Generating Multiple Correlated Probabilities for MUX-Based Stochastic Computing Architecture

Yili Ding,Yi Wu,and Weikang Qian
University of Michigan-Shanghai Jiao Tong University Joint Institute
Shanghai Jiao Tong University, Shanghai, China
Email: {huan-ye, eejessie, qianwk}@sjtu.edu.cn

***Abstract*— Stochastic computing is a paradigm that performs computation on stochastic bit streams using conventional digital circuits. A general design for stochastic computing is a MUX-based architecture, which needs multiple constant probabilities as inputs. Previous approaches generate these probabilities by separate combinational circuits. The resulting designs are not area-efficient. In this work, we use the fact that these constant probabilities to the MUX can have correlation and propose two novel algorithms that produce low-cost circuits for generating these probabilities. Experimental results showed that our method greatly reduces the cost of generating constant probabilities for the MUX-based stochastic computing architecture.**

I. Introduction

Stochastic computing (SC), introduced in 1960s, is an alternative computation paradigm that uses ordinary digital circuits to operate on stochastic bit streams [6]. A stochastic bit stream encodes a value equal to the probability of a one in that stream. For example, the stream A in Fig. 1 represents the value $6/8$. With stochastic bit streams as inputs and outputs, digital circuits can be viewed as constructs that perform computation in the real domain.

Compared with conventional digital computation on binary radix numbers, SC has advantages such as strong fault tolerance and simple digital design. Since the encoding has uniform weight for each bit, an error occurring at any bit in the stream only changes the encoded value slightly. Further, with SC, many arithmetic functions, such as addition, multiplication, and division, can be implemented using very simple digital circuits [5]. For example, as shown in Fig. 1, an AND gate performs multiplication, since the probability of obtaining a one in the output stream equals the product of probabilities of ones in the two input streams.

Due to its low hardware cost, SC has been used in some hardware-demanding applications, such as real-time image processing [2] and low-density parity-check decoding [14].

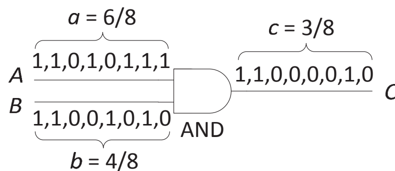

Fig. 1: An AND gate performs multiplication on values encoded by stochastic bit streams.

To utilize SC in different applications, several methods for synthesizing SC circuits were proposed recently [1], [8], [9], [12]. Among them, a popular method is to design a multiplexer (MUX)-based architecture [8], [9], [12]. The MUX-based design is simple and area-efficient. However, it is able to realize arbitrary arithmetic computation.

An important part of the MUX-based design is to generate multiple constant probabilities, which determine the function realized by the circuit. In previous works, these probabilities are generated separately. To produce each probability, independent random bits with probabilities 0.5[1] are first produced, for example, by using a linear feedback shift register (LFSR) [3]. Then a post-processing combinational circuit, such as a comparator or a chain of MUXes is applied to transform these 0.5 probabilities into the required probability [5], [15]. However, each post-processing circuit is in charge of only one constant input probability; it does not share any common logic with another.

However, as we will show in Section II-A, the multiple constant input probabilities of the MUX-based design do not have to be independent. Based on this fact, in this work, we consider sharing those post-processing circuits for generating different constant probabilities to reduce the total circuit area. However, given a single target probability, there are many different circuits that could produce the probability. With different choices of each individual circuit for generating each probability, the areas of the final shared circuits for generating multiple target probabilities are different. In this work, we propose two novel algorithms to synthesize small-area post-processing circuits for generating multiple constant probabilities.

The proposed methodology is important for designing MUX-based SC architecture, since the area of the circuit that provides the input probabilities could take about 80% of the entire SC circuit area [12]. Our experimental result showed that using our proposed synthesis method, we can reduce the area of the circuit for generating the input probabilities by up to 30%. Thus, our method helps reduce the total area of the MUX-based SC circuit.

The remainder of the paper is organized as follows. Section II introduces the background on MUX-based SC architecture and reviews several previous works on synthesizing circuits to generate probabilities for stochastic computation. Section III presents our assumptions and formulates the problem. Section IV presents a key concept used in our algorithm, namely mincost AND-inverter tree (MAIT). Section V presents our algorithms. Section VI shows the experimental results. Section VII concludes the paper.

II. Background and Related Work

A. MUX-Based Stochastic Computing Architecture

Several methods proposed to synthesize a SC circuit for implementing an arbitrary target function are all based on a general form shown in Fig. 2, which we refer to as MUX-based architecture [8], [9], [12]. This architecture contains a distribution-generating circuit and a MUX. The distribution-generating circuit transforms a Boolean input X into an output value S in the set $\{0, 1, \ldots, n\}$. If X is a random bit stream with probability x to be a one, then S will be a random variable and the probability of S to be i $(0 \leq i \leq n)$ will be a function on x, i.e., $P(S = i) = f_i(x)$. In [12], an adder was used as the distribution-generating circuit. Then $f_i(x)$ is a Bernoulli

[1]For simplicity, we will use "probability" to refer to the "probability of being a one".

distribution with respect to x: $f_i(x) = \binom{n}{i}x^i(1-x)^{n-i}$. In [9], an up-down counter was used. The counter is realized by a linear FSM where the input is a stochastic bit stream and the output is only determined by the current state. When the input bit is 1, the FSM will move to the next state and the output will be increased by 1. When the input bit is 0, the FSM will move to the previous state and the output will be decreased by 1. In this case, $f_i(x)$ is a special distribution corresponding to the steady-state distribution of a Markov chain.

If the intermediate value $S = i$, the output of the MUX, Y, will be set to its i-th input Z_i. In the MUX-based design, each Z_i is supplied with a random bit with a constant probability b_i. As a result, the probability of the output Y is a linear combination on the functions $f_i(x)$, i.e., $P(Y = 1) = \sum_{i=0}^{n} b_i f_i(x)$. By configuring the constant probabilities b_i's properly, the MUX-based SC architecture can implement an arbitrary target function [12].

