The Synthesis of Stochastic Circuits for Nanoscale Computation*

Weikang Qian, John Backes and Marc Riedel
Department of Electrical and Computer Engineering
University of Minnesota
200 Union St. S.E., Minneapolis, MN 55455
E-mail: {qianx030, back0145, mriedel}@umn.edu

***Abstract*—Emerging technologies for nanoscale computation such as self-assembled nanowire arrays offer a path to scaling beyond the current limits of CMOS. And yet, such technologies present specific challenges for logic synthesis. On the one hand, nanowire crossbar arrays provide an unprecedented density of bits with a high degree of parallelism. On the other hand, the circuits are characterized by high defect rates as well as inherent randomness in the interconnects due to the stochastic nature of self-assembly. Existing methods for logic synthesis rely on probing the circuit after fabrication and configuring it according to the topology that is discovered.**

We describe a general method for synthesizing logic that exploits both the parallelism and the random effects of the self-assembly, obviating the need for post-fabrication configuration. Our approach is based on stochastic computation with parallel bit streams. Circuits are synthesized through functional decomposition with symbolic data structures called multiplicative binary moment diagrams. Synthesis produces designs with randomized parallel components – *AND* operations and multiplexing – operating on the stochastic bit streams. These components are readily implemented in nanowire crossbar arrays. We present synthesis results for benchmarks circuits illustrating the method. The results show that our technique maps circuit designs onto nanowire arrays effectively, with a measured tradeoff between the degree of redundancy and the accuracy of the computation.

I. Introduction

As the semiconductor industry contemplates the end of Moore's Law, there has been considerable interest in novel materials and devices [4]. Technologies such as molecular switches and carbon nanowire arrays offer a path to scaling beyond the limits of conventional CMOS [5]. Most such technologies are in the exploratory phases, still years or decades from the point when they will be actualized. Accordingly, the development of software tools and techniques for logic synthesis remains speculative.

And yet, for some types of new technologies, we can identify broad traits that will likely impinge upon synthesis. For instance, nanowire arrays are stochastically self-assembled in tightly-pitched bundles. Accordingly, they exhibit the following [3]:

1) A high degree of parallelism.
2) Minimal control during assembly.
3) Inherent randomness in the interconnect schemes.
4) High defect rates.

*This work is supported by a grant from the MARCO Focus Center Research Program on Functional Engineered Nano-Architectonics.

Existing strategies for synthesizing logic for nanowire arrays are based on routing schemes similar to those used for field-programmable gate arrays (FGPAs) [3]. These rely on probing the circuit and programming interconnects after fabrication.

We describe a general method for synthesizing logic that exploits both the parallelism and the random effects of the self-assembly, obviating the need for such post-fabrication configuration. Our approach is based on stochastic computation with parallel bit streams. Circuits are synthesized through functional decomposition with symbolic data structures called multiplicative binary moment diagrams. Synthesis produces designs with randomized parallel components – *AND* operations and multiplexing – operating on the stochastic bit streams. These components are readily implemented in nanowire crossbar arrays. We present synthesis results for benchmarks circuits illustrating the method. The results show that our technique is effective in implementing designs with nanowire arrays, with a measured tradeoff between the degree of redundancy and the accuracy of the computation.

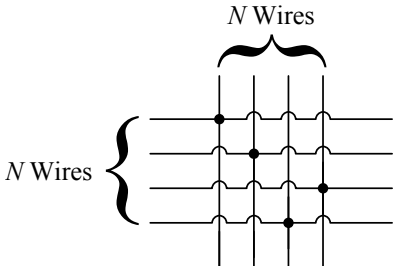

Fig. 1. $N \times N$ nanowire crossbar with random connections.

II. Circuit Model

Our discussion of synthesis is framed in terms of a conceptual model for nanowire arrays. (In Section V, we justify this model with implementation details.) A nanowire crossbar is illustrated in Figure 1. The connections between horizontal and vertical wires are *random*. However, we assume that these connections are nearly one-to-one, that is to say, nearly every

horizontal wire connects to exactly one vertical wire, and vice-versa. This is a specific attribute of types of nanowire arrays, controlled during self-assembly [3].

A. Parallel Stochastic Bit Streams

Our synthesis method implements digital computation in the form of parallel stochastic bit streams. We refer to a collection of parallel nanowires as a **bundle**. The **width** of a bundle is the number wires. Its current **weight** is the number of logical 1's on its wires. The **signal** that it carries is a real value between zero and one corresponding to the **fractional weight**: for a bundle of N wires, if k of the wires are 1, then the signal is k/N. Let $P(X = 1)$ denote the *probability* that any given wire in bundle X carries a 1.

B. Shuffling Devices

We implements computation with two basic nanowire constructs: **shuffled ANDs** and **Bundleplexers**. We describe these only in conceptual terms here; implementation details are postponed until Section V.

1) Shuffled AND: A shuffled AND has two bundles of N wires as inputs and a bundle of N wires as the output. Each wire in the output bundle is actually the output of an AND gate, which takes one input from the first input bundle and the other from the second. The choice of which inputs are fed into which AND gate is random. Figure 2 shows a simple shuffled AND with $N = 3$.

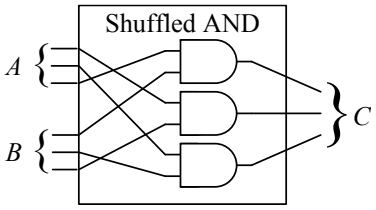

Fig. 2. A shuffled AND element, for bundles of width 3.

