tent map, specifically, it can be shown that the minimal nonlinearity necessary for the above search operation to work is in the chaotic region. Another specific feature of the tent map is that its piecewise linearity allows the encoding and search key operation to be very simple indeed.

To complete the search we now must detect the maximal state at 1. This can be accomplished in a variety of ways. For example, one can simply employ a level detector to register all elements at the maximal state. This will directly give the total number of matches, if any. So the total search process is rendered simpler as the state with the matching pattern is selected out and mapped to the maximal value, allowing easy

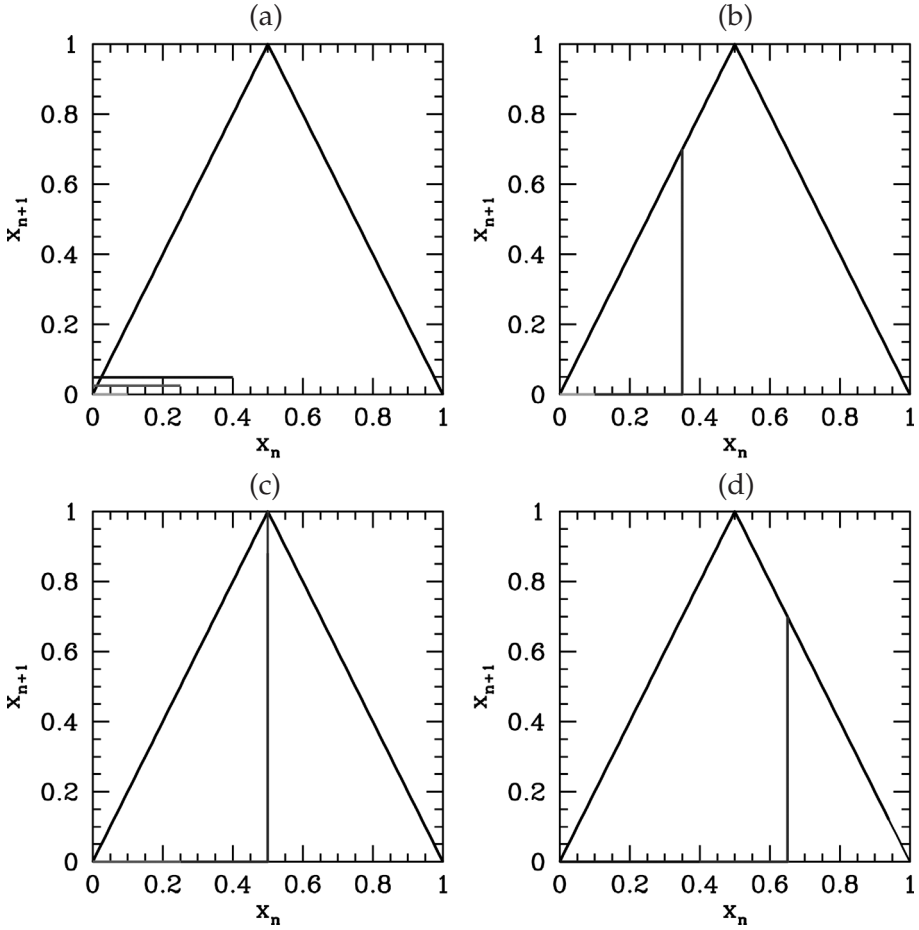

Figure 1.9 Schematic representation of the state of an element (i) matching a queried item (ii) higher than the queried item (iii) lower than the queried item. The top left panel shows the state of the system encoding a list element. Three distinct elements are depicted. The state of the first element is held at 0.1; the second element is held at 0.25 and the third element is held at 0.4. These are shown as lines of proportional lengths on the x-axis in (a). (b)–(d) show each of these elements with the search key added to their states. Here the queried for item is encoded by 0.25. So $Q^k = 1/2 - 0.25 = 0.25$. After the addition of the search key, the subsequent dynamical update yields the maximal state 1 only for the element holding 0.25. The ones with states higher and lower than the matching state (namely 0.1 and 0.4) are mapped to lower values. See also color figure on page 230.

detection. Further, by relaxing the detection level by a prescribed 'tolerance', we can check for the existence within our database of numbers or patterns that are close to the number or pattern being searched for.