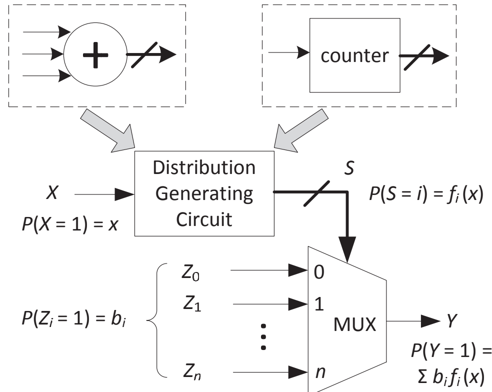

Fig. 2: MUX-based stochastic computing architecture that could realize an arbitrary target function.

One important feature of the MUX-based SC architecture is that those constant probabilities b_i's do not need to be independent. This is because due to the function of a MUX, at any time, only one input random bit Z_i will be selected as the output. Thus, the correlation among the constant input probabilities does not affect the output probability. In this work, we exploit this feature and propose to share the post-processing circuits for generating different constant probabilities to reduce the total area.

B. Related Work

There are several related works on designing post-processing combinational circuits to generate a *single* target probability. A few previous works considered the general situation that the input "source" probabilities could be arbitrary [13], [16].

However, it is usually easier to obtain input probabilities of 0.5. Thus, several other designs were proposed to transform a set of independent 0.5 probabilities into a target probability. These designs are all combinational circuits. It can be shown that with independent 0.5 probabilities as inputs, the output probabilities of any combinational circuits are of the form $\frac{C}{2^n}$, where $0 \leq C \leq 2^n$ is an odd integer. Thus, those designs all target at generating probabilities of the form $\frac{C}{2^n}$.

A common post-processing circuit is a comparator, as shown in Fig. 3(a). In order to produce a probability $\frac{C}{2^n}$, one input to the comparator is a random binary number R composed of n random bits of probability 0.5 and the other input is a constant binary number C. The comparator will output a one if $R < C$. The output probability is $\frac{C}{2^n}$ [15].

Another design is a chain of 2-to-1 MUXes, as shown in Fig. 3(b). The inputs $R_0, \ldots, R_{n-1}$ are independent random bits of probability 0.5, while $C_0, \ldots, C_{n-1}$ are deterministic bits. The output probability is $P(Y = 1) = \frac{C}{2^n}$, where $C = \sum_{i=0}^{n-1} C_i 2^i$ [5].

In the case that the stochastic bit stream represents a constant parameter (such as b_i's in the MUX-based design) instead of a variable (such as x in the MUX-based design), $C_0, \ldots, C_{n-1}$ are fixed. As a result, we can further simplify the chain of 2-to-1 MUXes to a chain of $(n-1)$ 2-input AND/OR gates, as was proposed in [7], Fig. 4(a) shows that when C_i = 0, the MUX is simplified to an AND gate and that when C_i = 1, the MUX is simplified to an OR gate. The signal A_i represents the output of the ith MUX. Fig. 4(b) shows an AND/OR chain for generating probability 7/16. Since $16 = 2^4$, only 3 gates are needed, in which two are OR gates and the other is an AND gate. Note that an OR gate can be realized with an AND gate and three inverters. Therefore, we can produce a probability $\frac{C}{2^n}$ $(0 \leq C \leq 2^n$ is odd) with $(n-1)$ 2-input AND gates and a number of inverters.

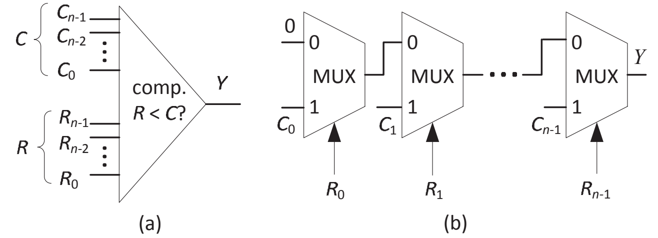

Fig. 3: Two post-processing circuits for generating a random bit with a specific probability. R_i's are independent random bits with probability 0.5. The output probability is $\frac{C}{2^n}$, where $C = \sum_{i=0}^{n-1} C_i 2^i$. (a) A comparator; (b) A chain of MUXes.

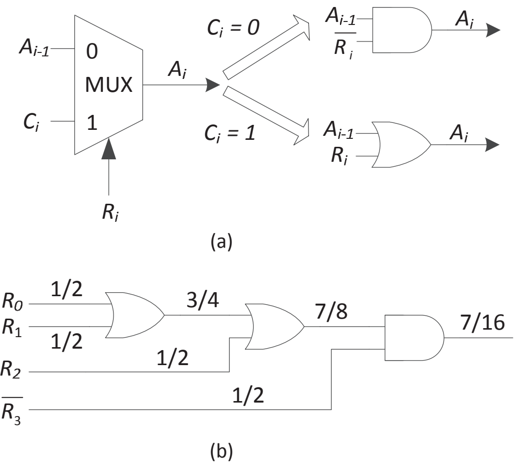

Fig. 4: (a) When C_i=0, the MUX can be simplified to an AND gate. When C_i=1, the MUX can be simplified to an OR gate. (b) In an example to generate the probability 7/16, the MUX chain is simplified to an AND/OR chain with two OR gates and one AND gate.

III. Assumptions and Problem Formulation

Our goal is to design a combinational circuit that generates multiple constant probabilities $b_1, \ldots, b_k$ used in the MUX-based SC architecture. Since it is usually easier to obtain "source" input probabilities of 0.5, for example, by using an LFSR, we assume that the inputs to the circuit are independent 0.5 probabilities.

Since combinational circuits can only generate probabilities of the form $\frac{C}{2^n}$, we assume that the actual target probabilities will be binary fractional numbers $\frac{C_1}{2^{n_1}}, \ldots, \frac{C_k}{2^{n_k}}$, which are close approximations to $b_1, \ldots, b_k$, respectively. Without loss of generality, we assume that C_i's are odd numbers and $n_1 \geq n_2 \geq \cdots \geq n_k$. As we discussed in Section II-A, those target probabilities $\frac{C_1}{2^{n_1}}, \ldots, \frac{C_k}{2^{n_k}}$ can be correlated.