Suppose that the signal carried by the first input bundle A is a, that carried by the second input bundle B is b, and that carried by the output bundle C is c. Provided that the bits in the first and second input bundles are independent, for large N we can assume that

$$\begin{aligned} c &= P(C = 1) && (1)\\ &= P(A = 1 \text{ and } B = 1) && (2)\\ &= P(A = 1) \cdot P(B = 1) && (3)\\ &= a \cdot b. && (4) \end{aligned}$$

We see that a shuffled AND in effect performs the *multiplication* of the signals carried by the two input bundles.

2) Bundleplexer: A bundleplexer has two bundles of N wires as its inputs and a bundle of N wires as its output. It is tagged with a fixed **selecting ratio**, $0 < s < 1$. The output bundle is composed of a randomly selected choice of sN bits from the first input bundle and $(1 - s)N$ bits from the second. The choice is not ordered: rather, a random shuffling occurs. Figure 3 shows a bundleplexer with $N = 4$ and $s = 3/4$. The output bundle has three wires from input bundle A and one wire from input bundle B.

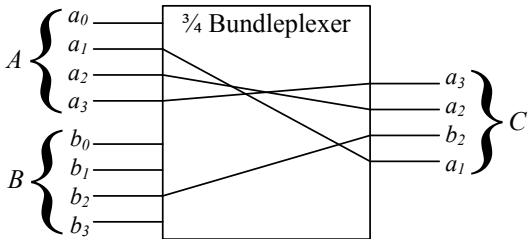

Fig. 3. A bundleplexer with $N = 4$ and $s = 3/4$.

Suppose that the signal carried by the first input bundle A is a, that carried by the second input bundle B is b and that carried by the output bundle C is c. For large N, we can assume that

$$\begin{aligned} c &= P(C = 1) && (5)\\ &= sP(A = 1) + (1 - s)P(B = 1) && (6)\\ &= sa + (1 - s)b. && (7) \end{aligned}$$

We see that a bundleplexer in effect performs a *scaled addition* on the signals carried by the two input bundles.

C. Stochastic Circuits

Our synthesis method produces a circuit design that operates on the fractional-weighted values carried by bundles of wires. Our approach is analogous to the formulation of a real-valued polynomial representation of a circuit, with arithmetic multiplication and addition. (In fact, we perform synthesis with symbolic data structures called binary moment diagrams.)

For example, consider a circuit with the Boolean truth table shown in the top-right in Figure 4. Its output y can be represented as

$$y = a + b - 2ab.$$

Evaluating this polynomial for all Boolean values of a and b gives the correct Boolean output y. We use shuffled ANDs for multiplication and bundleplexing for addition.

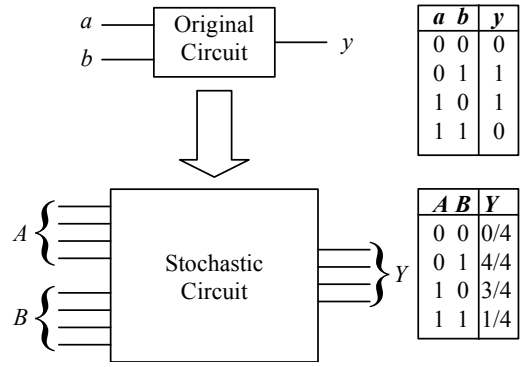

Fig. 4. An example of the formulation of a stochastic circuit.

For a circuit with m inputs and n outputs, we have m input bundles and n output bundles (each bundle consisting of N

parallel wires). For computation, all the wires in each input bundle are set to the corresponding Boolean input value (so all the wires in each bundle are set to 0 or to 1). With bundleplexing, wires are randomly selected from separate bundles. As a result, the internal bundles carry *stochastic* bit streams with fractional weightings.

We assume that the output of the circuit is directly usable in a fractional-weighted form. For instance, in sensor applications, an analog voltage discriminating circuit might be used to transform an output bundle of bits into a Boolean value. We assume direct quantization: an output signal greater than or equal to 0.5 corresponds to logical 1; less than this corresponds to 0.

Figure 4 illustrates the formulation. Bundles of width $N = 4$ are used. The truth table shown in the bottom-right gives the fractional weight on the output bundle Y. For inputs $A = 1$ and $B = 0$, we have $Y = 3/4$, which corresponds to logical 1. For $A = 1$ and $B = 1$, we have $Y = 1/4$, which corresponds to logical 0. Thus, the stochastic circuit implements the same Boolean function as that shown in the top-right truth table.

III. Synthesis of Stochastic Circuits

Our synthesis procedure begins with the specification of a combinational circuit, say in the form of a netlist, and produces a stochastic design consisting of shuffled AND elements and bundleplexer elements. Synthesis is performed through functional decomposition with a variant of binary decision diagrams called **multiplicative binary moment diagrams** (*BMDs) [2].