So nonlinear dynamics works as a powerful 'preprocessing' tool, reducing the determination of matching patterns to the detection of maximal states, an operation that can be accomplished by simple means, in parallel (see [23]).

1.9.3 Representative Example

Consider the case where our data is English language text, encoded as described above by an array of tent maps. In this case the distinct items are the letters of the English alphabet. As a result $M = 26$ and we obtain $r = 0.018\,518\,5\ldots$ from (1.7), and the appropriate threshold levels for each item is obtained via (1.8). More specifically, consider as our database the line 'the quick brown fox jumps over the lazy dog'; each letter in this sentence is an element of the database, and can be encoded using the appropriate threshold, as in Figure 1.10(a). Now the database, as encoded above, can be queried regarding the existence of specific items in the database. Figure 1.10 presents the example of querying for the letter 'd'. To do so the search key value corresponding to letter 'd' (1.9) is added globally to the state of all elements (b). Then through their natural evolution, upon the next time step, the state of the element(s) containing the letter 'd' is maximized (c). In Figure 1.11 we performed an analogous query for the letter 'o', which is present four times in our database, in order to show that multiple occurrences of the same item can be detected. Finally, in Figure 1.12 we consider a modified database (encoding the line 'a quick brown fox jumped over a lazy dog') and query for an item that is not part of the given database, namely the letter 'h'. As expected, Figure 1.12 (c) shows that none of the elements are maximized.

Further, by relaxing the detection level by a prescribed 'tolerance', we can check for the existence within our list of numbers or patterns that are close to the number or pattern being searched for. For instance, in the example above, by lowering the detection level to the value $1 - (2r)$, we can detect whether adjacent items to the queried one are present. Specifically in the example in Figure 1.12, we can detect that the neighboring letters 'g' and 'i' are contained in our database, though 'h' is not.

However, if we had chosen our representation such that the ordering put T and U before and after Y (as is the case on a standard QWERTY keyboard), then our inexact search would find spellings of *bot* or *bou*

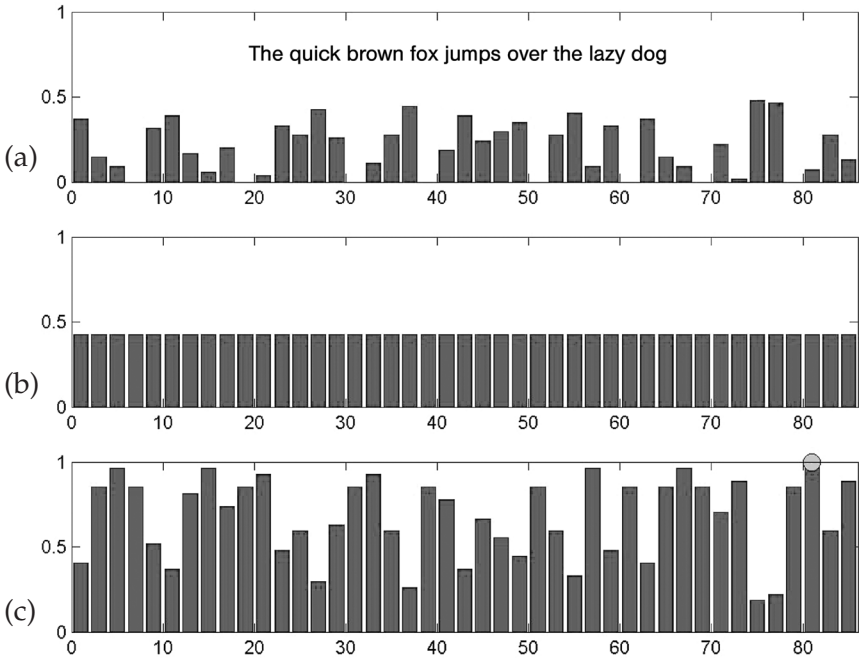

Figure 1.10 From top to bottom: (a) Threshold levels encoding the sentence 'the quick brown fox jumps over the lazy dog', (b) the search key value for letter 'd' is added to all elements, (c) the elements update to the next time step. For clarity we marked with a dot any elements that reach the detection level.

when *boy* was intended. Thus 'nearness' is defined by the choice of the representation and can be chosen advantageously depending on the intended use.