Previous work [7] showed that in order to generate a probability $\frac{C}{2^n}$ ($0 \leq C \leq 2^n$ is odd), the minimal number of input 0.5 probabilities needed is n. Thus, in order to generate the first output probability $\frac{C_1}{2^{n_1}}$, we need a minimum of n_1 probabilities of 0.5. Since the k output probabilities can be correlated, we are able to generate the other probabilities $\frac{C_2}{2^{n_2}}, \ldots, \frac{C_k}{2^{n_k}}$ with these n_1 probabilities of 0.5. Therefore, n_1 is the minimal number of probabilities of 0.5 needed for generating the output probabilities $\frac{C_1}{2^{n_1}}, \ldots, \frac{C_k}{2^{n_k}}$. Since generating each input probability of 0.5 takes some cost, we assume that we are supplied with just n_1 probabilities of 0.5.

Any combinational circuit can be realized by a network of 2-input AND gates and inverters, which is also known as an AND-inverter graph (AIG) [10]. Thus, we focus on designing an AIG to generate the multiple target probabilities. We measure the cost of an AIG by its number of AND gates, ignoring the inverters. This is a typical way in measuring the cost of an AIG [10].

Based on the above assumptions, the problem we try to solve is formulated as follows:

> Synthesize a low-cost AIG to generate k correlated target probabilities $\frac{C_1}{2^{n_1}}, \ldots, \frac{C_k}{2^{n_k}}$ ($0 \leq C_i \leq 2^{n_i}$ is odd and $n_1 \geq \cdots \geq n_k$) from n_1 independent probabilities of 0.5. The cost of an AIG is measured by its number of AND gates.

Before we discuss our proposed solution, we want to point out the difference of this problem to the traditional logic synthesis problem. A possible solution one can imagine is to exploit traditional logic synthesis techniques. To do this, one can first design a Boolean function to generate each individual probability and then combine all the individual circuits into a multiple-output circuit by traditional techniques such as extraction and substitution [4]. However, traditional logic synthesis only explores optimal circuit realization for a set of fixed Boolean functions. In contrast, when synthesizing circuit for generating probabilities, we need to consider many different sets of Boolean functions, since they could realize the same set of output probabilities. One observation is that since all the inputs are probabilities of 0.5, the permutation and negation of the input variables does not change the output probabilities. However, it changes the output Boolean functions. Furthermore, even two Boolean functions that are not equivalent under input permutation and negation could lead to the same set of output probabilities. An example of this is shown in Fig. 5. Both circuits realize the probability $\frac{7}{16}$. However, it is not hard to see that the Boolean functions of the two circuits are not equivalent even under input permutation and negation. In summary, traditional logic synthesis has limited power in finding the optimal solution to our problem. A set of new techniques should be developed, We will discuss our proposed solution in the following sections.

IV. Mincost AND-Inverter Tree (MAIT)

In order to design a low-cost circuit to generate multiple output probabilities that can be correlated, our basic strategy is to first build individual circuit for generating each probability and then share the common parts of these circuits.

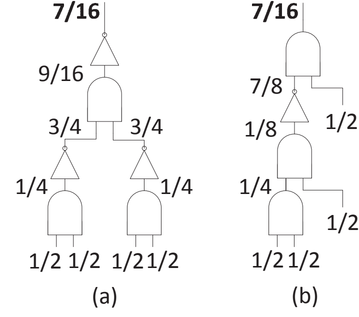

Fig. 5: Two circuits generating the probability $\frac{7}{16}$. The value near each wire denotes the probability of a one on that wire.

Given a target probability $\frac{C}{2^n}$ ($0 \leq C \leq 2^n$ is odd), there are many different AIGs with different costs that could generate the probability from input 0.5 probabilities. To get a low-cost final design, a heuristic is to realize each individual probability with an AIG of the minimal cost. We refer to such an AIG as a *mincost AIG* for probability $\frac{C}{2^n}$. In this work, we only consider building mincost AIG for realizing each target probability and sharing among mincost AIGs. We have the following claim on an mincost AIG:

Lemma 1

A mincost AIG for a probability $\frac{C}{2^n}$ ($0 \leq C \leq 2^n$ is odd) is in the form of a tree of AND gates and inverters and its cost is $(n-1)$, i.e., it has $(n-1)$ AND gates. □

Proof: Previous work [7] showed that in order to generate a probability $\frac{C}{2^n}$ ($0 \leq C \leq 2^n$ is odd), the minimal number of input 0.5 probabilities needed is n. Consider any AIG that generates the probability $\frac{C}{2^n}$. It has at least n inputs. Thus, it should have at least $(n-1)$ 2-input AND gates. In other words, its cost is at least $(n-1)$. Further, as we showed in Section II-B, we can indeed realize the probability $\frac{C}{2^n}$ with an AIG of cost $(n-1)$. Thus, a mincost AIG has cost $(n-1)$. For any mincost AIG, since it has n inputs and $(n-1)$ 2-input AND gates, it should be in the form of a tree of AND gates and inverters. □

To emphasize the property that a mincost AIG is a tree, in what follows, we will refer to a mincost AIG as a *mincost AND-inverter tree (MAIT)*. Note that each gate[2] in a MAIT is associated with a probability. The probability of each primary input is 0.5. The probability of each internal gate is equal to the output probability of that gate, which can be calculated recursively: the probability of an AND gate is the product of its two input probabilities; the probability of an inverter is one minus its input probability. Fig. 6(a) shows a MAIT for the probability $\frac{23}{32}$.

Our proposed algorithm to merge different MAITs for different probabilities is based on merging cuts of those MAITs, which are defined as follows:

Definition 1

A **cut** *of a MAIT is a set of gates in the tree that satisfies the following conditions:*

1) *Given two arbitrary gates in the set, any one of them is not an ancestor of the other.*
2) *Any primary input either belongs to the set or is a descendant of a gate in the set.* □

Based on the definition of a cut, we further define:

[2] In this work, each primary input of a circuit is also treated as a gate.