A. Multiplicative Binary Moment Diagrams

Like binary decision diagrams, *BMDs are a graphic representation of functions over Boolean variables; however, they can have non-Boolean ranges. Figure 6 shows an example of a *BMD for the circuit in Figure 5, implementing the function:

$$y = ((x_1 \wedge x_2) \wedge (x_3 \vee x_4)) \vee (x_3 \wedge x_4).$$

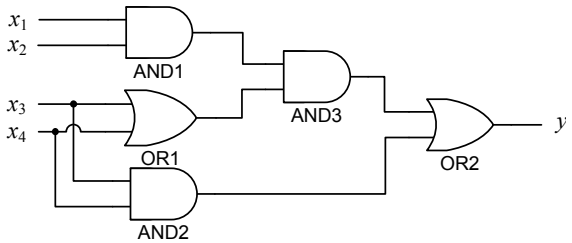

Fig. 5. A simple circuit.

There is a total ordering of the variables in a *BMD. Also, each non-terminal vertex has outgoing edges to two children. Three salient features of *BMDs are:

1) There can be more than two terminal vertices and each terminal vertex can have a number other than 0 or 1. (Terminal vertices are shown as square boxes at the leaves of the tree.)
2) Each edge has an associated weight which can either be an integer or a real value. (The weights are shown in square boxes written directly on the edges; an edge without a box is assumed to have weight 1; the weights of edges that connect to terminal vertices are simply the weights of the terminal vertices themselves.) Note that the edge pointing to the root can also have a weight. (For example, see Figure 7(b).) The function represented by a *BMD is the product of the root weight and the function at the root vertex.
3) The left edge from each vertex indicates the case where the function is *independent* of the vertex variable; the right edge indicates the case where the function depends *linearly* on that variable. (In diagrams, we sometimes exchange "left" and "right" to get prettier graphs; when we do so, we indicate this by annotating the edges with L and R.) Thus, the function f at a vertex with variable x is

$$f = w_L f_L + w_R f_R x, \tag{8}$$

where w_L is the weight of the left edge and w_R is the weight of the right edge; f_L is the function at the vertex pointed to by the left edge and f_R is the function at the vertex pointed to by the right edge. We obtain the function for the *BMD through such recursive decomposition.

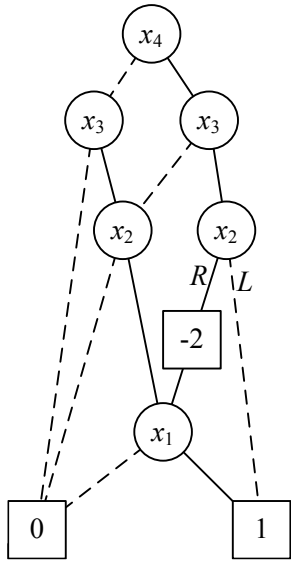

Fig. 6. The *BMD for the circuit in Figure 5.

In order to construct a *BMD, we begin with base functions corresponding to constants and individual variables, and then we build more complex functions by combining these. Given a circuit, we obtain a *BMD for its Boolean function by expressing it in terms of addition and multiplication operations. For the *BMD in Figure 6, the function is

$$f = x_1x_2x_3 + x_1x_2x_4 + x_3x_4 - 2x_1x_2x_3x_4. \tag{9}$$

In general, for a circuit with multiple outputs, there are separate *BMDs for each output. However, in the data structure, significant portions of different *BMDs can often be shared [2].

B. Decomposing a *BMD into Positive and Negative *BMDs

After obtaining a *BMD for a circuit, the next step in our procedure is to decompose it into two *BMDs, both with non-negative-weighted edges, such that the function of the original *BMD equals that of the first *BMD minus that of the second. We call the first *BMD the **positive *BMD** and the second the **negative *BMD**.

Figure 7 shows the positive and negative *BMDs for the *BMD in Figure 6. The function of the positive *BMD is $x_1x_2x_3+x_1x_2x_4+x_3x_4$ and the function of the negative *BMD is $2x_1x_2x_3x_4$. The procedure for this decomposition is given as pseudo-code in Figure 8.

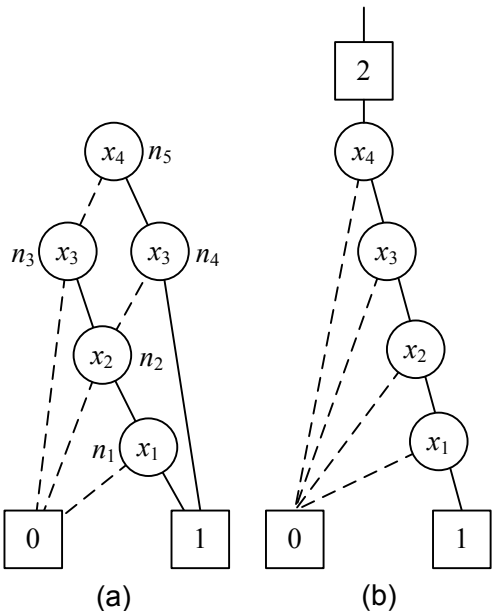

Fig. 7. The positive and negative *BMDs for the *BMD in Figure 6. (a) Positive *BMD; (b) Negative *BMD.

In the pseudo-code, we represent a *BMD as a **weighted pair** of the form (w, v), where v designates the root vertex and w is the root edge weight. (This pair also refers to the function represented by the *BMD.) A vertex $v = \Lambda$ denotes a terminal leaf. The function *Var*(v) returns the variable of vertex v. The function *Left*(v) returns the **left pair** of v: (w_L, v_L), where w_L is the weight of the left edge of v and v_L is the left child of v. Similarly, the function *Right*(v) returns the **right pair** of v: (w_R, v_R), where w_R is the weight of the right edge of v and v_R is the right child of v.