Also note that the system given by (1.1), realizable with the electronic circuit described in Figure 1.2, can also be utilized in a straightforward fashion to implement this storage and information processing method [13].

1.9.4 Implementation of the Search Method with Josephson Junctions

The equations modelling a resistively shunted Josephson junction [22] with current bias and rf drive are as follows:

$$C\frac{dV}{dt} + R^{-1}V + I_c \sin\phi = I_{dc} + I_{rf}\sin(\omega t) \tag{1.10}$$

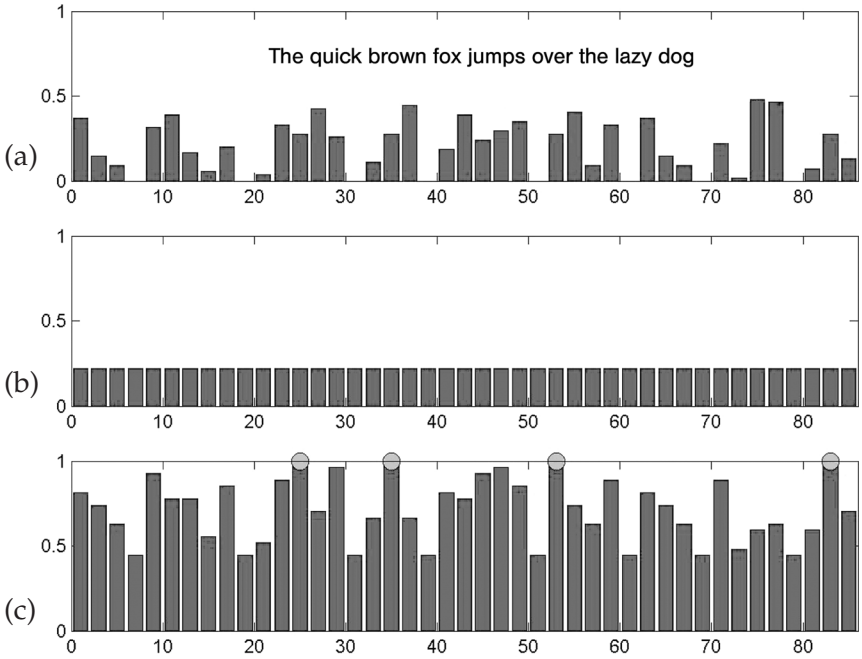

Figure 1.11 From top to bottom: (a) Threshold levels encoding the sentence 'the quick brown fox jumps over the lazy dog', (b) the search key value for letter 'o' is added to all elements, (c) the elements update to the next time step. For clarity we marked with a dot any elements that reach the detection level.

where 2 eV $= \hbar\dot{\phi}$, C is the Josephson junction capacitance, V is the voltage across the junction, R is the shunting resistance, I_c is the critical current of the junction, ϕ is the phase difference across the junction, I_{dc} is the current drive, I_{rf} is the amplitude of the rf-current drive.

If we scale currents to be in units of I_c and time to be in units of ω_p^{-1}, where $\omega_p = (2eI_c/\hbar C)^{1/2}$ is the plasma frequency, we obtain the scaled dynamical equations to be:

$$\frac{dv}{dt} = \beta_c^{-1/2}[i_{dc} + i_{rf}\sin(\Omega t) - v - \sin\phi] \tag{1.11}$$