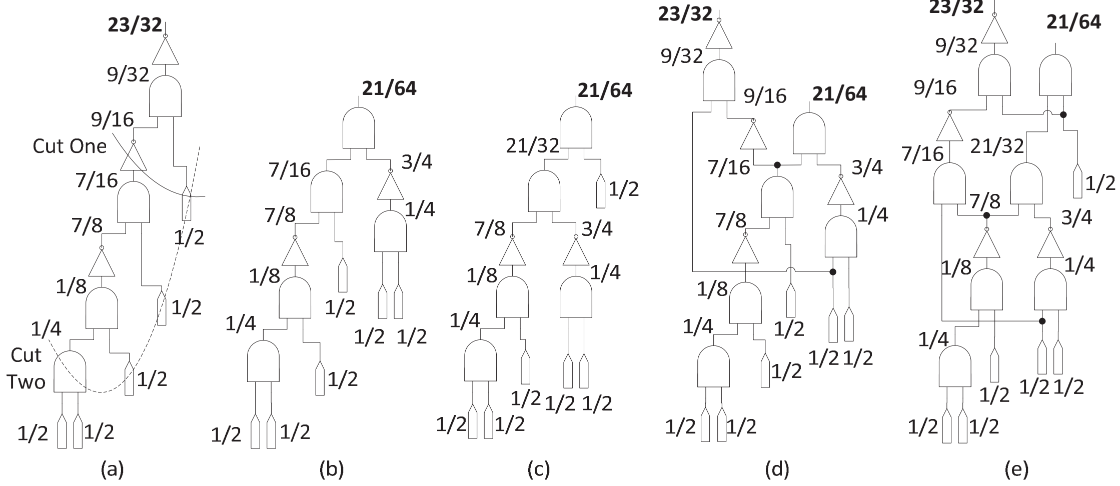

Fig. 6: The mincost AND-inverter trees (MAITs) and the merged AIGs. The value near each gate denotes the probability of that gate. (a) A MAIT for the probability $\frac{23}{32}$; (b) A MAIT for the probability $\frac{21}{64}$; (c) Another MAIT for the probability $\frac{21}{64}$; (d) An AIG obtained by merging the MAITs in (a) and (b). The AIG is also the result produced by the pairwise MAIT-merging algorithm; (e) An AIG obtained by merging the MAITs in (a) and (c).

Definition 2
The **set of values for a cut** *of a MAIT is the set of probabilities of those gates in the cut. Since some probabilities of the gates in the cut could be the same, therefore, the set of values for a cut is a multiset, i.e, a set in which elements are allowed to appear more than once.* □

Fig. 6(a) shows two cuts of a MAIT for the probability $\frac{23}{32}$. The set of values for Cut One is $\{\frac{9}{16}, \frac{1}{2}\}$ and the set of values for Cut Two is $\{\frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}\}$.

V. Algorithm

In this section, we present two algorithms for solving the problem of generating multiple target probabilities formulated in Section III.

A. Basic Strategy

Our basic strategy is to first build a MAIT for each output probability and then merge these MAITs. For example, suppose that we want to generate two target probabilities $\frac{23}{32}$ and $\frac{21}{64}$. Fig. 6(a) shows a MAIT for $\frac{23}{32}$ and Fig. 6(b) shows a MAIT for $\frac{21}{64}$. Fig. 6(d) shows an AIG generated by merging a common subtree of both MAITs that is rooted at an AND gate of probability $\frac{7}{16}$. This merging reduces the total cost by 3.

One thing to notice is that there may exist many MAITs for a single probability and different choices of those MAITs may result in different final AIGs. For example, Fig. 6(c) shows a MAIT for the probability $\frac{21}{64}$ different than the one shown in Fig. 6(b). If we merge this MAIT with the MAIT for the probability $\frac{23}{32}$ shown in Fig. 6(a), we get another AIG that generates the two target probabilities $\frac{23}{32}$ and $\frac{21}{64}$, as shown in Fig. 6(e). This AIG has one more AND gate than the AIG shown in Fig. 6(d).

Therefore, to minimize the cost of the final AIG, ideally, we need to first find all the MAITs for each target probability and then consider all combinations of MAITs for the target probabilities to eventually choose a combination that leads to an AIG with the minimal cost.

B. Pairwise MAIT-Merging Algorithm

In this section, we present the first algorithm. This algorithm begins by finding all the MAITs for each target probability. However, since considering all combinations of the MAITs for the target probabilities is computationally intractable, the algorithm actually considers the MAITs in pairs to choose a good combination. The procedure is shown in Algorithm 1.

Algorithm 1 Pairwise MAIT-merging algorithm.

1: {Given a set of binary fractional probabilities $\{\frac{C_1}{2^{n_1}}, \dots, \frac{C_k}{2^{n_k}}\}$.}
2: **for** $i = 1$ to k **do** $tset_i \Leftarrow buildMAITs(\frac{C_i}{2^{n_i}})$;
3: $maxSaveSum \Leftarrow 0$;
4: **for each** MAIT $tree_1$ in $tset_1$ **do**
5:     **for** $i = 2$ to k **do**
6:         $maxSave[i] \Leftarrow 0$;
7:         **for each** MAIT $tree_i$ in $tset_i$ **do**
8:             $save \Leftarrow mergeTwoMAITs(tree_1, tree_i)$;
9:             **if** $save > maxSave[i]$ **then**
10:                 $maxSave[i] \Leftarrow save$; $bestTree[i] \Leftarrow tree_i$;
11:     $saveSum \Leftarrow \sum_{i=2}^{k} maxSave[i]$;
12:     **if** $saveSum > maxSaveSum$ **then**
13:         $maxSaveSum \Leftarrow saveSum$; $glbBestTree[1] \Leftarrow tree_1$;
14:         **for** $i = 2$ to k **do** $glbBestTree[i] \Leftarrow bestTree[i]$;
15: **return** $buildAIG(glbBestTree[1], \dots, glbBestTree[k])$;

1) Building All MAITs for a Target Probability: The first step of the algorithm is to find all MAITs for each target probability by invoking the function $buildMAITs$. The set of the MAITs for the i-th output probability $\frac{C_i}{2^{n_i}}$ is stored in the set $tset_i$.