The function *MakeBranch*(x, (w_L, v_L), (w_R, v_R)) constructs a new *BMD. It returns a pair (w, v) designating a *BMD, such that $wf(v) = w_Lf(v_L)+w_Rf(v_R)x$. Here, $f(v)$ denotes the function of vertex v. The function *ApplyWeight*(w', (w, v)) multiplies the function of the pair (w, v) by a constant w' and returns the resulting pair. The functions *MakeBranch* and *ApplyWeight* are described in [2].

The function *PosNegBMD* in Figure 8 takes a pair (w, v) representing a *BMD as an input argument and returns two pairs (w_P, v_P) and (w_N, v_N) representing the positive and negative *BMDs, respectively. It first obtains the positive and negative *BMDs without considering the weight w. In the non-trivial case, i.e., when v is not a terminal vertex, it recursively calls *PosNegBMD* to obtain the positive and negative *BMDs of the *BMDs designated by the left and right pairs of v. Then, the procedure calls the function *MakeBranch* to construct a positive *BMD based on the two positive *BMDs of the left and right pairs of v. Similarly, it constructs a negative *BMD based on the two negative *BMDs of the left and right pairs of v. Finally, it calls the function *WeightChange* to apply the weight w to the previously obtained positive and negative *BMDs. If $w \geq 0$, then *WeightChange* just calls the function *ApplyWeight* to multiply both the positive and negative *BMDs by the weight w. Otherwise, the positive *BMD is taken to be the previously obtained negative *BMD multiplied by $-w$; the negative *BMD is taken to be the previously obtained positive *BMD multiplied by $-w$.

```
function PosNegBMD(pair (w, v))
  if v = Λ
    then (w_P, v_P) ← (1, Λ)
         (w_N, v_N) ← (0, Λ)
    else (w_PL, v_PL), (w_NL, v_NL) ← PosNegBMD(Left(v))
         (w_PR, v_PR), (w_NR, v_NR) ← PosNegBMD(Right(v))
         (w_P, v_P) ← MakeBranch(Var(v), (w_PL, v_PL), (w_PR, v_PR))
         (w_N, v_N) ← MakeBranch(Var(v), (w_NL, v_NL), (w_NR, v_NR))
  end if
  (w_P, v_P), (w_N, v_N) ← WeightChange(w, (w_P, v_P), (w_N, v_N))
  return (w_P, v_P), (w_N, v_N)

function WeightChange(wtype w, pair (w_P, v_P), pair (w_N, v_N))
  if w ≥ 0
    then (w_P, v_P) ← ApplyWeight(w, (w_P, v_P))
         (w_N, v_N) ← ApplyWeight(w, (w_N, v_N))
    else (w_P, v_P) ← ApplyWeight(−w, (w_N, v_N))
         (w_N, v_N) ← ApplyWeight(−w, (w_P, v_P))
  end if
  return (w_P, v_P), (w_N, v_N)
```

Fig. 8. Procedure for decomposing a *BMD into positive and negative *BMDs.

C. Transforming a *BMD into a Unit-Weight *BMD

In order to build a stochastic circuit, we need to transform *BMDs with integer edge weights into *BMDs of a special form, which we call **unit-weight**: a *BMD is unit-weight if the absolute values of the edge weights of each non-terminal vertex sum to 1.

Figure 9 gives the unit-weight *BMD corresponding to the positive *BMD in Figure 7(a). The unit-weight *BMD corresponding to the negative *BMD in Figure 7(b) is just itself. The procedure for this transformation is given as pseudo-code in Figure 10.

Assume that the original *BMD has variables $x_1, x_2, \cdots, x_n$ and that they are ordered as $x_n < x_{n-1} < \cdots < x_1$. (Here the root vertex has variable x_n.) Each vertex in the unit-weight *BMD has three data members recording the weights:

1) **LeftWeight**: The edge weight of its left branch.
2) **RightWeight**: The edge weight of its right branch.
3) **FuncWeight**: The weight used to keep the function at that vertex unchanged. The function at a vertex in the

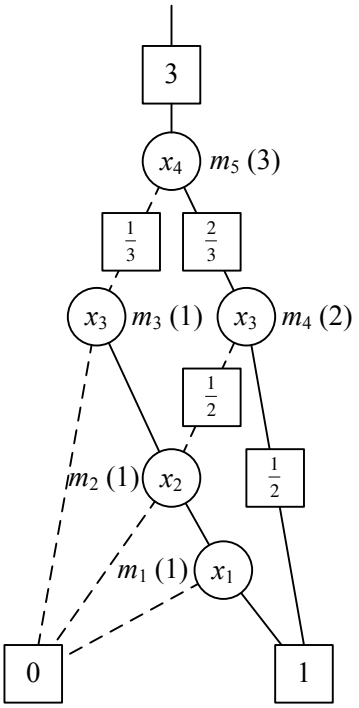

Fig. 9. The unit-weight *BMD corresponding to the *BMD in Figure 7(a). The numbers in parentheses gives the *FuncWeight* of the corresponding vertices.

```
function MakeUnitWeightBMD(*BMD OriginalBMD)
    UnitBMD ← OriginalBMD
    for each terminal vertex T of UnitBMD
        do T.FuncWeight ← TermWeight(T)
    end for
    for i ← 1 to n
        do for each vertex V of UnitBMD with variable x_i
                do V.LeftWeight ← ...
                       V.LeftWeight · V.LeftVertex.FuncWeight
                   V.RightWeight ← ...
                       V.RightWeight · V.RightVertex.FuncWeight
                   V.FuncWeight ← ...
                       abs(V.LeftWeight) + abs (V.RightWeight)
                   V.LeftWeight ← V.LeftWeight / V.FuncWeight
                   V.RightWeight ← V.RightWeight / V.FuncWeight
           end for
    end for
    RootEdgeWeight ← root.FuncWeight · RootEdgeWeight
    return UnitBMD
```

Fig. 10. Procedure for transforming a *BMD into a unit-weight *BMD.

unit-weight *BMD multiplied by its *FuncWeight* equals the function at the corresponding vertex in the original *BMD.

