ASPPLN: Accelerated Symbolic Probability Propagation in Logic Network

Weihua Xiao[1], Weikang Qian[1,2*]
[1]University of Michigan-Shanghai Jiao Tong University Joint Institute, Shanghai Jiao Tong University, China
[2]MoE Key Lab of Artificial Intelligence, Shanghai Jiao Tong University, China
{019370910014,qianwk}@sjtu.edu.cn

ABSTRACT

Probability propagation is an important task used in logic network analysis, which propagates signal probabilities from its primary inputs to its primary outputs. It has many applications such as power estimation, reliability analysis, and error analysis for approximate circuits. Existing methods for the task can be divided into two categories: simulation-based and probability-based methods. However, most of them suffer from low accuracy or bad scalability. In this work, we propose ASPPLN, a method for accelerated symbolic probability propagation in logic network, which has a linear complexity with the network size. We first introduce a new definition in a graph called *redundant input* and take advantage of it to simplify the propagation process without losing accuracy. Then, a technique called *symbol limitation* is proposed to limit the complexity of each node's propagation according to the *partial probability significances* of the symbols. The experimental results showed that compared to the existing methods, ASPPLN improves the estimation accuracy of switching activity by up to 24.70%, while it also has a speedup of up to 29×.

KEYWORDS

logic network, symbolic probability propagation, dominator, complexity

1 INTRODUCTION

Probability propagation is an important problem in logic network analysis. Given the signal probabilities at the *primary inputs* (*PIs*) of a logic network, it derives the signal probability of each node in the network by a propagation through the network. It is a key problem in many analysis tasks of digital circuits. For example, the estimation of the dynamic power of a digital circuit mainly relies on the computation of the switching activity for each gate in it, which is the probability of signal toggling at the gate's output [1]; the reliability analysis of a digital circuit depends on the estimation of the probability of the correct function at its *primary outputs* (*POs*) [2]; additionally, computation of each node's probability to be a 1 in a logic network is often needed in the design of approximate circuits for managing the error [3, 4]. Thus, it is crucial to develop an efficient and accurate probability propagation method.

Many different methods have been proposed to propagate the probabilities in a logic network. They can be divided into two major categories: simulation-based and probability-based methods [1]. Simulation-based methods simulate a logic network with random bit vectors at PIs, which are then propagated to the POs. Probabilities of each node in the network can be derived from its propagated bit vector. However, its accuracy heavily depends on the length of the input bit vectors: when a higher accuracy is in demand, longer input bit vectors are required and hence, the method is more time-consuming [5].

On the other hand, the probability-based methods directly propagate the user-specified probabilities at the PIs to the POs through the logic network, which are more efficient [1, 6, 7]. However, signal correlations caused by the reconvergent paths in the circuit are ignored during the direct probability propagation, which is the key drawback of this method and can often lead to a low accuracy [8]. To solve this problem, Krishnaswamy *et al.* proposed a non-symbolic method based on the probabilistic transfer matrix to estimate POs' error probabilities exactly [9]. More generally, a symbolic method was proposed to propagate the symbolic probability through each node in [10], which can totally solve the signal correlation issue. Unfortunately, these exact approaches are intractable for large circuits due to their exponentially growing runtime and memory requirement. Some other works proposed approximation techniques for exact symbolic approach at a trade-off between the runtime and the accuracy [8, 11]. However, most of their runtime is unpredictable due to their dependency on some special structures in logic networks. In addition, the approximate technique in [8] is too aggressive, leading to a great reduction in accuracy.

In this paper, oriented at the symbolic technique for probability propagation, we propose ASPPLN, a method for accelerated symbolic probability propagation in logic network. ASPPLN is based on a basic operation called elimination, which substitutes a symbol in a symbolic function by its probability. For acceleration, we first introduce an accuracy-lossless elimination method by identifying a special set of inputs call redundant inputs. Then, to further control the complexity of each node's symoblic probability propagation, we propose a new metric called *partial probability significance* (PPS) for each symbol appearing in the propagation, and eliminate the symbols with lower PPSs. The complexity of ASPPLN is linear to the network size and independent of the network structure, while the accuracy reduction is small due to the use of PPS. The experimental results showed that compared to the existing methods, ASPPLN improves the estimation accuracy of switching activity by up to 24.70%, while it also has a speedup of up to 29×.

*Corresponding author: Weikang Qian.

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549456

The rest of the paper is organized as follows. Section 2 introduces the preliminaries and the related works. Section 3 presents our proposed methodology. Section 4 shows the experimental results. Finally, Section 5 concludes the paper.

2 PRELIMINARIES AND RELATED WORKS

This section discusses the preliminaries and the related works.

2.1 Probabilities in Logic Network

In the probability propagation of logic networks, there are two types of probabilities to be estimated for each node [10].

The first type is the *static probability*, which is defined as the probability of a node's output to be 1. In the propagation of static probabilities, each node's static probability can be derived by propagating the given static probability of each PI through the logic network. Its applications include reliability analysis and error analysis for approximate computing. Typically, the static probability of each PI is set as 0.5 and regarded as independent.

The second type is the *transition probability*, which represents the probabilities of four transition events at each node, *i.e.*, $0 \rightarrow 0$, $0 \rightarrow 1$, $1 \rightarrow 0$ and $1 \rightarrow 1$, where $x \rightarrow y$ represents a transition from the logic value x to the logic value y. Its typical application is dynamic power estimation. Note that the transition events $0 \rightarrow 0$ and $1 \rightarrow 1$ are two special cases without the change of values, but they are needed in the applications of transition probability. For transition probability propagation, each PI is given with its four transition probabilities and then the transition probabilities of each node are derived after propagation. Each PI's four transition probabilities are often set independently as four 0.25s.