It can be seen that a MAIT for a target probability corresponds to a decomposition of the probability with one minus operation and multiplication until the source probability 0.5 is reached. The procedure $buildMAITs$ applies this idea to obtain all the MAITs for a target probability $\frac{C}{2^n}$. The procedure is implemented recursively based on whether the root is an AND gate or an inverter:

1) If the root is an AND gate, then its two input probabilities, which are of the form $\frac{D_1}{2^{m_1}}$ and $\frac{D_2}{2^{m_2}}$, satisfy that $\frac{C}{2^n} = \frac{D_1}{2^{m_1}} \cdot \frac{D_2}{2^{m_2}}$. We consider all pairs of $\frac{D_1}{2^{m_1}}$ and $\frac{D_2}{2^{m_2}}$ that satisfy the above equation. For each pair, we recursively find the two sets of MAITs for the values $\frac{D_1}{2^{m_1}}$ and $\frac{D_2}{2^{m_2}}$, respectively. Then, we build the set of MAITs for $\frac{C}{2^n}$ by connecting the root of one

MAIT for $\frac{D_1}{2^{m_1}}$ and the root of one MAIT for $\frac{D_2}{2^{m_2}}$ using an AND gate.

2) If the root is an inverter, then its input probability is equal to $(1 - \frac{C}{2^n})$. We recursively find the set of MAITs for that probability. Notice that in a MAIT, it is meaningless to have two inverters connected together. Thus, in obtaining the set of MAITs for $\frac{C}{2^n}$, we will only choose a MAIT for $(1-\frac{C}{2^n})$ whose root is an AND gate and connect an inverter to the root of that MAIT to form a MAIT for $\frac{C}{2^n}$.

To reduce some redundant computation, in our implementation, we also hash the set of MAITs for each intermediate probability we encounter. When a probability is visited again, we will just return its set of MAITs from the hash table.

2) Searching A Good Combination of MAITs: After generating all the MAITs for each target probability, we search the combinations of the MAITs for all the target probabilities and pick a combination that could be merged into a low-cost AIG. This procedure corresponds to Lines 3–15 of Algorithm 1. Ideally, to obtain the optimal combination, we need to consider all the combinations of the MAITs for all the target probabilities. However, doing this will greatly increase the runtime, since the total number of combinations is exponential to the number of target probabilities.

In our problem, we have a constraint that the number of input probabilities is n_1. Therefore, all of the n_1 input probabilities are used to generate the target probability $\frac{C_1}{2^{n_1}}$, while only part of these input probabilities are used to generate each remaining target probability $\frac{C_i}{2^{n_i}}$ $(2 \leq i \leq k)$. Based on this fact and the runtime concern, we apply the following heuristic in searching for a good combination of MAITs: we pick a MAIT for the first probability $\frac{C_1}{2^{n_1}}$ as a "base" graph and merge the MAIT for each of the other probabilities with this "base" graph.

Specifically speaking, we first choose a MAIT $tree_1$ from $tset_1$, the set of all MAITs for $\frac{C_1}{2^{n_1}}$. Then, for each $i = 2, \ldots, k$, we pick a MAIT $tree_i$ from $tset_i$, the set of all MAITs for $\frac{C_i}{2^{n_i}}$. Then, for each $i = 2, \ldots, k$, we merge $tree_i$ with $tree_1$. Although not exact, the cost of the merging based on this set of MAITs $tree_1, \ldots, tree_k$ is approximated as the sum of the cost saving by merging $tree_i$ with $tree_1$ over all $i = 2, \ldots, k$. With this strategy, we only need to consider a pair of MAITs in the form $(tree_1, tree_i)$, where $2 \leq i \leq k$. Thus, instead of searching for the combination $(tree_1, tree_2, \ldots, tree_k)$ that results in the optimal merging, we only need to search for a single $tree_i$ (for each $i = 2, \ldots, k$), such that merging $tree_i$ with $tree_1$ leads to the largest cost saving among all $tree_i \in tset_i$. This significantly reduces the runtime.

Lines 3–15 of Algorithm 1 implement the above strategy. In the outermost loop, it chooses a MAIT $tree_1 \in tset_1$. Then, for each other target probability $\frac{C_i}{2^{n_i}}$, it picks a MAIT $tree_i \in tset_i$. It calls the function $mergeTwoMAITs$ (to be discussed in Section V-B3), which returns the maximal cost saving that can be achieved by merging $tree_1$ and $tree_i$. By considering all the MAITs in the set $tset_i$, the algorithm chooses a MAIT $bestTree[i]$ in that set so that the amount of the cost saving achieved by merging $tree_1$ and $bestTree[i]$ is the largest. Let that largest amount of cost saving be $maxSave[i]$. Then, the total amount of cost saving by merging all the selected MAITs $bestTree[2], \ldots, bestTree[k]$ with the "base" MAIT $tree_1$ is $\sum_{i=2}^{k} maxSave[i]$. Note that the cost saving and the corresponding $bestTree[i]$'s are only for a specific $tree_1$ we pick at the beginning. To obtain the global best choice on $tree_1, tree_2, \ldots, tree_k$, we apply the above procedure for all the choices of $tree_1 \in tset_1$. Finally, we select the global best choice on the MAITs and build the AIG based on that choice by merging the MAIT for the i-th target probability with the MAIT for the first target probability for all $i = 2, \ldots, k$. After this, a traditional logic synthesis routine is applied to further optimize the AIG for its cost.

3) Merging Two MAITs: A core subroutine of the algorithm is the procedure $mergeTwoMAITs$, which merges two MAITs. It takes two MAITs as the input, with one MAIT $tree_1$ for the probability $\frac{C_1}{2^{n_1}}$ and the other MAIT $tree_i$ for a probability $\frac{C_i}{2^{n_i}}$ $(2 \leq i \leq k)$. It merges these two MAITs into an AIG that has the minimal cost. It returns the amount of cost saving, which is the sum of the costs of the two input MAITs minus the cost of the merged AIG.