In Figure 9, we show the *FuncWeight* of each vertex in parentheses. For example, the *FuncWeight* for vertex m_4 equals 2 and the function for m_4 is $f'_4 = \frac{1}{2}x_1x_2 + \frac{1}{2}x_3$. The function for the corresponding vertex n_4 in Figure 7(a) is $f_4 = x_1x_2 + x_3 = 2f'_4$.

In the initialization, we set the *FuncWeight* for each terminal vertex to the weight of that vertex. Then the procedure modifies the edge weights for all the vertices with variable x_1; then for all the vertices with x_2; and so on through to x_n. For each vertex v in the unit-weight *BMD, let its *LeftWeight* and the *LeftWeight* of the corresponding vertex in the original *BMD be w_{UL} and w_{OL}, respectively. Let its *RightWeight* and the *RightWeight* of the corresponding vertex in the original *BMD be w_{UR} and w_{OR}, respectively. Let its *FuncWeight* be w_{UF}.

Denote the *FuncWeight* of its left child and its right child as w_{FL} and w_{FR}, respectively. We have the following equations to determine w_{UL}, w_{UR} and w_{UF}:

$$w_{UF} = |w_{FL} \cdot w_{OL}| + |w_{FR} \cdot w_{OR}|,$$
$$w_{UL} = \frac{w_{FL} \cdot w_{OL}}{w_{UF}},$$
$$w_{UR} = \frac{w_{FR} \cdot w_{OR}}{w_{UF}}.$$

Finally, we set the root edge weight w_U of the unit-weight *BMD to

$$w_U = w_O \cdot w_F(root),$$

where w_O is the root edge weight for the original *BMD and $w_F(root)$ is the *FuncWeight* for the root vertex.

If the original *BMD has integer edge weights, then the edge weights of the unit-weight *BMD built by the procedure *MakeUnitWeightBMD* are all rational numbers. The root edge weight w_U is an integer.

D. Transforming a Unit-Weight *BMD into a Stochastic Circuit

In our method, we transform both the positive and negative *BMDs into unit-weight *BMDs. (We refer to these as **UnitPosBMD** and **UnitNegBMD**, respectively.) Both of these have non-negative edge weights.

Given a unit-weight *BMD with non-negative edge weights, we can transform it directly into a stochastic circuit. For each vertex in the *BMD, we build a stochastic circuit with an output bundle that implements the function at that vertex. (Here, when we say "a bundle implements a function", we mean that the signal carried by the bundle equals the function output for all input combinations.)

The procedure is as follows. We first set n bundles of inputs such that their signals are equal to the Boolean inputs $x_1, x_2, \cdots, x_n$ of the original circuit. Also, we provide an input bundle with all bits equal to 1 (equivalent to a constant logical value of 1).

Next, we build bundles implementing the functions of vertices in the unit-weight *BMD with variable x_1; then bundles implementing the functions of vertices with variable x_2; and so on through to those vertices with variable x_n.

For the circuit corresponding to a non-terminal vertex v_k with variable x_i, suppose that the functions of its left child vertex and right child vertex are f_L and f_R, respectively, and that *LeftWeight* and *RightWeight* are w_L and w_R, respectively. We have $0 \le w_L \le 1$, $0 \le w_R \le 1$ and $w_L + w_R = 1$. According to Equation (8), the function of v_k is

$$f(v_k) = w_L f_L + w_R f_R x_i. \tag{10}$$

At this point, since we are building the circuit according to the order of the vertex indices, we have already constructed a bundle s_L implementing f_L and a bundle s_R implementing

f_R. To build the bundle implementing $f(v_k)$, we first build a shuffled AND on the input bundles s_R and x_i. Call the result of the shuffled AND s_C and its signal f_C. Since $P(x_i = 1) = 0$ or 1, we have

$$P(f_R = a, x_i = b) = P(f_R = a) \cdot P(x_i = b), \forall a, b \in \{0, 1\},$$

which means that the bits in the bundle s_R and the primary input bundle for x_i are independent. Thus, according to Equation (4), we have

$$f_C = f_R x_i.$$

If $w_L \neq 0$, then we build a bundleplexer with inputs s_L and s_C. We set the selecting ratio of this bundleplexer to be $s = w_L$ with respect to s_L. Thus, according to Equation (7), the output bundle of the bundleplexer implements the function

$$w_L f_L + (1 - w_L) f_C = w_L f_L + w_R f_R x_i,$$

and so implements the function of vertex v_k.

A circuit fragment illustrating these steps is shown in Figure 11. In the circuit, we use a gate called "SAND" to denote a shuffled AND operation and a gate called "BUX" to denote bundleplexing. (We denote bundles by crossing a single wire with a slash and writing the number of wires, N, next to it.) The number on a bundleplexer denotes its selecting ratio with respect to the input bundle that is bubbled. The same conventions are also used in Figure 12.