$$\frac{d\phi}{dt} = \beta_c^{1/2} v$$

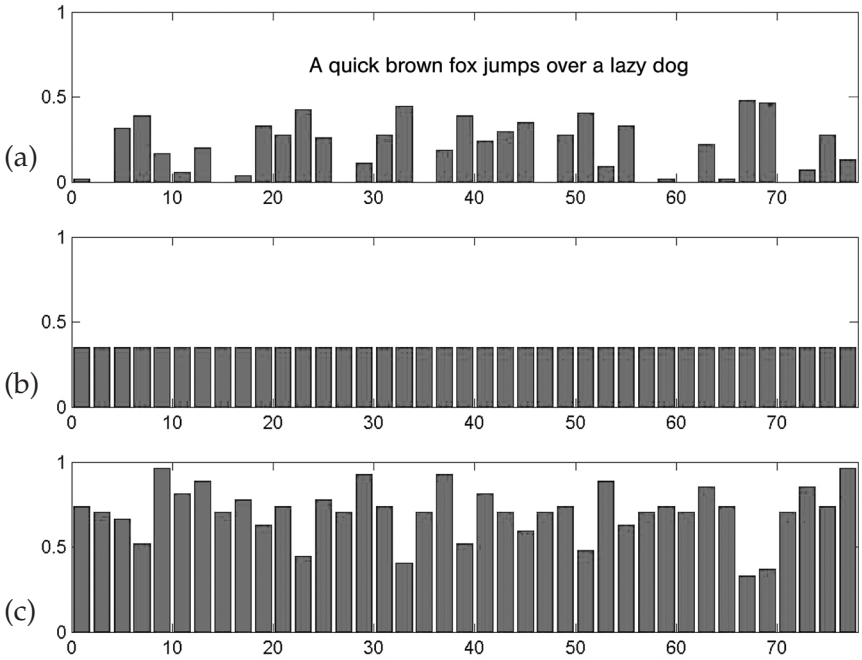

Figure 1.12 From top to bottom: (a) Threshold levels encoding the sentence 'a quick brown fox jumps over a lazy dog'. (b) The search key value for letter 'h' is added to all elements. (c) The elements update to the next time step. No elements reach the detection level as 'h' does not occur in the encoded sentence.

where $\beta_c = 2eI_cR^2C/\hbar$ is the McCumber parameter. Here we choose representative values $\omega_p \sim 36$ GHz, $\Omega = 0.11$, $\beta_c = 4$, $i_{rf} = 1.05$, $i_{dc} = 0.011$.

Using an additional external bias current (added to i_{dc}) to encode an item, one obtains an inverted 'tent-map'-like relation between the absolute value of the Josephson-junction voltage and the biasing input (essentially like a broadened tent map). So, exactly as before, a section of this (broadened) 'map' can be used for encoding and a complementary key can be chosen. The output is a match if it drops below a certain voltage (for instance the one showed by a line in Figure 1.13).

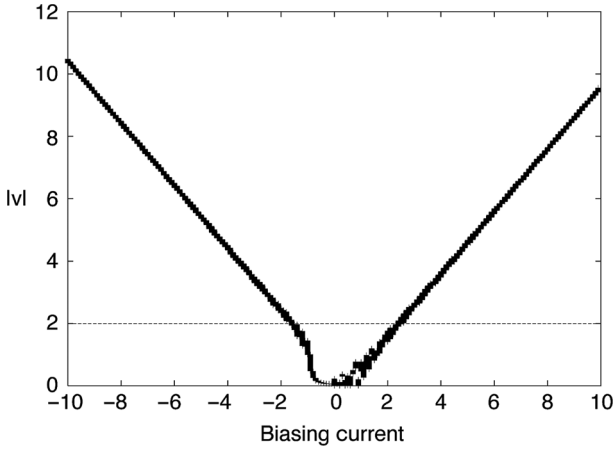

Figure 1.13 Absolute voltage $|v|$ of the Josephson junction vs Input (bias current) given by (2.1).

1.9.5 Discussions

A significant feature of this scheme is that it employs a single simple global shift operation, and does not entail accessing each item separately at any stage. It also uses a nonlinear folding to select out the matched item, and this nonlinear operation is the result of the natural dynamical evolution of the elements. So the search effort is considerably minimized as it utilizes the native processing power of the nonlinear dynamical processors. One can then think of this as a natural application, at the machine level, in a computing machine consisting of chaotic modules. It is also equally potent as a special-applications 'search chip', which can be added on to regular circuitry, and should prove especially useful in machines which are repeatedly employed for selection/search operations.

So in terms of the time scales of the processor the search operation requires one dynamical step, namely one unit of the processor's intrinsic update time. The principal point here is the scope for parallelism that exists in our scheme. This is due to the selection process occurring through one global shift, which implies that there is no scale-up (in principle) with size N. Additionally, this search does not need an ordered set, further reducing operational time.