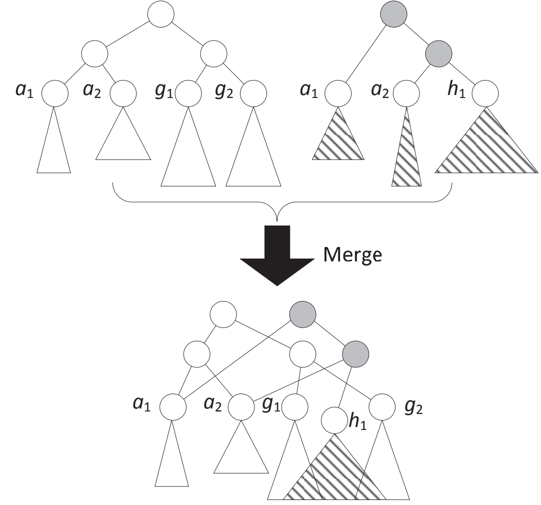

Fig. 7: An illustration of the procedure of merging two MAITs.

Our strategy to merge two MAITs is based on matching the cuts of these two MAITs, which is illustrated in Fig. 7. Specifically, we pick a cut c_1 from the first MAIT $tree_1$ and a cut c_2 from the second MAIT $tree_i$. We compare the set of values for the cut c_1 and the set of values for the cut c_2. Assume that the set of common values of these two sets are $\{a_1, \ldots, a_N\}$. Then, for each $j = 1, \ldots, N$, we replace the gate on the cut c_2 with probability a_j by the gate on the cut c_1 with the same probability.

Notice that one constraint of the synthesis problem is that the number of input 0.5 probabilities is n_1. Assume that besides $a_1, \ldots, a_N$, the other probabilities on the cut c_1 are $g_1, \ldots, g_L$ and the other probabilities on the cut c_2 are $h_1, \ldots, h_M$. Due to the assumption that $n_1 \geq n_i$, the number of 0.5 probabilities used to generate the probabilities $g_1, \ldots, g_L$ is larger than or equal to the number of probabilities used to generate the probabilities $h_1, \ldots, h_M$. In order to satisfy the constraint on the number of input probabilities, we merge the 0.5 probabilities used to generate the probabilities $h_1, \ldots, h_M$ with those 0.5 probabilities used to generate the probabilities $g_1, \ldots, g_L$, as shown in Fig. 7.

It is easily seen that the proposed method of merging MAITs does not introduce reconvergent path into the final AIG. In other words, for every pair of gates in the final AIG, there exists at most one path between that pair of gates. This is an important property, because the output probability of each AND gate in the final AIG is still equal to the product of its two input probabilities. As a result, each output probability of the final AIG is still equal to the output probability of an original MAIT. The merged AIG correctly realizes the set of target probabilities.

To find the optimal merging, we consider all pairs of cuts from the two MAITs and finally choose the best pair of cuts for merging.

C. Quick MAIT-Merging Algorithm

The pairwise MAIT-merging algorithm shown in the previous section turns out to be time-consuming. One reason is that it considers all possible cuts in a MAIT for merging purpose. To reduce the time complexity, we propose the second algorithm. It only focuses on merging a special type of cut in a MAIT, which we refer to as a $\frac{1}{4}$-cut:

Definition 3
*A $\frac{1}{4}$***-cut** *of a MAIT is a cut that only consists of gates of the probability $\frac{1}{2}$ or $\frac{1}{4}$.* □

For example, Cut Two in Fig. 6(a) is a $\frac{1}{4}$-cut.

Note that there is only one way to generate the probability $\frac{1}{4}$: using an AND gate with two input probabilities as $\frac{1}{2}$. Therefore, if we can merge two gates of probability $\frac{1}{4}$ in two different MAITs, we can reduce the cost of final AIG by 1. The rationale behind this algorithm is that for any target probability, we can usually find a MAIT of the probability such that it has a $\frac{1}{4}$-cut with many $\frac{1}{4}$'s. Then, merging those $\frac{1}{4}$ cuts for different target probabilities could lead to a significant cost saving.

1) Building AIG by Merging $\frac{1}{4}$-Cuts: In order to build the AIG for k target probabilities $\frac{C_1}{2^{n_1}}, \ldots, \frac{C_k}{2^{n_k}}$, we first pick k MAITs $tree_1, \ldots, tree_k$ for the k probabilities, respectively. Then, we pick k $\frac{1}{4}$-cuts $c_1, \ldots, c_k$ from these k MAITs, respectively. A core step in building the final AIG is to merge the k MAITs based on the k selected $\frac{1}{4}$-cuts so that the cost saving is as large as possible. Besides, we should stick to the constraint that the number of input 0.5 probabilities of the AIG is n_1.

Assume that the number of $\frac{1}{4}$'s on the cut c_i is q_i $(1 \leq i \leq k)$. Let $r = \max\{q_1, \ldots, q_k\}$. It's easy to see that $2r \leq n_1$. Our strategy of merging the k MAITs based on the k selected $\frac{1}{4}$-cuts includes the following two steps:

1) Build a "base" graph that consists of $(n_1 - 2r)$ primary input of probabilities 0.5 and r AND gates, each with two input probabilities of 0.5 and an output probability of $\frac{1}{4}$. Note that the graph has n_1 input 0.5 probabilities in total.
2) Merge each MAIT $tree_i$ $(1 \leq i \leq k)$ with the "base" graph. There are q_i AND gates of probability $\frac{1}{4}$ on the cut c_i of $tree_i$. We merge these q_i AND gates with the q_i AND gates in the "base" graph. This merging uses $2q_i$ input 0.5 probabilities of the "base" graph, which still has $(n_1 - 2q_i)$ "unmerged" input 0.5 probabilities. Note that besides q_i probabilities of $\frac{1}{4}$, the cut c_i contains $(n_i - 2q_i)$ probabilities of 0.5. Since by our assumption, $n_i \leq n_1$, we can merge the remaining $(n_i - 2q_i)$ 0.5 probabilities on the cut c_i with $(n_i - 2q_i)$ "unmerged" 0.5 probabilities of the "base" graph. This merging reduces q_i AND gates.

Using the above procedure, the number of input 0.5 probabilities of the resulting AIG is n_1. Further, the resulting AIG has no reconvergent path, which means that the set of k output probabilities of the AIG is equal to the set of k output probabilities of the k separate MAITs used in the merging.

The cost saved by the above procedure is

$$\sum_{i=1}^{k} q_i - r, \tag{1}$$

since by merging each MAIT $tree_i$ with the "base" graph, q_i AND gates are saved, but the construction of the "base" graph requires an extra of r AND gates.

