VACSEM: Verifying Average Errors in Approximate Circuits Using Simulation-Enhanced Model Counting

Chang Meng[1], Hanyu Wang[2], Yuqi Mai[3], Weikang Qian[3,4], and Giovanni De Micheli[1]
[1]Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
[2]Department of Information Technology and Electrical Engineering, ETH Zurich, Zurich, Switzerland
[3]University of Michigan-SJTU Joint Institute and [4]MoE Key Lab of AI, SJTU, Shanghai, China
Emails: chang.meng@epfl.ch, hanywang@student.ethz.ch, {yq-mai, qianwk}@sjtu.edu.cn, giovanni.demicheli@epfl.ch

***Abstract*—Approximate computing is an effective computing paradigm to reduce area, delay, and power for error-tolerant applications. Average error is a widely-used metric for approximate circuits, measuring the deviation between the outputs of exact and approximate circuits. This paper proposes VACSEM, a formal method to verify average errors in approximate circuits using simulation-enhanced model counting. VACSEM leverages circuit structure and logic simulation to speed up verification. Experimental results show that VACSEM is on average 35× faster than the state-of-the-art method.**

***Index Terms*—formal verification, average error, simulation, model counting**

I. Introduction

Approximate computing [1], [2] is a popular low-power design paradigm for error-tolerant applications, such as image processing, machine learning, and data mining. It trades off accuracy to reduce circuit area, delay, and power. When working with approximate circuits, *error metrics* are essential to evaluate the degree of approximation.

One widely-used error metric is the *average error*, which calculates the average *deviation* between the outputs of exact and approximate circuits. For instance, *error rate* (*ER*) measures the percentage of input patterns with erroneous output patterns. ER is useful for evaluating the accuracy of classifiers, control circuits, *etc*. *Mean error distance* (*MED*) quantifies the average absolute difference between the outputs of exact and approximate circuits. It is typically applied to assess the accuracy of arithmetic circuits like adders and multipliers.

Despite the wide usage of average errors, formally verifying them remains challenging and time-consuming due to the need to compute exact values for all possible input patterns of a circuit. While exhaustive logic simulation is a straightforward method for verification, it is impractical for larger circuits with exponentially growing number of input patterns. To address this, recent techniques rely on *decision diagrams* (*DDs*) [3]–[6]. DD-based methods typically involve creating an *approximation miter* to represent the deviation between exact and approximate circuits, converting this miter into DDs, and then calculating the average error based on these DDs. For example, in [3], Venkatesan *et al.* introduced a single-output approximation miter, where the single output signal indicates whether the deviation exceeds a given threshold. This miter is then transformed into a *binary decision diagram* (*BDD*) to compute the probability of the error exceeding the threshold. By varying the threshold and constructing a series of BDDs, they derive the cumulative distribution function of the deviation, and consequently, the average error. In [4], Wu and Qian proposed a multiple-output approximation miter, where the n outputs directly represent an n-bit deviation in binary. This miter is converted into a multiple-root BDD, and the signal probability of each output is calculated, with the results accumulated to obtain the average error. In [5], Mrazek improved existing BDD-based methods and proposed efficient techniques for formally verifying MED. In [6], Froehlich *et al.* applied *algebraic decision diagrams* (*ADDs*) for error verification. They suggest determining the remainder of an approximate circuit through symbolic computer algebra, constructing an ADD to represent the remainder, and computing the average error with an ADD traversal algorithm. Although DD-based methods sometimes can handle larger circuits than exhaustive simulation, they still suffer from limited scalability [3]–[6].

An alternative approach to verifying average errors is through *model counting*, also known as *#SAT* [7]. Given a Boolean formula F, model counting computes the number of variable assignments making F TRUE. *State-of-the-art* (*SOTA*) model counters, such as [8]–[11], are designed for general applications and lack specific optimization for circuit verification. These solvers operate on the *conjunctive normal form* (*CNF*) representation [12] of Boolean functions and do not take advantage of circuit structure to enhance their efficiency. To overcome this limitation, we integrate circuit simulation into the model counter and present *VACSEM*, a formal approach for verifying average errors in approximate circuits using simulation-enhanced model counting. Our main contributions are summarized as follows:

- We leverage the circuit structure as prior knowledge for the #SAT solver and customize an efficient circuit simulator for #SAT solving.
- We analyze and identify scenarios where logic simulation is faster than traditional #SAT solvers. Based on this analysis, we design a dynamic controller that determines when to activate circuit simulation in our #SAT solver.
- Experimental results demonstrate that VACSEM is 35× faster than the SOTA method and dramatically enhances scalability. It enables rapid verification of 128-bit adders within a second and 16-bit multipliers in minutes.