2.2 Exact Symbolic Probability Propagation

In [12], an exact symbolic propagation method for static probability was proposed. Each node n_i is associated with 4 attributes:

(1) The static probability value of the node, denoted as $n_i.Prob$. In the following, we will call it the probability of the node for short.
(2) The symbol representing the static probability of the node, denoted as $n_i.Symb$. In the following, for simplicity, we will call it symbol of the node. We will also refer to the symbol of n_i as s_i.
(3) The global symbolic probability function, denoted as $n_i.gFunc$. It is a symbolic expression on the static probability of the node in terms of the symbols of the PIs of the network. In the following, for simplicity, we will also refer to the global symbolic probability function of n_i as F_i.
(4) The local symbolic probability function, denoted as $n_i.lFunc$. It is a symbolic expression on the static probability of the node in terms of the symbols of the fanins of the node, which can be derived directly according to the logic function of the node. In the following, for simplicity, we will also refer to the local symbolic probability function of n_i as f_i.

The symbolic probability propagation is actually to compute each node's global symbolic probability function. The propagation method relies on an important definition below [8].

Definition 1. *Given a symbolic function $f(s_1, \cdots, s_m)$, its* **super expression** *supexp(f) is defined as a new symbolic function by replacing each term s_i^p $(p > 1)$ with s_i.*

Actually, the existence of a power term of a symbol s_i with degree more than 1 (i.e., s_i^p with $p > 1$) indicates signal correlation, caused by the reconvergent paths from the node corresponding to the symbol s_i [12]. By the *supexp* operation, the effect of the signal correlation is eliminated. Note that there are two kinds of signal correlation, *spatial correlation* and *temporal correlation*. Their difference is that the former does not consider the delay of logic gates during propagation, while the latter does [8]. In this paper, we only focus on the spatial correlation, but it is easy to extend our method to handle the temporal correlation as [8] does.

The key step of the symbolic probability propagation method in [12] is to propagate the global symbolic probability functions from the fanins of a node to its output. To achieve this, for a node n_i with two fanins n_j and n_k, the method first substitutes the global symbolic probability functions of n_j and n_k into the local symbolic probability function of n_i, deriving a new symbolic probability function called *pseudo global symbolic probability function*, and then applies the *supexp* operation to it to obtain the global symbolic probability function of n_i. The step can be expressed as:

$$F_i = supexp(f_i(s_j = F_j, s_k = F_k)). \tag{1}$$

Fig. 1 shows an example of propagating the symbolic probability functions in a logic network, which shows the symbol s_i, the local symbolic probability function f_i, and the global symbolic probability function F_i of each node. Note that each internal node's global symbolic probability function is derived by Eq. (1). For example, the pseudo global symbolic probability function of node 6 is $f_6(s_2, s_4 = F_4) = s_1 s_2^2$ and its final global symbolic probability function is $F_6 = supexp(s_1 s_2^2) = s_1 s_2$. After obtaining each node's global symbolic probability function F_i, the probability of each node can be derived by substituting the probability of each PI into F_i.

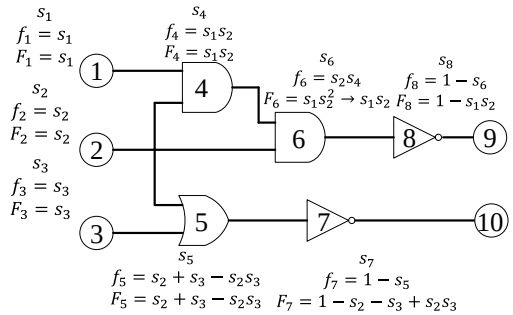


Figure 1: An example of exact symbolic probability propagation.

We note that the above exact symbolic static probability propagation method can be easily extended to handle transition probabilities by allocating each node with four symbols, four local symbolic probability functions, and four global symbolic probability functions for representing the four transition events [8].

2.3 Graph Dominator

A logic network is a directed acyclic graph (DAG) $G = (V, E)$ that implements a logic function, where V and E are the sets of nodes and edges, respectively.[1] In a DAG, a dominator is defined as follows [13].

Definition 2. *A node $v \in V$ is a* **dominator** *of a node $u \in V$ with respect to (w.r.t.) a node set $W \subseteq V$ if v is contained in every path starting from u to a node in W. If W is the PO set, node v is called an* **absolute dominator** *of node u.*

In other words, if a node v is an absolute dominator of another node u, then all the paths starting from u will reconverge at v. For example, node 6 is an absolute dominator of node 1 in Fig. 1.

2.4 Related Works

We describe several previous works tackling the problem of signal correlation during probability propagation for probability-based techniques. In [14], it conducts the probability propagation under the framework of Bayesian network. To tackle the signal correlation, it proposes to split the initial logic network into sub-networks and then transforms each into a structure called *junction tree*. Then, local message passing is applied on each junction tree to finish the probability propagation. However, this kind of technique is only limited to handle the spatial signal correlation [15]. In contrast, symbolic methods can be applied for both temporal and spatial correlations. The work [8] proposes a technique that only conducts symbolic probability propagation in a depth-limited sub-network for each node and approximately simplifies the symbolic expression according to a special structure called *active nodes*. However, the simplification is aggressive, which can lead to a relative large accuracy reduction for some cases. Similarly, the method in [11] constructs a sub-network for each node according to a special structure called *sources of the first reconvergence*. Then, a simulation is applied to each sub-network to derive the probability of each node. However, the complexity of the probability propagation methods in [8] and [11] depends on the special structures, i.e., active nodes and sources of the first reconvergence, causing an unpredictable runtime. Recently, some works take advantage of graph neural network (GNN) to do probability propagation on logic networks, with applications in switching activity estimation [16] and static probability estimation [17], due to the ability of GNN to extract features from graph-structural data. However, existing GNN-based methods focus merely on extracting features from the graph topology, while they often fail to capture the underlying logic functions, thus limiting their accuracy in probability propagation tasks.