If $w_L = 0$, then $w_R = 1$ and Equation (10) simplifies to $f(v_k) = f_R x_i$. Thus, the output bundle s_C of the shuffled AND implements the function of the vertex v_k.

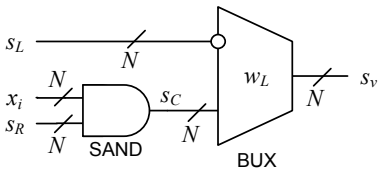

Fig. 11. A circuit fragment illustrating the computation of the function of a vertex v_k.

Since both *UnitPosBMD* and *UnitNegBMD* are unit-weight *BMDs with non-negative edge weights, we can build two stochastic circuits implementing the functions of the root vertices of these *BMDs.

Finally, we connect the output bundles of the two circuits to an analog counter. The 1's in the output bundle of the circuit for *UnitPosBMD* will increment the counter by w_{UP}, while the 1's in the output bundle of the circuit for *UnitNegBMD* will decrement it by w_{UN}, where w_{UP} and w_{UN} are the root edge weights of *UnitPosBMD* and *UnitNegBMD*, respectively. We call the increment and decrement coefficients of the counter the **scaling factors**.

For the *UnitPosBMD* shown in Figure 9 and the *UnitNegBMD* shown in Figure 7(b), we obtain the stochastic circuit shown in Figure 12. The output bundle of BUX1 implements the function of vertex m_4 and the output bundle of BUX2 implements the function of vertex m_5, the root vertex of *UnitPosBMD*. The output bundle of SAND3 implements the function of the root vertex of *UnitNegBMD*. Finally, we connect the output bundle of BUX2 and the output bundle of SAND3 to a counter. The output of the counter implements the function of the original *BMD in Figure 6. Thus, the circuit implements the same logic as the circuit in Figure 5.

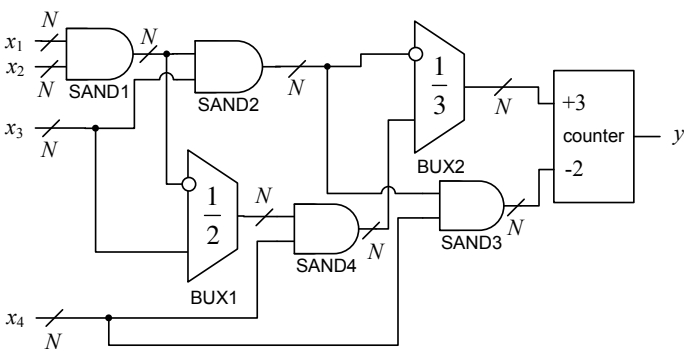

Fig. 12. The stochastic circuit obtained from the *UnitPosBMD* in Figure 9 and the *UnitNegBMD* in Figure 7(b). The number on the counter indicates the amount that it increments or decrements the count for each 1 on the corresponding bundle. The output of the counter implements the function of the original *BMD in Figure 6.

E. Summary of Synthesis Procedure

In summary, our procedure for synthesizing a stochastic circuit consists of the following five steps:

1) Build a *BMD for each output of the circuit.
2) Decompose each *BMD into positive and negative *BMDs.
3) Transform these into unit-weight *BMDs.
4) Transform the unit-weight *BMDs into stochastic designs with shuffled ANDs and bundleplexers.
5) Realize the outputs with cumulative increment and decrement operations on the outputs.

IV. Experimental Results

We chose 15 small benchmark circuits from the IWLS '93 set to test our synthesis technique. (For sequential circuits in this group, we have extracted the combinational part.) Table I shows some statistics of the original benchmark circuits and the corresponding stochastic circuits. Column "#Devices in Orig. Ckt." gives the number of devices in the original circuit and column "#Devices in Stoch. Ckt." gives the number of devices in the stochastic circuit. Here the devices are the shuffled AND elements and bundleplexers used in the design. The column "Ratio" gives the ratio of the number of devices in the stochastic circuit to the number of devices in the original circuit. We see that that this ratio is on average one and a half.

Given that the width of the bundles is finite, the outputs of the stochastic circuit might be erroneous. We analyze the **error ratio** defined as the number of outputs that return an incorrect value. For example, assume that for a given combination of inputs, the outputs of the stochastic circuit are $\vec{o_s} = (o_1, o_2, o_3, o_4) = (0.78, 1.01, -0.02, 0.16)$ and that the correct values are $\vec{o} = (1, 1, 1, 0)$. After discriminating, we get a Boolean output $\vec{o_s}' = (1, 1, 0, 0)$. Comparing $\vec{o_s}'$ with $\vec{o}$, we

Circuit	#Inputs	#Outputs	#Devices in Orig. Ckt.	#Devices in Stoch. Ckt.	Ratio (col. 5 / col. 4)
C17	5	2	14	26	1.86
b1	3	4	18	18	1.00
majority	5	1	18	23	1.28
lion	4	3	19	30	1.58
daio	5	4	26	29	1.12
mc	5	7	36	47	1.31
cm138a	6	8	43	104	2.42
bbtas	5	5	44	74	1.68
cm42a	4	10	49	61	1.24
tcon	17	16	58	73	1.26
beecount	6	7	62	108	1.74
decod	5	16	69	194	2.81
sqrt8ml	8	4	74	87	1.18
sqrt8	8	4	79	87	1.10
c8	28	18	184	272	1.48
Average					1.54

TABLE I
SYNTHESIS RESULTS FOR SELECTED IWLS '93 BENCHMARK CIRCUITS.