2) Choosing the Best MAIT and Its Best $\frac{1}{4}$-Cut: We want to choose a MAIT for each target probability and a $\frac{1}{4}$-cut of that MAIT to maximize the cost saved, which is calculated by Eq. (1). In fact, in order to maximize the value calculated by Eq. (1), we only need to maximize the value q_i for each target probability $\frac{C_i}{2^{n_i}}$. In other words, we only need to choose a MAIT for $\frac{C_i}{2^{n_i}}$ and a $\frac{1}{4}$-cut of that MAIT so that the number of $\frac{1}{4}$'s on that cut is the largest. This means that we only need to consider each individual MAIT, which is much simpler than the pairwise MAIT-merging algorithm, which needs to examine combinations of MAITs.

Further, we notice that given a MAIT for a target probability, it is very easy to find a $\frac{1}{4}$-cut of it that contains the maximal number of $\frac{1}{4}$'s. Indeed, we can obtain the best cut by performing a depth-first search (DFS) on the MAIT, starting from its root. The DFS finishes recursion and returns only when it encounters a gate of either probability $\frac{1}{2}$ or probability $\frac{1}{4}$; otherwise, it recurses on the subtrees of the current gate. Each time the DFS encounters a gate of either probability $\frac{1}{2}$ or probability $\frac{1}{4}$, it marks that gate before it returns. We can see that after the DFS finishes, all the marked gates form a $\frac{1}{4}$-cut of the MAIT and that cut contains the maximal number of $\frac{1}{4}$'s.

Now the remaining part of the algorithm is to find all the MAITs of a target probability. This can be achieved in a similar way as we showed in Section V-B1. Once we have found all the MAITs, we will choose the best one such that its best cut contains the maximal number of $\frac{1}{4}$'s among all the best cuts of all the MAITs.

In summary, the quick MAIT-merging algorithm first finds the best MAIT for each individual target probability, which has a $\frac{1}{4}$-cut with the maximal number of $\frac{1}{4}$'s. Then, it builds the AIG by merging these $\frac{1}{4}$ cuts. We notice that the merged AIG may have some functional redundant AND gates. Thus, we further apply traditional logic synthesis to minimize the AIG using logic synthesis tool ABC [11]. The synthesis commands we use in ABC are as follows: "balance; rewrite -l; rewrite -lz; balance; rewrite -lz; balance".

VI. Experimental Results

To demonstrate the performance of the proposed algorithms, we carried out two experiments. The main factor that affects the performance of the proposed algorithms is the number of the target probabilities. We studied how this factor affects the performance of the proposed algorithms.

The circuits generated by our algorithms were compared with a design realized by a simple modification to the previous approach [7]. We select this approach since among all the previously proposed designs that transform input 0.5 probabilities into a target probability, this approach generates a design with the minimal number of AND gates. The design generated by the approach is in the form of AND/OR chain. However, the approach [7] only realizes a single target probability. For our purpose of realizing multiple target probabilities that can be correlated, we modify it as follows. We first generate a single chain for each individual target probability. Then, we share the common gates on the multiple AND/OR chains for realizing different target probabilities. We call this design a "shared chain" design. For fair comparison, the "shared chain" design is transformed into an AIG.

In the first experiment, we studied the effect of the number of target probabilities on the performances of different approaches. We set the number of target probabilities to be $10, 20, \ldots, 100$. For each number of target probabilities, we randomly generated 100 sets of target probabilities with size equal to that number. The result is presented as the average over the 100 sets. Because the pairwise MAIT-merging algorithm is time-consuming and memory-consuming, the powers of the denominators of all the target probabilities were chosen to

TABLE I: The number of AND gates in AIG and runtime for different approaches in generating multiple probabilities.

	shared chain		pairwise MAIT-merging			quick MAIT-merging		
#targets	#AND gates	runtime/s	#AND gates	runtime/s	save/% (shared)	#AND gates	runtime/s	save/% (shared)
10	41	0.01159	29	2.01	29.27	37	0.01081	9.76
20	65	0.02116	49	2.82	24.62	63	0.01698	3.08
30	85	0.04912	67	3.59	21.18	85	0.02249	0.00
40	102	0.07507	85	4.57	16.67	106	0.02934	-3.92
50	120	0.11194	102	4.97	15.00	125	0.03779	-4.17
60	133	0.16073	116	5.67	12.78	143	0.04578	-7.52
70	147	0.21248	131	6.41	10.88	159	0.05261	-8.16
80	159	0.19543	144	6.81	9.43	176	0.06420	-10.69
90	172	0.30470	159	7.28	7.56	192	0.06911	-11.63
100	183	0.38238	171	8.75	6.56	206	0.07854	-12.57

be no more than 10. Table I shows the number of AND gates and the runtime of our proposed approaches, compared with the "shared chain" approach. The "save" columns list the percentage of the number of AND gates saved by our proposed algorithms in comparison with the "shared chain" design. To further study the effect of our methods on actual circuit area, we map the AIG using logic synthesis tool ABC with MCNC standard cell library. Table II shows the mapped area of the three approaches as well as the area saving of our proposed algorithms.

From Tables I and II, we can see that both the number of AND gates and the mapped area generated by the pairwise MAIT-merging algorithm are smaller than the "shared chain" design. For the quick MAIT-merging algorithm, when the number of targets is larger than 30, the number of AND gates of the AIG generated by the algorithm is larger than that of the "shared chain" design. The reason is that the powers of the denominators of all the targets are no more than 10. With the number of targets increasing, more intermediate probabilities of the AND/OR chains can be shared; this causes a larger saving of AND gates than that of the quick MAIT-merging algorithm. Based on this observation, an interesting idea is to mix the quick MAIT-merging algorithm with the "shared chain" method, which could potentially take the advantage of both. We will further explore this idea in our future work.

In terms of the runtime, the quick MAIT-merging algorithm is much faster than the pairwise MAIT-merging algorithm, although it is slower than the "shared chain" approach, which takes almost no time (so we do not show it). However, the runtime of quick MAIT-merging algorithm is still almost negligible.