Regarding information storage capacity, note that we employ an M-state encoding, where M can in principle be very large. This offers much gain in encoding capacity. As in the example we present above,

the letters of the alphabet are encoded by one element each; binary coding would require much more hardware to do the same.

Specifically, consider the illustrative example of encoding a list of names and then searching the list for the existence of a certain name. In the current ASCII encoding technique, each ASCII letter is encoded into two hexadecimal numbers or 8 bits. Assuming a maximum name length of k letters, this implies that one has to use $8 \times k$ binary bits per name. So typically the search operation scales as $O(8kN)$.

Consider, in comparison, what our scheme offers. If base 26 ('alphabetical' representation) is used, each letter is encoded into one dynamical system (an 'alphabit'). As mentioned before, the system is capable of this dense encoding as it can be controlled on to 26 distinct fixed points, each corresponding to a letter. Again assuming a maximum length of k letters per name, one needs to use k 'alphabits' per name. So the search effort scales as kN. Namely, the storage is eight times more efficient and the search can be done roughly eight times faster as well!

If base S encoding is employed, where S is the set of all possible names (size$(S) \leq N$), then each name is encoded into one dynamical system with S fixed points (a 'superbit'). So one needs to use just 1 'superbit' per name, implying that the search effort scales simply as N, i.e. $8\,k$ times faster than the binary encoded case. In practice, the final step of detecting the maximal values can conceivably be performed in parallel. This would reduce the search effort to two time steps (one to map the matching item to the maximal value and another step to detect the maximal value simultaneously). In that case the search effort would be $8\,kN$ times faster than the binary benchmark.

Alternate ideas to implement the increasingly important problem of search have included the use of quantum computers [26]. However, the method here has the distinct advantage that the enabling technology for practical implementation need not be very different from conventional silicon devices. Namely, the physical design of a dynamical search chip should be realizable through conventional CMOS circuitry. Implemented at the machine level, this scheme can perform unsorted database searches efficiently. CMOS circuit realizations of chaotic systems, like the tent map, operate in the region of 1 MHz. Thus a complete search for an item comprising of: search key addition, update, threshold detection and database restoration, should be able to be performed at 250 kHz, regardless of the size of the database. Commercial efforts are underway to construct VLSI circuitry in GHz ranges and are showing promising results in terms of power, size and speed.

Finally, regarding the general outreach of the scheme: nonlinear systems are abundant in nature, and so embodiments of this concept are very conceivable in many different physical systems, ranging from fluids to electronics to optics. Potentially good candidates for physical realization of the method include nonlinear electronic circuits and optical devices, which have distributed degrees of freedom [24]. Also, systems such as single electron tunneling junctions [25], which are naturally piecewise linear maps, can conceivably be employed to make such search devices. All this underscores the general scope of this concept.

1.10 VLSI Implementation of Chaotic Computing Architectures: Proof of Concept

We are currently developing a VLSI implementation of chaotic computing in a demonstration integrated circuit chip. The demonstration chip has a parallel read/write interface to communicate with a microcontroller, with standard logic gates. The read/write interface responds to a range of addresses to give access to internal registers, and the internal registers will interface with the demonstration chaotic computing circuits.

For the demonstration we selected circuits that were based upon known experimental discrete component implementations and, as such, the circuits are larger than is necessary in this first generation of chip. Currently, the TSMC 0.18 μm process is the IC technology chosen for the development. This process was chosen to demonstrate that the chaotic elements work in smaller geometries, and the extra metal layers in this process will provide a margin of safety for any routing issues that might develop.

For our proof of concept on the VLSI chip a small ALU (Arithmetic Logic Unit) with three switchable functions: two arithmetic functions (adder, multiplier, divider, barrel shifter or others) and one function of scratchpad memory, is being implemented. The ALU switches between at least two arithmetic functions and a completely different function like a small FIFO (First-In, First-Out memory buffer). This experiment takes a significant step toward showing the possibilities for future configurable computing. The three functions are combined into a single logic array controlled through a microcontroller interface. The micro-

controller can switch functions and then write data to the interface and read the results back from the interface. Figure 1.14 shows the simplified representation of this experiment [17].