3 METHODOLOGY

In this section, we elaborate ASPPLN. Same as many previous works [8, 10], we assume that the input probabilities are independent. Unless otherwise specified, we describe ASPPLN for computing static probability. However, it can be easily adapted to handle transition probability.

[1]We focus on combinational circuit, which is the main part of a sequential circuit and can be represented as a DAG.

3.1 Overview

The exact symbolic probability propagation method described in Section 2.2 is intractable for a large network, since the number of product terms in a global symbolic probability function grows exponentially with the network size. Instead, ASPPLN conducts a *depth-limited* symbolic probability propagation, *i.e.*, computing the global symbolic probability function of each node in a *sub-network*, similar to [8]. A sub-network is built for each node n in the input network by a reverse depth-first search (DFS) from n under a depth limit and then collecting the visited nodes. In the following, we will simply refer to it as the sub-network of node n.

However, even though the symbolic probability propagation is conducted only within the sub-network of a node, the computation of the global symbolic probability functions of some nodes may still be time-consuming when the number of the inputs of a sub-network is large. In ASPPLN, we take advantage of a basic operation called *elimination*, *i.e.*, substituting the probability values of some inputs into a global symbolic probability function to simplify it. In other words, we eliminate some input symbols in the global symbolic probability function of a node instead of preserving them until the propagation reaches the output node of a sub-network. However, eliminating random symbols at random nodes of a sub-network may lead to a relatively large reduction in the final accuracy of the computed probability at the output node. Thus, this calls for a study on a good strategy on symbol elimination. In the following, we propose two methods for symbol elimination. The first one gives an elimination strategy that does not introduce any accuracy loss in a sub-network, which will be described in detail in Section 3.2. However, only applying this method can still be time-consuming. Therefore, we propose another method that eliminates the symbols at the cost of some accuracy loss, which will be described in detail in Section 3.3. The entire ASPPLN flow will be presented in Section 3.4.

3.2 Accuracy-lossless Symbol Elimination Method

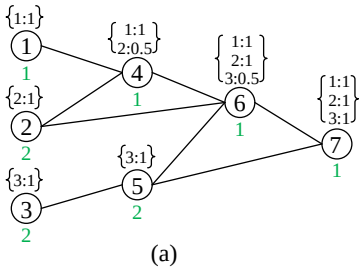
This section presents an accuracy-lossless symbol elimination method. It is applied to the sub-network of each node. Thus, the following discussion is restricted to a sub-network. For a sub-network of node n, we regard that it has a single output, which is n. The basic idea is to find a set of input symbols *w.r.t.* each node in a sub-network such that their removal will not lead to an accuracy loss. Note that in the symbolic probability propagation within a sub-network, the global symbolic probability function of each node in the sub-network is only expressed as a function of the input symbols of the sub-network.

We first give the following definition.

Definition 3. *In a sub-network $G = (V, E)$ of a node n, a node $v \in V$ is a* **final absolute dominator** *of a node $u \in V$, if v is an absolute dominator of u of the sub-network and v has only one path to the output n of G, that is, there is no reconvergent path starting from v.*

For example, consider a sub-network of node 7 shown in Fig. 2(a). Node 4 is a final absolute dominator of node 1.

The following theorem shows that we can exploit final absolute dominator to simplify the global symbolic probability function.



Node	vDom
1	[]
2	[]
3	[]
4	[1]
5	[]
6	[1,2]
7	[1,2,3]

(b)

Figure 2: An example of computing redundant inputs: (a) a sub-network; (b) a table showing all redundant inputs of each node in the sub-network.

Theorem 1. *In a sub-network $G = (V, E)$ of a node n, if a node v is a final absolute dominator of a node u, then the corresponding symbol of node u can be eliminated at node v's global symbolic probability function without introducing error into the symbolic probability propagation within the sub-network.*

Due to the space limit, we only give a sketch of the proof for the above theorem. Since v is a final absolute dominator of u, by Definition 3, v is also an absolute dominator of u. Thus, by Definition 2, there is no path from node u to the output n of the sub-network without containing node v. In other words, if u's symbol is contained in the computed global symbolic probability function of a node w after node v, it must be propagated from node v. Since no reconvergent path from node v exists, there is only one path connecting v to w. Thus, no power term of the form $(u.Symb)^p$ with $p > 1$ is included in the global symbolic probability function of node w. Thus, no accuracy reduction can happen after u's symbol is eliminated from v's global symbolic probability function.

We further define a special input of the sub-network, called *redundant input*, for each node.

Definition 4. *In a sub-network, a* **redundant input** *of a node v is an input of the sub-network such that node v is a final absolute dominator of the input.*

For example, consider the sub-network shown in Fig. 2(a). Node 1, an input of the sub-network, is a redundant input of node 4.

By Theorem 1 and Definition 4, for any node v in a sub-network, we can eliminate the symbols of all the redundant inputs of node v from the global symbolic probability function of v, which simplifies the symbolic propagation without accuracy reduction.

Next, we propose a network flow-based method to obtain all redundant inputs for each node in a sub-network. It is shown in Algorithm 1. Its basic idea is to propagate flows from the inputs of the sub-network to its output and check the amount of flow gathered at each node to determine all redundant inputs of it.

The input of the algorithm is a sub-network *Ntk* with a single output *Ntk.Root*. Line 3 initializes 3 data members of each node v:

(1) $v.nPath$: It represents the number of paths from node v to the output node.
(2) $v.Flow$: It is a map where each key k is a *source input* of v, which is defined as an input of *Ntk* that has a path to node v, and the value associated with the key k, denoted as $v.Flow[k]$, is the propagated flow from the input k to node v.

Algorithm 1: The procedure *GetRedunInputs* for obtaining all the redundant inputs for each node in a sub-network.