The VACSEM code is open-source and available at XXX (withheld for blind review purposes).

The rest of the paper is organized as follows. Section II introduces the preliminaries. Section III outlines our motivation

and key idea. Section IV provides a detailed description of the VACSEM methodology. Section V presents the experimental results. Finally, Section VI concludes the paper.

II. Preliminaries

This section introduces the preliminaries to VACSEM.

A. Average Errors

This paper focuses on formally verifying average errors under a uniform distribution. Let $\vec{y} : \mathbb{B}^I \rightarrow \mathbb{B}^O$ and $\vec{y'} : \mathbb{B}^I \rightarrow \mathbb{B}^O$ be the multiple-output Boolean functions of an exact and an approximate circuit, respectively. The average error quantifies the average deviation between $\vec{y}$ and $\vec{y'}$:

$$\text{Average error} = \frac{1}{2^I} \sum_{\vec{x} \in \mathbb{B}^I} F(\vec{y}(\vec{x}), \vec{y'}(\vec{x})), \tag{1}$$

where $F(\vec{y}, \vec{y'})$ is referred to as the *deviation function*.

For example, ER is the probability of an input pattern producing a wrong output for the approximate circuit. Its deviation function is

$$F_{ER}(\vec{y}, \vec{y'}) = \begin{cases} 0, & \text{if } \vec{y} = \vec{y'}, \\ 1, & \text{if } \vec{y} \neq \vec{y'}. \end{cases} \tag{2}$$

Hence, ER can be computed by $ER = \frac{\#SAT(F_{ER})}{2^I}$, where $\#SAT(F_{ER})$ denotes the number of input patterns on $\vec{x}$ that make F_{ER} TRUE, and I is the number of inputs. Computing ER involves solving a #SAT problem, $\#SAT(F_{ER})$.

Another example is MED, measuring the average absolute error between the outputs of the exact and approximate circuits. Its deviation function represents the absolute error as $F_{MED}(\vec{y}, \vec{y'}) = |int(\vec{y}) - int(\vec{y'})|$, where the function $int(\vec{v})$ returns the integer encoded by the binary vector $\vec{v}$. Since both $\vec{y}$ and $\vec{y'}$ have O bits, F_{MED} can be represented with O bits, denoted as $f_1, f_2, \ldots, f_O$. This paper considers a typical binary encoding of F_{MED}:

$$F_{MED} = \sum_{j=1}^{O} 2^{j-1} \cdot f_j. \tag{3}$$

MED, viewed as the expectation of the deviation function F_{MED} ($\mathbb{E}[F_{MED}]$), can be computed as follows [4], [13]:

$$\begin{aligned} \text{MED} = \mathbb{E}[F_{MED}] &= \mathbb{E}\left[\sum_{j=1}^{O} 2^{j-1} \cdot f_j\right] \\ &= \sum_{j=1}^{O} 2^{j-1} \cdot \mathbb{E}[f_j] = \sum_{j=1}^{O} 2^{j-1} \cdot \frac{1}{2^I} \cdot \#SAT(f_i), \end{aligned} \tag{4}$$

where $\#SAT(f_i)$ denotes the number of input patterns on $\vec{x}$ that make f_i TRUE, and I is the number of inputs. The computation of MED breaks down into solving O #SAT problems, *i.e.*, $\#SAT(f_1), \ldots, \#SAT(f_O)$.

Apart from ER and MED, verifying other average error metrics can also be converted into #SAT problems similarly.