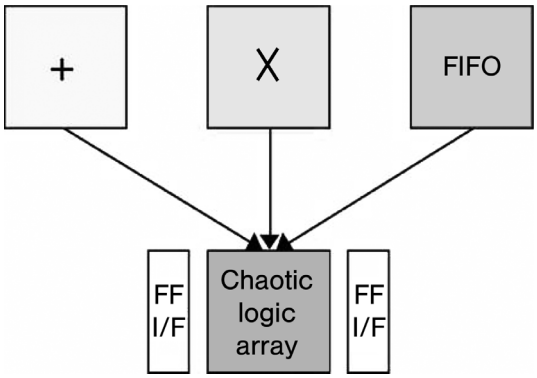

Figure 1.14 Simplified schematic of the proof of concept VLSI implementation of an ALU which can switch between at least two arithmetic functions, and a completely different function such as a small FIFO (First-In, First-Out memory buffer).

Recently, ChaoLogix Inc. designed and fabricated a proof of concept chip that demonstrates the feasibility of constructing reconfigurable chaotic logic gates, henceforth ChaoGates, in standard CMOS-based VLSI (0.18 μm TSMC process operating at 30 MHz with a 3.1×3.1 mm die size and a 1.8 V digital core voltage). The basic building block ChaoGate is shown schematically in Figure 1.14. ChaoGates were then incorporated into a ChaoGate Array in the VLSI chip to demonstrate higher-order morphing functionality including the following:

1. A small Arithmetic Logic Unit (ALU) that morphs between higher-order arithmetic functions (multiplier and adder/accumulator) in *less than one clock cycle*. An ALU is a basic building block of computer architectures.

2. A Communications Protocols (CP) Unit that morphs between two different complex communications protocols *in less than one clock cycle*: Serial Peripheral Interface (SPI, a synchronous serial data link) and an Inter Integrated Circuit Control bus implementation (I2C, a multi-master serial computer bus).

While the design of the ChaoGates and ChaoGate Arrays in this proof of concept VLSI chip was not optimized for performance, it clearly

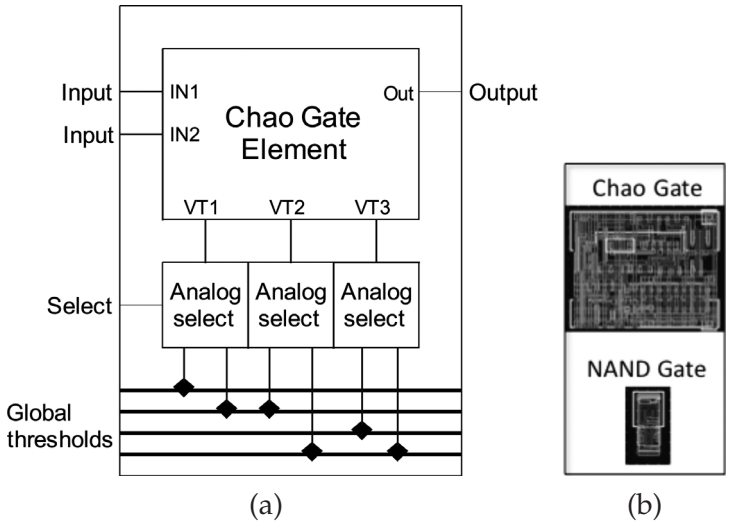

Figure 1.15 (a) Schematic of a two-input, one-output morphable ChaoGate. The gate logic functionality (NOR, NAND, XOR, . . .) is controlled (morphed), in the current VLSI design, by global thresholds connected to VT1, VT2 and VT3 through analog multiplexing circuitry. (b) A size comparison between the current ChaoGate circuitry implemented in the ChaoLogix VLSI chaotic computing chip and a typical NAND gate circuit. (Courtesy of ChaoLogix Inc.)

demonstrates that ChaoGates can be constructed and organized into reconfigurable chaotic logic gate arrays capable of morphing between higher-order computational building blocks. Current efforts are focused upon optimizing the design of a single ChaoGate to levels where they are comparable to or smaller than a single NAND gate in terms of power and size yet are capable of morphing between all gate functions in under a single computer clock cycle. Preliminary designs indicate that this goal is achievable and that all gates currently used to design computers may be replaced with ChaoGates to provide added flexibility and performance.