Input: A network *Ntk* with a single output *Ntk.Root*.
Output: A new single-output network updated with all redundant inputs of each node.

```
1  // initialize
2  foreach node v of Ntk do
3  |  v.nPath ← 0; v.Flow ← {}; v.vDom ← [];
4  // derive each node's number of paths to the output node
5  Ntk.Root.nPath ← 1;
6  foreach node v of Ntk except Ntk.Root in reverse topological order do
7  |  foreach fanout node w of v do
8  |  |  v.nPath ← w.nPath + v.nPath;
9  // obtain each node's redundant inputs vDom
10 foreach PI i of Ntk do i.Flow[i] ← 1;
11 foreach node v of Ntk in topological order do
12 |  // update v's Flow
13 |  foreach fanin node fi of v do
14 |  |  foreach key k of fi.Flow do
15 |  |  |  if key k is not in v.Flow then
16 |  |  |  |  add key k into v.Flow; v.Flow[k] ← 0;
17 |  |  |  else v.Flow[k] ← v.Flow[k] + fi.Flow[k]/fi.nFo;
18 |  // update v's redundant inputs
19 |  foreach key k of v.Flow do
20 |  |  if v.Flow[k] = 1 and v.nPath = 1 then
21 |  |  |  v.vDom.push(k);
22 return Ntk;
```

(3) $v.vDom$: It is a vector storing all redundant inputs of v.

Lines 5–8 process each node v in a reverse topological order to update its member *nPath* by adding up the members *nPath* of its fanout nodes. An example of this step is shown in Fig. 2(a), where the member *nPath* of each node is shown by the green number below it. For example, the member *nPath* of node 5 is obtained by summing those of nodes 6 and 7, which gives 2.

In the remaining part of the algorithm, it obtains all redundant inputs *vDom* of each node by working on the member *Flow*. Initially, Line 10 assigns each input of the sub-network with an input flow of 1. Then, for each node of the sub-network in a topological order, its member *Flow* is updated. Lines 13–17 show the update rule for a node v. The rule basically sets the set of keys in $v.Flow$ as the union of the sets of source inputs of all fanin nodes of v. For each key k in $v.Flow$, the associated value $v.Flow[k]$ is given by the sum of the flows propagated from the input k through each fanin fi of v. Note that the flow propagated through a fanin fi to v is calculated as $fi.Flow[k]$ divided by the fanout number of fi, denoted as $fi.nFo$. This means that an input flow to a node is evenly distributed over its fanouts.

Example 1. *In Fig. 2(a), the obtained member Flow of each node is shown above it. For example, node 6 has three source inputs, nodes 1, 2, and 3, which propagate flows of 1, 1, and 0.5 to node 6, respectively. Among them, the flow from node 2 is calculated as the sum of the flows from node 2 to node 6 via nodes 2 and 4 divided by their corresponding numbers of fanouts, i.e.,* $1/2 + 0.5/1 = 1$.

After the member *Flow* of node v is obtained, Line 19 checks all source inputs of v. If a source input propagates a flow of 1 to v,

then v is actually an absolute dominator of the input. The reason is that the flow of 1 guarantees that all paths starting from the source input reconverge at node v. Furthermore, if node v has only one path to the output of the sub-network, the source input is a redundant input of node v, which is then added into $v.vDom$ (see Line 21). Fig. 2(b) shows the member $vDom$ of each node.

3.3 Accuracy-lossy Symbol Elimination Method

Although the redundant inputs can be exploited to simplify the symbolic propagation, they may not exist for some nodes due to the structure constraint of a sub-network. For example, as shown in Fig. 2(b), node 5 in Fig. 2(a) has no redundant inputs, since it has more than one path to the output of the sub-network. For those nodes without redundant inputs, we have no chance to simplify their global symbolic probability functions, which may lead to a high complexity of symbolic propagation for later nodes.

To better control the runtime of the computation of each node's global symbolic probability function, we first study the root reason for the high computation complexity. The symbolic propagation at each node actually consists of some multiplications and additions over symbolic functions, whose computation complexity is proportional to the number of product terms in the obtained symbolic function. Since in our symbolic propagation, each node's global symbolic probability function has no power terms with degree more than 1, thus, for a symbolic function with n symbols, it has 2^n product terms in the worst case.

Given the above observation, we propose a basic acceleration technique called *symbol limitation*, which is an accuracy-lossy symbol elimination method. Specifically, when computing the global symbolic probability function of a node, this technique eliminates some symbols in the global symbolic probability functions of its fanins before substitution, such that the number of symbols in the symbolic function after substitution is no more than a given bound. For better selecting symbols to be eliminated, we propose a metric called *partial probability significance* (PPS). It is defined *w.r.t.* a symbol s and a node v in a sub-network. It measures the influence to the probability of the output node of the sub-network when symbol s is eliminated from the global symbolic probability functions of the fanins of node v before substitution. In some sense, PPS can capture the influence to the accuracy of eliminating a symbol in a sub-network.

To compute the PPS *w.r.t.* a symbol s and a node v in a sub-network, we conduct two rounds of symbolic propagation in the sub-network. However, instead of preserving all input symbols of the sub-network at each node, we only preserve symbol s during the propagation and substitute symbols of the other inputs with their corresponding probabilities. We call this special propagation a *partial symbolic propagation*. In the first round, we do a partial symbolic propagation with regard to symbol s, after which the symbolic function of the sub-network's output node is expressed only in symbol s. Then, we eliminate symbol s from the output node's symbolic function and derive a value denoted as PPS_1. In the second round, another partial symbolic propagation is conducted in the sub-network with regard to symbol s, in which we substitute the probability of symbol s into the symbolic functions of the fanins of node v before propagating them to v. Similarly, we derive another value by eliminating symbol s from the output node's symbolic function, denoted as PPS_0. Finally, the PPS *w.r.t.* symbol s and node v is calculated as $|PPS_1 - PPS_0|$.