Circuit	Max. Scaling Factor	α: Width of Bundles over Max. Scaling Factor				
		5	10	20	50	100
C17	4	8.36	3.13	1.02	0.00	0.00
b1	3	5.63	1.72	0.00	0.16	0.00
majority	9	4.69	1.88	0.94	0.31	0.00
lion	4	4.27	1.56	0.31	0.00	0.00
daio	6	4.53	2.19	0.70	0.04	0.00
mc	6	3.97	2.12	0.42	0.07	0.00
cm138a	8	0.55	0.51	0.22	0.02	0.00
bbtas	7	5.84	1.91	0.78	0.09	0.00
cm42a	4	0.91	0.56	0.03	0.03	0.00
tcon	2	1.50	0.23	0.01	0.00	0.00
beecount	14	4.20	3.35	1.14	0.29	0.05
decod	16	4.81	1.90	0.72	0.11	0.05
sqrt8ml	24	3.56	1.76	0.82	0.39	0.04
sqrt8	24	6.60	1.52	0.86	0.12	0.12
c8	6	5.93	3.09	1.03	0.12	0.01
Average	9.13	4.36	1.83	0.60	0.12	0.02

TABLE II
ERROR PERCENTAGES VS. α, THE RATIO OF THE WIDTH OF THE BUNDLES TO THE MAXIMAL SCALING FACTOR.

find that 1 out of 4 bits is incorrect, so the error ratio is 25%. Of course, with larger bundle widths the error ratio will be lower.

In our experiments, we do an average across a number of input combinations with the following rule: if the number of inputs is less than or equal to 5, then we run through all the input combinations; otherwise, we randomly select $2^5 = 32$ input combinations and run experiments on them. Considering the inherent randomness in the circuit construction, we also run 20 trials for each input combination and average the results. (In our simulations, the randomness of the construction is generated by the standard C function `rand()`.) We find that the width of the bundles needed to obtain an error ratio below a given threshold is linearly proportional to the maximal scaling factor of all the counters.

Define α as the ratio of the width of the bundles to the maximal scaling factor. We run experiments to see how the error ratio changes with increasing α. We set α to five different values: 5, 10, 20, 50, 100. The result for each circuit is shown in Table II. The error ratio is shown in the form of percentages. The smaller the error ratio, the better the result. We also give the maximal scaling factor for each circuit. The width of the bundles in each circuit is the maximal scaling factor multiplied by α.

From Table II, we see that:

1) With α increasing, the error ratio decreases.
2) For all the circuits, the error ratio is below 4% when $\alpha = 10$.
3) For most of the circuits, the error ratio is below 1% when $\alpha = 20$.
4) When $\alpha = 100$, the error ratio is almost 0.

Some applications are characterized by a tolerance for less than perfectly accurate computation. For example, in image processing applications, a small error in a processed image will be masked by the limits of the display device and by the limits of human vision [1]. For such applications, a non-zero error ratio is acceptable. Suppose that we choose 1% as our error ratio threshold. Then we obtain $\alpha \approx 20$. Given that the maximal scaling factor is around 10, on average, the width of the bundles in the stochastic circuit will be roughly 200.

V. IMPLEMENTATION OF STOCHASTIC ELEMENTS WITH NANOWIRE CROSSBAR ARRAYS

General features of nanowire technology are illustrated in Figure 13. The connections between horizontal and vertical wires are FET-like junctions with nearly a one-to-one ratio, i.e., there is nearly always one FET-like junction per horizontal nanowire. This is a specific attribute of nanowire arrays, controlled through doping during self-assembly [3].

When high or low voltages are applied to input nanowires, the FET-like junctions that cross these develop a high or low impedance, respectively. Because the doping regions for the junctions are randomly placed across the crossbar, the connections are random. We exploit this randomness to implement the shuffled AND and bundleplexing constructs.

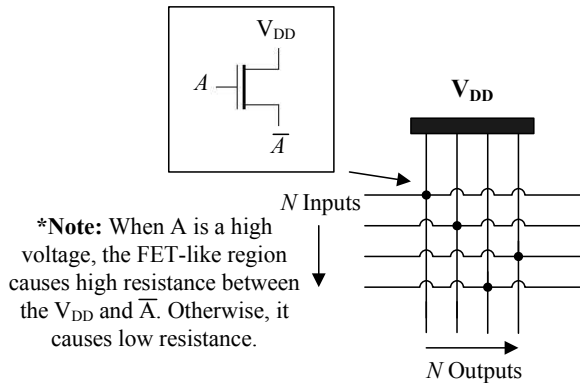

Fig. 13. The nanowire crossbar architecture.

A. Shuffled AND

In order to implement a shuffled AND on two input bundles, four crossbars are required. Two invert the signals on the input

bundles. Two more invert the results and compute the AND of pairs of randomly shuffled signals from each bundle.

This is illustrated in Figure 14. Consider the third wire from the bottom. It produces the AND of a_0 and b_1. To see this, note that the horizontal wire with input a_0 runs through a FET-like junction that inverts the value on the first vertical nanowire from the left. Similarly the input b_1 gets inverted on the second vertical nanowire from the right. These vertical nanowires are tied together by FET-like junctions on the horizontal nanowire that produces the output. This effectively computes the complement of the OR of the inverted values, so the AND of a_0 and b_1.