1.11 Conclusions

In summary, we have demonstrated the direct and flexible implementation of all the basic logic gates utilizing nonlinear dynamics. The richness of the dynamics allows us to select out all the different gate responses from the same processor by simply setting suitable threshold levels. These threshold levels are known exactly from theory and

are thus available as a look-up table. Arrays of such logic gates can conceivably be programmed on the run (for instance, with a stream of threshold values being sent in by an external program) to be optimized for the task at hand. For example, such a morphing device may serve flexibly as an arithmetic processing unit or an unit of memory and can be swapped, as the need demands, to be one or the other. Thus architectures based on such logic implementations may serve as ingredients of a general-purpose reconfigurable computing device more powerful and fault tolerant [11] than statically wired hardware.

Further, we have demonstrated the concept of using nonlinear dynamical elements to store information efficiently and flexibly. We have shown how a single element can store M items, where M can be large and can vary to best suit the nature of the data being stored and the application at hand. So we obtained information storage elements of flexible capacity, which are capable of naturally storing data in different bases or in different alphabets or multilevel logic. This cuts down space requirements significantly, and can find embodiment in many different physical contexts.

Further, we have shown how this method of storing information can be naturally exploited for processing as well. In particular, we have demonstrated a method to determine the existence of an item in the database. The method involves a single global shift operation applied simultaneously to all the elements comprising the database and this operation, after one dynamical step, pushes the element(s) storing the matching item (and only those) to a unique, maximal state. This extremal state can then be detected by a simple level detector, thus directly giving the number of matches. So nonlinear dynamics works as a powerful 'preprocessing' tool, reducing the determination of matching patterns to the detection of maximal states. The method can also be extended to identify inexact matches as well. Since the method involves just one parallel procedural step, it is naturally setup for parallel implementation on existing and future implementations of chaos based computing hardware ranging from conventional CMOS based VLSI circuitry to more esoteric chaotic computing platforms such as magneto based circuitry [27] and high speed chaotic photonic integrated circuits operating in the GHz frequency range [28].

References

1 Mano, M.M. *Computer System Architecture*, 3rd edition, Prentice Hall, Englewood Cliffs, 1993; Bartee, T.C. *Computer Architecture and Logic Design*, New York, Mc-Graw Hill, 1991.

2 Sinha, S. and Ditto, W.L. *Phys. Rev. Lett.* **81** (1998) 2156.

3 Sinha, S., Munakata, T. and Ditto, W.L, *Phys. Rev. E* **65** (2002) 036214; Munakata, T., Sinha, S. and Ditto, W.L, IEEE *Trans. Circ. and Systems* **49** (2002) 1629; Munakata, T. and Sinha, S., Proc. of COOL Chips VI, Yokohama, (2003) 73.

4 Sinha, S. and Ditto, W.L. *Phys. Rev. E* **59** (1999) 363; Sinha, S., Munakata, T. and Ditto, W.L *Phys. Rev. E* **65** 036216; W.L. Ditto, K. Murali and S. Sinha, Proceedings of *IEEE Asia-Pacific Conference on Circuits and Systems* (APCCAS06), Singapore (2006) pp. 1835–38.

5 Murali, K., Sinha, S. and Ditto, W.L., Proceedings of the STATPHYS-22 Satellite conference Perspectives in Nonlinear Dynamics Special Issue of Pramana 64 (2005) 433

6 Murali, K., Sinha, S. and Ditto, W.L., *Int. J. Bif. and Chaos (Letts)* **13** (2003) 2669; Murali, K., Sinha S., and I. Raja Mohamed, I.R., *Phys. Letts. A* **339** (2005) 39.

7 Murali, K., Sinha, S., Ditto, W.L., Proceedings of *Experimental Chaos Conference* (ECC9), Brazil (2006) published in *Phil. Trans. of the Royal Soc. of London* (Series A) (2007); Murali, K., Sinha, S. and Ditto, W.L., Proceedings of *IEEE Asia-Pacific Conference on Circuits and Systems* (APCCAS06), Singapore (2006) pp. 1839–42.