Example 2. *Fig. 3(a) (resp. Fig. 3(b)) shows the partial symbolic propagation for computing PPS_1 (resp. PPS_0) w.r.t. symbol s_2 and node 6 in the sub-network shown in Fig. 2(a). Table 1 lists the symbol and local symbolic probability function of each node and the probability of each input. Initially, the symbolic functions of all input nodes other than node 2 is set as their corresponding probabilities. Each node's symbolic function is derived as an expression of symbol s_2 or a constant after the partial symbolic propagation, which is shown above each node. For example, for computing the symbolic function of node 6 in Fig. 3(b), we first substitute the probability of node 2 into the symbolic functions of nodes 4, 2, and 5, which gives $0.2 + 0.6 \cdot 0.5 = 0.5$, 0.5, and 0.5, respectively. Then, these values are substituted into node 6's local symbolic probability function to derive its symbolic function, which gives $0.5 \cdot 0.5 \cdot 0.5 = 0.125$. Then, PPS_1 (resp. PPS_0) is derived by substituting the probability of node 2 into the symbolic function of node 7 shown in Fig. 3(a) (resp. Fig. 3(b)), which gives $0.5+0.2 \cdot 0.5 = 0.6$ (resp. 0.5625). Finally, the PPS w.r.t. symbol s_2 and node 6 equals $|0.6 - 0.5625| = 0.0375$.*

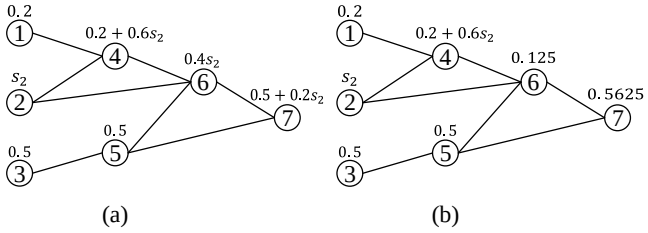


Figure 3: An example of the partial symbolic propagation for computing (a) PPS_1 and (b) PPS_0 *w.r.t.* symbol s_2 and node 6.

Table 1: The symbol and local symbolic probability function of each node, and the probability of each input in the sub-network of Fig. 2(a).

Node	Symbol	Local symbolic probability function	Probability
1	s_1	s_1	0.2
2	s_2	s_2	0.5
3	s_3	s_3	0.5
4	s_4	$s_1 + s_2 - 2s_1s_2$	–
5	s_5	$1 - s_3$	–
6	s_6	$s_2s_4s_5$	–
7	s_7	$s_5 + s_6 - s_5s_6$	–

Now, we present the details of the symbol limitation method. We set the upper bound of the number of symbols kept in the global symbolic probability function of each node as K. For each node v, we calculate the PPS *w.r.t.* node v and each symbol existing in the global symbolic probability functions of the fanins of v. Note that by the method to calculate PPS, we can see that the larger the PPS is, the more important is its contribution to the accuracy

of the probability of the output node. Thus, we keep the top K symbols with the *largest* PPS values and eliminate the others from the global symbolic probability function of each fanin of v. Finally, the updated global symbolic probability function of each fanin is substituted into the local symbolic probability function of v to get its global symbolic probability function.

3.4 Flow of ASPPLN

Algorithm 2: The flow of ASPPLN.

```
Input: A logic network Ntk, probabilities of the PIs, the depth limit
       L, and the maximum number of symbols K.
Output: A logic network updated with each node's probability.
1  // initialize
2  foreach node n of Ntk do initialize n.Symb and n.lFunc;
3  foreach PI Pi of Ntk do initialize Pi.Prob;
4  // derive each node's probability
5  foreach node n of Ntk in topological order do
6      SubNtk ← DFS(Ntk, n, L);
7      // initialize SubNtk's PIs
8      foreach input i of SubNtk do i.gFunc ← i.Symb;
9      // initialize non-input nodes of SubNtk
10     foreach non-input node v of SubNtk do v.gFunc ← v.lFunc;
11     GetRedunInputs(SubNtk);
12     // do symbolic probability propagation in SubNtk
13     foreach node v of SubNtk in topological order do
14         // do symbol limitation
15         SymbLimit(v, K);
16         // eliminate symbols according to vDom
17         foreach node d of v.vDom do
18             v.gFunc.Sub(d.Symb, d.Prob);
19     // compute the output node's probability
20     foreach symbol symb in SymbSet(n.gFunc) do
21         SrcNode ← getNode(symb);
22         n.Prob ← n.gFunc.Sub(SrcNode.Symb, SrcNode.Prob);
23     Normalize(n.Prob);
24 return Ntk;
```

We show the flow of ASPPLN in Algorithm 2. Its inputs include a logic network *Ntk*, the probabilities of the PIs, the depth limit L for constructing the sub-network of each node for symbolic propagation, the maximum number K of preserved symbols at each node during symbolic propagation.

Line 2 initializes each node's symbol and local symbolic probability function according to its logic function. Line 3 allocates the given probability to each PI. In Lines 5–23, it computes each node's probability in a topological order of *Ntk*. For each node n in *Ntk*, a single-output sub-network *SubNtk* is constructed through a reverse DFS from node n with a depth limit of L (Line 6). Then, a symbolic propagation is performed within the sub-network *SubNtk* (Lines 8–18).