B. Approximation Miter

The approximation miter is a fundamental circuit used in average error verification. This paper adopts the approximation miter proposed in [4], depicted in Fig. 1, which implements the deviation function $F(\vec{y}(\vec{x}), \vec{y'}(\vec{x}))$ in Eq. (1). The miter takes the inputs $\vec{x}$ of both the exact and approximate circuits as its inputs. It produces $m \geq 1$ outputs, *i.e.*, $f_1, f_2, \ldots, f_O$, encoding the deviation function F. For instance, an approximation miter for ER consists of a single output f_1, representing the deviation F_{ER} defined in Eq. (2). In contrast, an approximation miter for MED has $m = O$ outputs encoding the deviation F_{MED} in Eq. (3). The approximation miter will be converted into DDs or CNF formulae for average error verification.

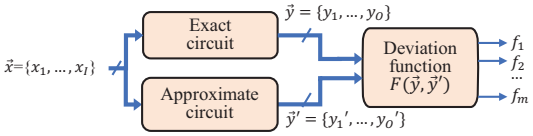


Fig. 1. Approximation miter for average error verification [4].

C. Model Counting and Conjunctive Normal Form

Given a Boolean function F, *model counting* (also called *#SAT*) computes the number of variable assignments making F TRUE. The Boolean function F is typically represented in CNF [14]. A CNF formula is a logical "AND" of one or more clauses, where each *clause* is a logical "OR" of one or more literals. A *literal* can be either the positive or negative form of a variable. To make a CNF formula TRUE, each clause must be TRUE. When a clause contains only a single literal, that literal must be set to TRUE to satisfy the clause. Such a clause is called a *unit clause*, and the literal in a unit clause is called a *unit literal*. The process of assigning all unit literals TRUE is called *unit propagation*.

III. Motivating Example and Key Idea

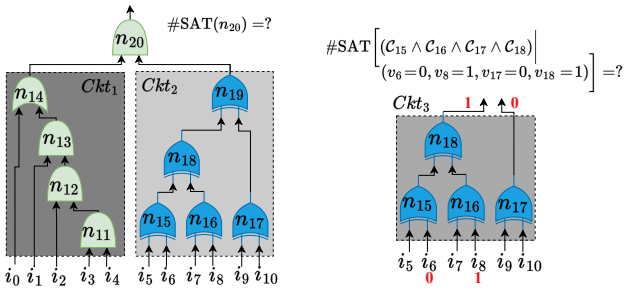
This section presents the motivation for VACSEM through a simple example and outlines our key idea.

Consider an approximation miter with 11 *primary inputs* (*PIs*), $\{i_0, i_1, \ldots, i_{10}\}$, and 1 *primary output* (*PO*), n_{20}, shown in Fig. 2(a). Consider the solving of two problems, $\#SAT(n_{14})$ and $\#SAT(n_{19})$, using traditional *Davis-Putnam-Logemann-Loveland-based (DPLL) method* [7] and circuit simulation.

The corresponding circuit of $\#SAT(n_{14})$, Ckt_1, has 3 AND gates, making it friendly for unit propagation with the DPLL method. For instance, if we assume $i_0 = 0$, then from $n_{14} = 1$, we can deduct $n_{13} = 1$, which then propagates to other variables, *i.e.*, $i_1 = i_2 = i_3 = i_4 = n_{11} = n_{12} = 1$. This process significantly reduces the number of variables to be decided, speeding up the computation of $\#SAT(n_{14})$ with only two decisions: $i_0 = 0$ and $i_0 = 1$. However, when simulating Ckt_1 to obtain $\#SAT(n_{14})$, we need to enumerate all 2^5 patterns of supporting PIs, simulate 4 gates, and count how many patterns lead to $n_{14} = 1$. In this case, DPLL method is more efficient.

The corresponding circuit of $\#SAT(n_{19})$, Ckt_2, has 5 XOR gates, making it less suitable for unit propagation. A SOTA DPLL-based #SAT solver like GANAK [9] makes 9 decisions to solve $\#SAT(n_{19})$. However, with circuit simulation, we can perform bitwise-XOR operations on 64-bit integers that store all 2^6 possible patterns on the 6 supporting PIs. This allows us to simulate all the input patterns in parallel, requiring only 5 XOR operations on 64-bit integers, which is much faster than the DPLL method.