8 Prusha B.S. and Lindner J.F., *Phys. Letts. A* **263** (1999) 105.

9 Cafagna, D. and Grassi, G., *Int. Sym. Signals, Circuits and Systems* (ISSCS 2005) **2** (2005) 749.

10 Chlouverakis, K. E. and Adams, M. J., *Electronics Lett.* **41** (2005) 359.

11 Jahed-Motlagh, M.R., Kia, B., Ditto, W.L. and Sinha, S., *Int. J. of Bif. and Chaos* **17** (2007) 1955.

12 Murali, K. and Sinha, S., *Phys. Rev. E* **75** (2007) 025201

13 Miliotis, A., Murali, K., Sinha, S., Ditto, W.L., and Spano, M.L., *Chaos, Solitons & Fractals*, Volume 42, (2009) Pages 809–819.

14 Miliotis, A., Sinha, S. and Ditto, W.L., *Int. J. of Bif. and Chaos* **18** (2008) 1551–59; Miliotis, A., Sinha, S. and Ditto, W.L., Proceedings of *IEEE Asia-Pacific Conference on Circuits and Systems* (APCCAS06), Singapore (2006) pp. 1843–46.

15 Murali, K., Miliotis, A., Ditto, W.L. and Sinha, S., *Phys. Letts. A* **373** (2009) 1346–51

16 Murali, K., Sinha, S., Ditto, W.L., Bulsara, A.R., *Phys. Rev. Lett.* **102** (2009) 104101; Sinha, S., Cruz, J.M., Buhse, T., Parmananda, P., *Europhys. Lett.* **86** (2009) 60003

17 Ditto, W., Sinha, S. and Murali, K., US Patent Number 07096347 (August 22, 2006).

18 Taubes, G., *Science* **277** (1997) 1935.

19 Biswas, S. and D., *Phys. Rev. Lett.* **71** (1993) 2010; Glass, L. and Zheng, W., *Int. J. Bif. and Chaos* **4** (1994) 1061; Sinha, S., *Phys. Rev. E* **49** (1994) 4832; Sinha, S., *Phys. Letts. A* **199** (1995) 365; Sinha, S. and Ditto, W.L., *Phys. Rev. E* **63** (2001) 056209; Sinha, S., *Phys. Rev. E* **63** (2001) 036212; Sinha, S., *Nonlinear Systems*, Eds. Sahadevan, R. and Lakshmanan, M.L., (Narosa, 2002) 309–28; Ditto, W.L. and Sinha, S., *Phil. Trans. of the Royal Soc. of London* (Series A) **364** (2006) 2483.

20 Murali, K. and Sinha, S., *Phys. Rev. E* **68** (2003) 016210.

21 Maddock, R.J. and Calcutt, D.M., *Electronics: A Course for Engineers*, Addison Wesley Longman Ltd., (1997) p. 542; Dimitriev, A.S. *et al*, J. Comm. Tech. Electronics, **43** (1998) 1038.

22 Cronemeyer, D.C., *et al*, Phys. Rev. B **31** (1985) 2667.

23 For instance, content-addressable memory (CAM) is a special type of computer memory used in certain very-high-speed searching applications, such as routers. Unlike standard computer memory (random access memory or RAM) in which the user supplies a memory address and the RAM returns the data word stored at that address, a CAM is designed such that the user supplies a data word and the CAM searches its entire memory to see if that data word is stored anywhere in it. What we attempt to design here is a CAM-like device.

24 Sukow, D.W., et al. *Chaos* **7** (1997) 560; Blakely, J.N., Illing, L. and Gauthier, D.J., *Phys. Rev. Lett.* **92** (2004); Blakely, J.N., Illing, L. and Gauthier, D.J., *IEEE Journal of Quantum Electronics* **40** (2004) 299.

25 Yang, T. and Chua, L.O., *Int. J. of Bif. and Chaos* **10** 1091 (2000).

26 Grover, LK., *Phys. Rev. Letts.* (1997) **79** 325.

27 Koch, R., Scientific American, 293(2), 56 (2005).

28 Yousefi, M., Barbarin, Y., Beri, S., Bente, E. A. J. M., Smit, M. K., Notzel, R., and Lenstra, D., *Phys. Rev. Lett.* **98**, 044101 (2007)