Before propagating symbolic functions in *SubNtk*, Line 8 initializes the global symbolic probability function *gFunc* of each input of the sub-network as its symbol and Line 10 initializes the global symbolic probability function of each remaining node as its local symbolic probability function *lFunc*. Then, Line 11 obtains the redundant inputs for each node in *SubNtk* by Algorithm 1. Next, we do symbolic propagation for each node v of *SubNtk* in a topological order. Line 15 does the symbol limitation for node v under the upper bound K as described in Section 3.3. Moreover, the redundant inputs are exploited to simplify its symbolic function (Lines 17–18). Note that in this step, as shown in Line 18, we substitute the symbol of each redundant input d in the global symbolic probability function of v by the probability of d, which has already been derived according to the topological order. After finishing the symbolic propagation within the sub-network *SubNtk*, we substitute the probabilities of the remaining symbols into the the global symbolic probability function of the output node n of *SubNtk* to get the probability of n (Lines 20–22). However, in the case of computing transition probabilities, the sum of the four probabilities may not equal 1 at some nodes. To obtain legal transition probabilities, Line 23 normalizes the four probabilities at each node after propagating in its sub-network so that their sum equals 1.

Next, we analyze the computation complexity of Algorithm 2. Assume that the maximum number of fanins in a network is I. Thus, the sub-network constructed at each node has $O(I^L)$ nodes and $O(I^L)$ inputs. Clearly, the symbol limitation process at Line 15 dominates the computation complexity of Algorithm 2. Thus, we focus on its complexity analysis. Due to the upper bound K of the number of preserved symbols at each node and the maximum fanin number I, the complexity of the symbolic computation excluding PPS computation for each node is $O(2^{KI})$. Since the PPS computation preserves only one symbol during the partial symbolic propagation process, the complexity of the PPS computation *w.r.t.* one symbol and one node is proportional to the size of the sub-network, which is $O(I^L)$. The complexity of the PPS computation *w.r.t.* one node and all the input symbols of the network is $O(I^L) \cdot O(I^L) = O(I^{2L})$. Thus, the complexity for a single call of the function *SymbLimit* is $O(2^{KI} + I^{2L})$. Consequently, the complexity of Algorithm 2 can be derived as $O((2^{KI} + I^{2L})I^L|V|)$, where $|V|$ is the number of nodes in the network. Given that I, L, and K are constants, ASPPLN has a linear complexity in terms of the size of the whole network.

4 EXPERIMENTAL RESULTS

This section presents the experimental results of ASPPLN.

4.1 Experimental Setup

We implement ASPPLN in C++. We use ABC [18] to parse the input circuits. All the experiments are conducted on an 8-core AMD 5800U processor running at 1.9GHz with 16GB RAM. A set of circuits from the IWLS benchmark suite [19] and the EPFL benchamrk suite [20] are used. They are listed in Table 2 with their corresponding numbers of PIs (#PI), numbers of POs (#PO), numbers of nodes (#Nodes), and numbers of levels (#Levels). We consider two estimation targets, static probability and switching activity. We implement a Monte-Carlo (MC) simulation method, accelerated by the bit-parallel technique, to derive each node's actual static probability or switching activity. For each benchmark, at most 2^{26} random input patterns are generated for the bit-parallel MC simulation, after which we observe that the static probability or switching activity values can converge. Two metrics are used to measure the accuracy of a method for a circuit: maximum error and mean error, which are the maximum error and the mean error, respectively, of

the method in estimating static probability or switching activity compared to the MC simulation over all the nodes in the circuit.

Table 2: Benchmarks used in the experiments.

Circuit	#PI	#PO	#Nodes	#Levels	Circuit	#PI	#PO	#Nodes	#Levels
C432	36	7	362	30	i6	138	67	545	5
C499	41	32	597	27	term1	34	10	643	23
C880	60	26	705	41	z4ml	7	4	81	5
C1355	41	32	981	45	Adder	256	129	1020	255
C3540	50	22	1742	67	Max	512	130	2865	287
C7552	207	108	4408	68	Bar	135	128	3336	12
cm151a	12	2	63	14	Sine	24	25	5416	225
cm163a	16	5	84	12	Square	64	128	18486	250
c8	28	18	253	9	Multiplier	128	128	27062	274
dalu	75	16	2625	60	Log2	32	32	32060	444

4.2 Parameter Tuning

In this section, we tune the two parameters L and K in ASPPLN to study their influence to the accuracy and runtime of ASPPLN, such that we can better set their values. The ranges of L and K are $L \in \{3, 4, 5, 6\}$ and $K = 2k$ with $k \in \{0, 1, \cdots, 7\}$, respectively. Consequently, there are 32 parameter combinations in total. We test all of them and apply ASPPLN to compute the static probabilities for four circuits, *Adder*, *Max*, *Bar*, and *Sine*. We assign each PI with the static probability of 0.5. However, we remark that ASPPLN can be applied to handle arbitrary input static probabilities. We show the average of the mean errors of the four circuits under each parameter combination in Fig. 4. Moreover, the average of their runtime under each parameter combination is shown in Fig. 5, where the *Time* axis is in the logarithmic scale.

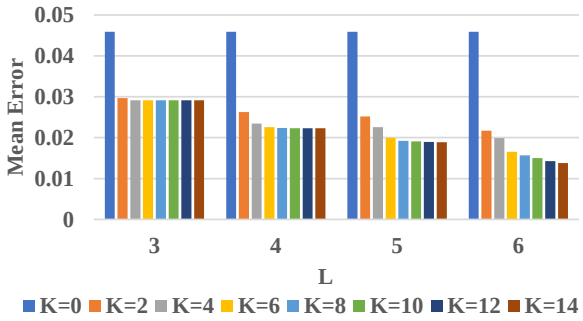


Figure 4: The mean error under different combinations of L and K.