Based on these observations, our key idea is integrating simulation into DPLL-based #SAT solver. Given that logic simulation and the DPLL method have their strengths in different scenarios, we design a dynamic controller to determine



(a) Example approximation miter. (b) Sub-circuit for simulation.

Fig. 2. Example of simulation-based #SAT solving.

when VACSEM should employ logic simulation and when it should use the DPLL method on the fly.

IV. VACSEM Methodology

This section elaborates VACSEM. As shown in Fig. 3, VACSEM consists of two phases: 1) circuit-aware construction of #SAT problems and 2) simulation-enhanced #SAT solving. They are described in Sections IV-A and IV-B, respectively.

A. Phase 1: Circuit-Aware Construction of #SAT Problems

Phase 1 converts the task of verifying average errors into #SAT problems represented in CNF, meanwhile retaining the circuit's topology for subsequent circuit simulation.

The conversion process is depicted on the left of Fig. 3. Initially, we construct an approximation miter, identical to the one shown in Fig. 1, to measure the deviation between the outputs of the exact and approximate circuits. The miter has m outputs, $f_1, f_2, \ldots, f_m$, encoding the deviation as $F = \sum_{j=1}^{m} 2^{j-1} f_j$. Then, we split the approximation miter into m sub-miters, each containing a single output f_j $(1 \leq j \leq m)$. Next, we perform logic synthesis on each sub-miter to reduce its size, which contributes to reducing the number of variables and clauses in the corresponding CNF formula. Afterward, each synthesized sub-miter is converted into a CNF formula, ready to be solved by our simulation-enhanced #SAT solver.

To convert a sub-miter to a CNF formula, we traverse each gate in the sub-miter in a topological order. For each gate n_k, we employ its consistency function [7] to generate a set of clauses $\mathcal{C}_k$. These clause sets for all gates are then combined using the logical AND operation, along with a unit clause containing the output variable. This results in the corresponding CNF formula for the sub-miter, as demonstrated in the example below.

Example 1. *For the miter in Fig. 2(a), Table I provides the clause set for each gate, ordered topologically. These clause sets in Table I are combined using the logical AND operation along with the unit clause, (n_{20}), to create the CNF formula* $f = \left(\bigwedge_{k=11}^{20} \mathcal{C}_k\right) \wedge n_{20}$.

This conversion ensures a clear *one-to-one* mapping between each node in the miter and each variable in the CNF formula. For instance, in Example 1, gate n_{14} corresponds to variable v_{14}, and PI i_0 corresponds to variable v_0.

Besides, there exists another *one-to-one* mapping between each gate n_k in the miter circuit and each clause set $\mathcal{C}_k$ in the CNF formula, as shown in Table I. This mapping from $\mathcal{C}_k$ to n_k enables us to easily identify the specific sub-circuit within the miter circuit that corresponds to a given CNF formula. To achieve this, we analyze each clause in the CNF formula, and determine its associated clause sets. By collecting all clause sets associated with the CNF formula, we can identify the corresponding gates, forming a sub-circuit of the miter circuit.

TABLE I. Conversion from the miter in Fig. 2(a) to CNF formula.

Gate	Clause set for the gate
$n_{11} = i_3 \wedge i_4$	$\mathcal{C}_{11} = (v_3 \vee \neg v_{11}) \wedge (v_4 \vee \neg v_{11}) \wedge (\neg v_3 \vee \neg v_4 \vee v_{11})$
$n_{12} = i_2 \wedge n_{11}$	$\mathcal{C}_{12} = (v_2 \vee \neg v_{12}) \wedge (v_{11} \vee \neg v_{12}) \wedge (\neg v_2 \vee \neg v_{11} \vee v_{12})$
$n_{13} = i_1 \wedge n_{12}$	$\mathcal{C}_{13} = (v_1 \vee \neg v_{13}) \wedge (v_{12} \vee \neg v_{13}) \wedge (\neg v_1 \vee \neg v_{12} \vee v_{13})$
$n_{14} = i_0 \vee n_{13}$	$\mathcal{C}_{14} = (\neg v_0 \vee v_{14}) \wedge (\neg v_{13} \vee v_{14}) \wedge (v_0 \vee v_{13} \vee \neg v_{14})$
$n_{15} = i_5 \oplus i_6$	$\mathcal{C}_{15} = (\neg v_5 \vee \neg v_6 \vee \neg v_{15}) \wedge (v_5 \vee v_6 \vee \neg v_{15}) \wedge (v_5 \vee \neg v_6 \vee v_{15}) \wedge (\neg v_5 \vee v_6 \vee v_{15})$
$\ldots$	$\mathcal{C}_{16} \sim \mathcal{C}_{19}$ are omitted here$\cdots$
$n_{20} = n_{14} \wedge n_{19}$	$\mathcal{C}_{20} = (v_{14} \vee \neg v_{20}) \wedge (v_{19} \vee \neg v_{20}) \wedge (\neg v_{14} \vee \neg v_{19} \vee v_{20})$