TABLE II: The mapped area for different approaches in generating multiple probabilities.

	shared chain	pairwise MAIT-merging		quick MAIT-merging	
#targets	area	area	save/% (shared)	area	save/% (shared)
10	88	67	23.86	81	7.95
20	144	110	23.61	134	6.94
30	189	149	21.16	180	4.76
40	228	185	18.86	223	2.19
50	269	223	17.10	262	2.60
60	302	255	15.56	299	0.99
70	334	288	13.77	335	-0.30
80	366	320	12.57	369	-0.82
90	394	351	10.91	404	-2.54
100	423	381	9.93	439	-3.78

In the second experiment, we demonstrated the cost of the quick MAIT-merging algorithm in realizing target probabilities with denominators that have a large power. The setting of this experiment is almost the same as the first experiment except that the maximal power of denominators of the targets is raised to 31. The results are shown in Table III and Table IV for the number of AND gates in the AIG and the area after technology mapping, respectively. We did not show the results of the pairwise MAIT-merging algorithm, because it failed in generating probabilities with denominators that have a large power. From Table III and Table IV, we can see that the quick MAIT-merging algorithm has a very satisfying performance in reducing the cost (in both the number of AND gates and the mapped area) compared with the "shared chain" design. Especially, it is up to 30% better than the "shared chain" design in terms of mapped area. In addition, the quick MAIT-merging algorithm is very fast, as shown in Table III. Even when the number of the target probabilities is 100, it takes much less than 1 second to finish.

TABLE III: The performance of the quick MAIT-merging algorithm in terms of number of AND gates when denominators of target probabilities have a large power.

	shared chain		quick MAIT-merging		
#targets	#AND gates	runtime/s	#AND gates	runtime/s	save/%(shared)
10	137	0.03041	93	0.01705	32.12
20	236	0.06655	155	0.03879	34.32
30	319	0.11775	210	0.06445	34.17
40	398	0.16860	264	0.09079	33.67
50	473	0.21182	318	0.12587	32.77
60	540	0.25726	356	0.15048	34.07
70	607	0.31644	412	0.20101	32.13
80	680	0.42420	449	0.22281	33.97
90	749	0.49145	508	0.25828	32.18
100	809	0.54831	540	0.27269	33.25

TABLE IV: The performance of the quick MAIT-merging algorithm in terms of mapped area when denominators of target probabilities have a large power.

	shared chain	quick MAIT-merging	
#targets	mapped area	mapped area	save/%(shared)
10	271	198	26.94
20	463	324	30.02
30	623	436	30.02
40	771	545	29.31
50	908	655	27.86
60	1032	735	28.78
70	1157	850	26.53
80	1285	923	28.17
90	1410	1035	26.60
100	1518	1100	27.54

VII. Conclusion

In this paper, we consider the problem of synthesizing combinational circuits to generate multiple probabilities for the MUX-based stochastic computing architecture. Using the fact that these

target probabilities could have correlation, we proposed two novel algorithms that produce low-cost circuits for generating these probabilities. The essential idea of those two algorithms is to find a good combination of the mincost AND-inverter trees (MAITs) for all the target probabilities and then merge these MAITs to form a low-cost AIG. Experimental results showed that the quick MAIT-merging algorithm is very fast. However, its performance in reducing the cost of the circuit is very satisfying.

ACKNOWLEDGEMENT

This work is supported by a grant from the National Natural Science Foundation of China (NSFC), Project No. 61204042.

REFERENCES

[1] A. Alaghi and J. Hayes, "A spectral transform approach to stochastic circuits," in *International Conference on Computer Design*, 2012, pp. 315–321.
[2] A. Alaghi, C. Li, and J. Hayes, "Stochastic circuits for real-time image-processing applications," in *Design Automation Conference*, 2013, pp. 1–6.
[3] J. Alspector, J. Gannett, S. Haber, M. Parker, and R. Chu, "A VLSI-efficient technique for generating multiple uncorrelated noise sources and its application to stochastic neural networks," *IEEE Transactions on Circuits and Systems*, vol. 38, no. 1, pp. 109–123, 1991.
[4] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Transactions on Computer-Aided Design*, vol. 6, no. 6, pp. 1062–1081, 1987.
[5] B. Brown and H. Card, "Stochastic neural computation I: Computational elements," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 891–905, 2001.
[6] B. Gaines, "Stochastic computing systems," in *Advances in Information Systems Science*. Plenum, 1969, vol. 2, ch. 2, pp. 37–172.
[7] P. Jeavons, D. A. Cohen, and J. Shawe-Taylor, "Generating binary sequences for stochastic computing," *IEEE Transactions on Information Theory*, vol. 40, no. 3, pp. 716–720, 1994.
[8] P. Li, D. Lilja, W. Qian, K. Bazargan, and M. Riedel, "The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic," in *International Conference on Computer-Aided Design*, 2012, pp. 480–487.
[9] P. Li, W. Qian, M. Riedel, K. Bazargan, and D. Lilja, "The synthesis of linear finite state machine-based stochastic computational elements," in *Asia and South Pacific Design Automation Conference*, 2012, pp. 757–762.
[10] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *Design Automation Conference*, 2006, pp. 532–536.
[11] A. Mishchenko *et al.*, "ABC: A system for sequential synthesis and verification," 2007. [Online]. Available: http://www.eecs.berkeley.edu/ alanmi/abc/
[12] W. Qian, X. Li, M. Riedel, K. Bazargan, and D. Lilja, "An architecture for fault-tolerant computation with stochastic logic," *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 93–105, 2011.
[13] W. Qian, M. Riedel, H. Zhou, and J. Bruck, "Transforming probabilities with combinational logic," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 9, pp. 1279–1292, 2011.
[14] S. Tehrani, S. Mannor, and W. Gross, "Fully parallel stochastic LDPC decoders," *IEEE Transactions on Signal Processing*, vol. 56, no. 11, pp. 5692–5703, 2008.
[15] S. Toral, J. Quero, and L. Franquelo, "Stochastic pulse coded arithmetic," in *International Symposium on Circuits and Systems*, vol. 1, 2000, pp. 599–602.
[16] C. Wang and W. Qian, "Optimizing multi-level combinational circuits for generating random bits," in *Asia and South Pacific Design Automation Conference*, 2013, pp. 139–144.