As shown in Fig. 4, the mean error reduces with the parameter L, which is expected. Under the same L, the mean error also shows a decreasing trend with the parameter K. However, for smaller Ls, further increasing K beyond a value does not improve the accuracy. The reason is that when K is large enough, it is larger than the maximum number of symbols during propagation in each sub-network. In other words, no symbols will be eliminated under this value of K and thus, no improvement of accuracy happens for an even larger K. In Fig. 5, there is a clear increasing trend of the

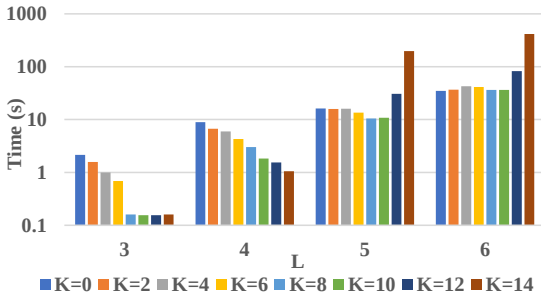


Figure 5: The runtime under different combinations of L and K.

runtime with L. Under the same L, the runtime reduces with K when L is relatively small. However, when L is large, the runtime reduces with K initially, but it increases with K later. The main reason is that the increase of K leads to a longer runtime of symbolic propagation excluding the PPS computation, but meanwhile, it also reduces the number of times needed to compute PPS. Such a joint effect may lead to the overall runtime decrease with K. From this experimental study, we also observe that ASPPLN with $L = 3$ and $K = 10$ can lead to a short runtime and a relatively high accuracy. Thus, this combination of L and K is used in the following experiments.

Table 3: Comparison of ASPPLN with DeepGate [17] on static probability estimation.

Circuit	DeepGate			ASPPLN		
	Max	Mean	Runtime (s)	Max	Mean	Runtime (s)
Adder	0.1104	**0.0308**	4.59	**0.0722**	0.0402	**0.05**
Max	**0.2908**	0.0371	6.98	0.3496	**0.0308**	**0.17**
Bar	**0.1058**	**0.0518**	3.07	0.2258	0.0562	**0.17**
Sine	0.6807	0.1259	9.50	**0.5383**	**0.0506**	**0.39**
Square	0.6366	0.1059	98.76	**0.3037**	**0.043**	**1.05**
Multiplier	**0.3315**	0.0744	109.30	0.4212	**0.0448**	**1.19**
Log2	**0.7395**	0.1077	139.57	0.7885	**0.0518**	**1.50**
Average	0.4136	0.0762	53.11	**0.3856**	**0.0453**	**0.65**

4.3 Performance of ASPPLN in Static Probability Estimation

In this section, we study the performance of ASPPLN in static probability estimation. We compare it with a state-of-the-art GNN-based method, DeepGate [17]. We assign each PI with the static probability of 0.5. As DeepGate only supports AND-inverter graph (AIG), we only do comparison over the last seven circuits in Table 2 from the EPFL benchmark suite, whose initial formats are AIG.

Table 3 compres ASPPLN and DeepGate in terms of maximum error (Max), mean error (Mean), and runtime. The last row of the table shows the average of the seven circuits. For each circuit, we highlight the smallest maximum error, mean error, and runtime in bold. We can see that although the state-of-the-art GNN-based method, DeepGate, achieves smaller maximum error and mean error for a few circuits, on average, ASPPLN outperforms DeepGate on both maximum error and mean error with a reduction of 6.77%

Table 4: Comparison of ASPPLN with two prior methods and bit-parallel MC simulation on switching activity estimation.

Circuit	[8]			[11]			ASPPLN			Bit-Parallel MC
	Max	Mean	Runtime (s)	Max	Mean	Runtime (s)	Max	Mean	Runtime (s)	Runtime (s)
C432	0.434	0.105	436.86	0.238	**0.0404**	4.58	**0.226**	0.0456	**2.81**	3.31
C499	0.0992	0.006	44.77	0.083	0.013	6.56	**0.0342**	**0.0044**	**3.98**	12.72
C880	0.0656	0.0118	1.20	0.1142	0.0202	10.69	**0.0656**	**0.0118**	**0.83**	15.06
C1355	**0.1156**	0.0376	468.02	0.148	**0.0198**	11.55	0.142	0.031	**2.97**	19.38
C3540	N/A	N/A	N/A	**0.2282**	0.0258	13.11	0.244	**0.0226**	**12.64**	12.95
C7552	N/A	N/A	N/A	0.46	0.0364	78.69	**0.41**	**0.0338**	**12.31**	90.23
cm151a	0.0898	0.0188	0.05	**0.0698**	0.0202	0.28	0.0898	**0.0186**	0.13	**0.03**
cm163a	0.1636	**0.013**	0.20	**0.1058**	0.0224	0.41	0.1332	0.015	0.17	**0.03**
c8	0.23	0.0226	369.77	0.0872	0.0184	7.70	**0.0758**	**0.0166**	**1.73**	4.00
dalu	0.39	**0.0402**	**2.25**	0.424	0.0688	54.58	**0.2896**	0.042	5.83	103.59
i6	0.1262	0.0124	9.16	0.063	0.0136	**3.16**	**0.0348**	**0.0056**	3.31	24.94
term1	0.328	0.0332	1162.80	0.31	0.0274	4.77	**0.258**	**0.0244**	**3.73**	27.06
z4ml	0.1984	**0.0424**	1.98	**0.1578**	0.0462	0.92	0.1952	0.0446	0.92	**0.02**
Adder	0.04	**0.0124**	5.23	0.076	0.0152	158.17	**0.04**	0.0126	**2.13**	46.20
Max	**0.2448**	0.0184	28.39	0.248	0.017	511.86	0.2474	**0.017**	**9.22**	111.09
Bar	0.1136	0.043	63.83	**0.0968**	**0.0182**	162.47	0.1136	0.0414	**10.33**	107.73
Sine	0.4794	0.0568	565.48	0.5126	0.0608	209.44	**0.3922**	**0.0276**	**36.50**	203.69
Square	0.3274	0.0334	980.81	0.4232	0.051	341.78	**0.3038**	**0.0322**	**71.70**	640.02
Multiplier	**0.233**	0.0296	1378.52	0.2638	0.0288	743.66	0.2358	**0.0288**	**114.47**	2175.05
Log2	**0.4978**	0.0618	7060.38	0.5416	0.0632	1163.08	0.4998	**0.03**	**161.38**	2409.34
Average	0.2320	0.0332	698.87	0.2202	0.0314	188.65	**0.1876**	**0.0250**	**24.01**	327.96

and 40.55%, respectively. Meanwhile, ASPPLN is 82× faster than DeepGate.