Furthermore, we arrange the clause sets in the same order as the topological order of gates. This organization inherently preserves the circuit topology within the resulting CNF formula. We will utilize the topology for logic simulation, Here is an example of converting a miter circuit to a CNF formula while maintaining the circuit topology.

Example 2. *Consider the CNF formula* $g = (\mathcal{C}_{15} \wedge \mathcal{C}_{16} \wedge \mathcal{C}_{17} \wedge \mathcal{C}_{18}) \big| (v_6 = 0, v_8 = 1, v_{17} = 0, v_{18} = 1)$. *In this CNF formula,* v_6, v_8, v_{17}, *and* v_{18} *have been decided by the #SAT solver as either 0 or 1. We analyze each clause in* f, *and collect* f*'s associated clause sets in the order that they appear in* f, *i.e.,* $\{\mathcal{C}_{15}, \mathcal{C}_{16}, \mathcal{C}_{17}, \mathcal{C}_{18}\}$. *These clause sets correspond to gates* n_{15}, n_{16}, n_{17}, *and* n_{18}, *which follow a topological order and forms a sub-circuit* Ckt_3 *shown in Fig. 2(b).*

B. Phase 2: Simulation-Enhanced #SAT Solving

After converting a miter circuit into a #SAT problem in CNF, Phase 2 utilizes circuit simulation to efficiently solve the #SAT problem. This subsection first shows the overall flow of the proposed simulation-enhanced #SAT solver. Then, it describes how to integrate circuit simulation into #SAT solver. After that, we explore the conditions under which simulation offers significant acceleration and introduce a dynamic simulation controller for deciding when to activate logic simulation.

Algorithm 1: *SIMSAT*(f), simulat.-enhanced #SAT solver.

Input: CNF formula f
Output: *#SAT*(f), #assignments making f TRUE
// If f is simulation-friendly, then simulate
if *SimulationController*(f) = *ENABLE_SIM* **then**
  return *SolveBySimulation*(f);
// Otherwise, solve by traditional DPLL method
Decision variable $l \leftarrow$ *DecideVariable*(f);
for *literal* $lit \leftarrow \{l, \neg l\}$ **do**
  Simplified formula $f|_{lit} \leftarrow$ *UnitPropagation*(f, lit);
  if $f|_{lit} \equiv 0$ **then** *#SAT*($f|_{lit}$) $\leftarrow 0$;
  else if $f|_{lit}$ *only contains a unit clause* **then**
    #SAT($f|_{lit}$) $\leftarrow 1$;
  else
    #SAT($f|_{lit}$) $\leftarrow 1$;
    A set of mutually independent formulae $\mathcal{DF} \leftarrow$ *DecomposeFormulae*($f|_{lit}$);
    foreach *formula* $d_i \in \mathcal{DF}$ **do**
      #SAT($f|_{lit}$) $\leftarrow$ *#SAT*($f|_{lit}$) $\times$ *SIMSAT*(d_i);
return *#SAT*($f|_l$) + *#SAT*($f|_{\neg l}$)

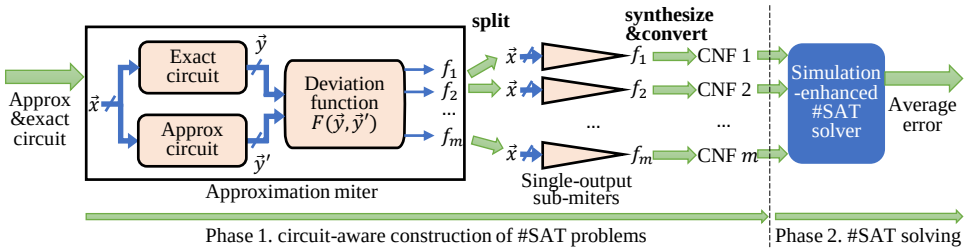


Fig. 3. VACSEM framework.