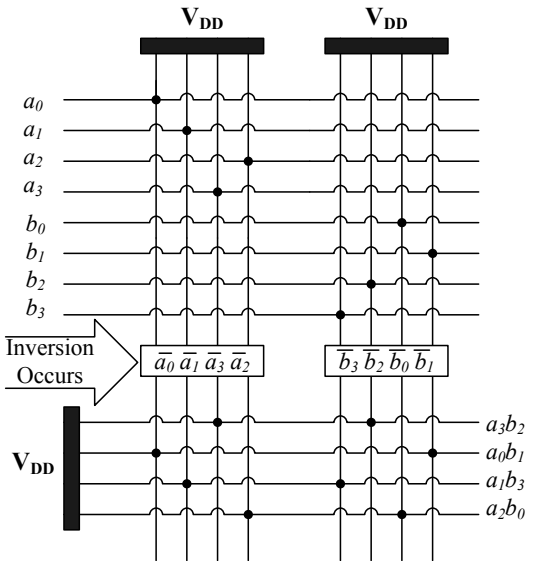

Fig. 14. The nanowire crossbar architecture implementing a shuffled AND.

B. Bundleplexing

In order to implement the bundleplexing operation on two input bundles, we set a different density of doping for the FET-like regions on the corresponding crossbars. The density dictates that a certain ratio of the output stream is affected by one input stream and the rest affected by the other input stream. The implementation is composed of three crossbars. Two select wires from the input bundles and invert the values. A third inverts the values a second time, producing the output.

This is illustrated in Figure 15, which shows a bundleplexer with a selecting ratio of $\frac{3}{4}$. We dope the first three vertical wires in the upper-most crossbar and the right-most vertical wire in the middle crossbar. This effectively chooses three bits from bundle A and one bit from bundle B and inverts these. The lower-most crossbar inverts these choices a second time. This gives the requisite output values: a randomly shuffled selection of bits from the two input bundles, with a ratio of $\frac{3}{4}$.

VI. Discussion & Future Directions

The trials with benchmarks in Section IV show that our technique produces circuits with tunable characteristics: with small bundle widths, the circuits require relatively little area

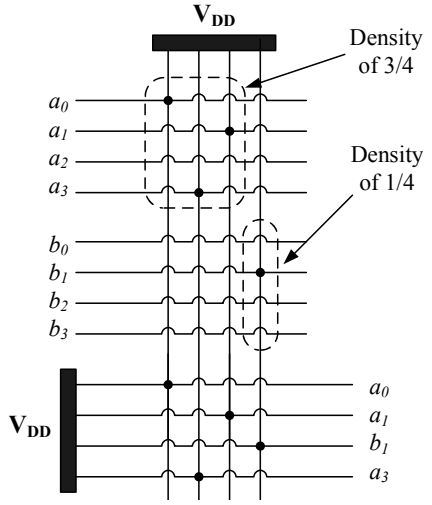

Fig. 15. The nanowire crossbar architecture implementing a bundleplexer.

yet compute somewhat inaccurately; with larger bundle widths, the circuits consume more area yet compute more accurately. With sufficiently wide bundles, the computation is perfectly accurate (i.e., no errors occur in the outputs). For many applications, such as control circuitry, perfect accuracy is a requisite. However, for other applications, such as image processing and telemetry, the tolerance for errors might be quite high. Stochastic circuits are particularly applicable in these domains.

Although not the focus of this paper, defect and fault-tolerance provide the impetus for our work. Indeed, with parallel stochastic bit streams, the random shuffles need not be perfect. There can be errors in the shuffling ANDs and bundleplexing: bits can be flipped or duplicated. With sufficiently wide streams, quantization at the output will map the resulting fractional weights to the correct Boolean values. We are working to analyze and optimize fault and defect tolerance with stochastic implementations.

Also, in future work, we will tailor the synthesis of stochastic circuits to particular forms of nanowire technology, such as hybrid Nano/CMOS architectures [6][7].

References

[1] Z. Asgar, S. Kodakara and D. Lilja, "Fault-Tolerant Image Processing Using Stochastic Logic," *Technical Report*, http://www.zasgar.net/zain/publications/publications.php, 1995.

[2] R. Bryant and Y. Chen, "Verification of Arithmetic Functions With Binary Moment Diagrams", Design Automation Conference, pp. 535–541, 1995.

[3] A. DeHon, "Nanowire-Based Programmable Architectures," *ACM Journal on Emerging Technologies in Computing Systems*, Vol. 1, No. 2, pp. 109–162, 2005

[4] International Technology Roadmap for Semiconductors, http://www.itrs.net/Links/2006Update/2006UpdateFinal.htm, 2006.

[5] Mission Statement, MARCO Center Functional Engineered Nano Architectonics (FENA), available at www.fena.org, 2006.

[6] G. Snider and R. Williams, "Nano/CMOS Architectures Using a Field-Programmable Nanowire Interconnect," *Nanotechnology*, No. 18, pp. 1–11, 2007.

[7] D. Strukov and K. Likharev, "CMOL FPGA: A Reconfigurable Architecture for Hybrid Digital Circuits with Two-Terminal Nanodevices," *Nanotechnology*, No. 16, pp. 888–900, 2005.