4.4 Performance of ASPPLN in Switching Activity Estimation

In this section, we study the performance of ASPPLN in switching activity estimation. We compare it with two existing methods [8, 11]. For switching probability estimation, which is the key task of the dynamic power estimation, we derive each node's four transition probabilities *w.r.t.* $0 \rightarrow 0$, $0 \rightarrow 1$, $1 \rightarrow 0$, and $1 \rightarrow 1$ transitions. The switching activity equals the sum of the probabilities for $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions. Each PI is assigned with the probability of 0.25 for each of the four transition cases, which is common in dynamic power estimation. However, we remark that ASPPLN can be applied to handle any input transition probabilities. For the bit-parallel MC simulation, its random input patterns are also generated according to the assigned transition probabilities. In the implementation of the method in [11], the sub-network of each node is simulated according to each node's input transition probabilities.

Table 4 compares ASPPLN with the methods in [8] and [11] in terms of maximum error (Max), mean error (Mean), and runtime. It also lists the runtime of the bit-parallel MC simulation. Note that *N/As* of [8] for *C3540* and *C7552* is because the runtime is too long, which is because some nodes in the two circuits have a large number of fanins, leading to a large number of symbols at these nodes during the propagation process of [8]. The last row of the table shows the average of all results over all the circuits except *C3540* and *C7552*. For each circuit, we highlight the smallest maximum error, mean error, and runtime in bold. We can see that ASPPLN has a smaller maximum error and mean error than the methods in [8] and [11]. The reason is that our PPS-based symbol limitation technique can lead to a smaller accuracy reduction. On average, ASPPLN has a relative 19.14% and 14.80% reduction in maximum error over the methods in [8] and [11], respectively. ASPPLN also has a relative 24.70% and 20.38% reduction in mean error over these two methods, respectively. Moreover, the comparison in runtime shows the efficiency of ASPPLN: it is on average 29.1× and 7.9× faster than the methods in [8] and [11], respectively. It is also 13.7× faster than the bit-parallel MC simulation method.

5 CONCLUSION

In this work, we propose ASPPLN, a novel approach for accelerating symbolic probability propagation in a logic network. It consists of two acceleration techniques: (i) symbolic function simplification by redundant inputs and (ii) symbol limitation according to PPS. ASPPLN has a linear complexity with the size of the logic network. It outperforms several existing works on symbolic probability propagation in terms of runtime, while offering a better accuracy in probability estimation.

ACKNOWLEDGMENT

This work is supported by the National Key R&D Program of China under grant number 2021ZD0114701. The authors thank Mr. Min Li, Mr. Zhengyuan Shi, and Prof. Qiang Xu from the Chinese University of Hong Kong for the help on the experiments on DeepGate.

REFERENCES

[1] F. N. Najm. A survey of power estimation techniques in VLSI circuits. *IEEE Transactions on Very Large Scale Integration Systems*, 2(4):446–455, 1994.
[2] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[3] Z. Zhang et al. Optimal slope ranking: An approximate computing approach for circuit pruning. In *ISCAS*, pages 1–4, 2018.
[4] I. Scarabottolo et al. Partition and propagate: An error derivation algorithm for the design of approximate circuits. In *DAC*, pages 1–6, 2019.
[5] J. Han et al. A stochastic computational approach for accurate and efficient reliability evaluation. *IEEE Transactions on Computers*, 63(6):1336–1350, 2014.
[6] N. Mohyuddin, E. Pakbaznia, and M. Pedram. Probabilistic error propagation in logic circuits using the Boolean difference calculus. In *ICCD*, pages 7–13, 2008.
[7] J. Echavarria et al. Probabilistic error propagation through approximated Boolean networks. In *DAC*, pages 1–6, 2020.
[8] J. Costa et al. Power estimation using probability polynomials. *Design Automation for Embedded Systems*, 9(1):19–52, 2004.
[9] S. Krishnaswamy and other. Accurate reliability evaluation and enhancement via probabilistic transfer matrices. In *DATE*, pages 282–287, 2005.
[10] J. Monteiro et al. Estimation of average switching activity in combinational logic circuits using symbolic simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(1):121–127, 1997.
[11] S. Sivaswamy, K. Bazargan, and M. Riedel. Estimation and optimization of reliability of noisy digital circuits. In *ISQED*, pages 213–219, 2009.
[12] K. P. Parker and E. J. McCluskey. Probabilistic treatment of general combinational networks. *IEEE Transactions on Computers*, C-24(6):668–670, 1975.
[13] L. Thomas and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.
[14] S. Bhanja and N. Ranganathan. Dependency preserving probabilistic modeling of switching activity using Bayesian networks. In *DAC*, pages 209–214, 2001.
[15] L. Wan and D. Chen. Analysis of digital circuit dynamic behavior with timed ternary decision diagrams for better-than-worst-case design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(5):662–675, 2012.
[16] Y. Zhang, H. Ren, and B. Khailany. GRANNITE: Graph Neural Network Inference for Transferable Power Estimation. In *DAC*, pages 1–6, 2020.
[17] M. Li et al. DeepGate: Learning neural representations of logic gates. In *DAC*, pages 1–7, 2022.
[18] A. Mishchenko et al. ABC: A system for sequential synthesis and verification. http://people.eecs.berkeley.edu/~alanmi/abc/, 2022.
[19] K. McElvain. IWLS'93 benchmark set: Version 4.0. Technical report, 1993.
[20] L. Amarú, P.-E. Gaillardon, and G. D. Micheli. The EPFL combinational benchmark suite. In *IWLS*, pages 1–5, 2015.