1) Overall Flow of Simulation-Enhanced #SAT Solver: As shown in Algorithm 1[1], the simulation-enhanced #SAT solver inputs a CNF formula f, and outputs *#SAT*(f), *i.e.*, the number of input patterns making f TRUE.

The key feature of the proposed #SAT solver is that it integrates circuit simulation into the traditional DPLL-based solver (Lines 1–2). The simulation controller, *SimulationController*(f), evaluates whether circuit simulation is more efficient to solve *#SAT*(f) than the DPLL method by analyzing the CNF formula f. If f's corresponding circuit is simulation-friendly (*e.g.*, Ckt_2 in Fig. 2(a)), the simulation controller returns *ENABLE_SIM*, and the #SAT problem is solved by circuit simulation (details in Section IV-B2). Otherwise, the simulation controller returns *DISABLE_SIM*, and the #SAT problem is solved by the DPLL method (Lines 3–14).

The DPLL method works as follows (Lines 3–14). It selects a variable from f based on certain heuristics. Then, it assigns both l and $\neg l$ to TRUE in turn (Line 4) and solves their respective sub-problems *#SAT*($f|_l$) and *#SAT*($f|_{\neg l}$) (Lines 5–13). For each literal *lit*, unit propagation is applied to simplify f, leading to a simplified formula $f|_{lit}$ (Line 5). Two trivial cases are considered for $f|_{lit}$. The first is when $f|_{lit}$ is always 0, resulting in *#SAT*($f|_{lit}$) = 0 (Line 6). The second is when $f|_{lit}$ only contains a unit clause (*e.g.*, $f|_{lit} = var_1$), in which *#SAT*($f|_{lit}$) = 1 (Lines 7–8). If $f|_{lit}$ does not fall into the two cases, it is split into several mutually independent formulae (*i.e.*, $\mathcal{DF} = \{d_1, d_2, \ldots, d_s\}$), where no two formulae share the same variable (Line 11). For example, in Fig. 2(a), the corresponding formulae of Ckt_1 and Ckt_2 are independent, since their supporting variables are disjoint. Then, each formula can be solved independently, updating *#SAT*($f|_{lit}$) by recursively computing $\Pi_{i=1}^{s}$*SIMSAT*(d_i) (Lines 12–13). Finally, *#SAT*(f) is obtained by summing up *#SAT*($f|_l$) and *#SAT*($f|_{\neg l}$) (Line 14).

2) #SAT Solving by Circuit Simulation: The function *SolveBySimulation*(f) in Algorithm 1 computes *#SAT*(f) by circuit simulation. We start by locating f's corresponding circuit, denoted as $\mathcal{G}$, using the method introduced in Section IV-A. As discussed before, we can easily obtain the topological order of gates in $\mathcal{G}$. Then, the simulation process consists of 3 steps: initializing patterns on $\mathcal{G}$'s inputs, updating patterns on $\mathcal{G}$'s gates, and counting consistent patterns in $\mathcal{G}$.

Step 1. Initializing patterns on $\mathcal{G}$'s inputs: Consider an input node n_k of $\mathcal{G}$. For each input node n_k in $\mathcal{G}$, we identify its corresponding variable v_k in f. Depending on whether v_k has been decided by the #SAT solver (which may occur during the variable decision process in Algorithm 1 Line 3 or unit propagation in Algorithm 1 Line 5), we apply different initialization strategies. If v_k has not been decided, we enumerate node n_k's two possible values, 0 and 1. Otherwise, we set node n_k's value to the decided value.

Assuming that there are K input nodes in $\mathcal{G}$ whose corresponding variables are not decided, there will be a total of 2^K possible input patterns. For each input node, we use a 2^K-bit *simulation vector* to represent its simulation patterns, where the k-th bit indicates the 0/1 value of the input node under the k-th input pattern. Besides, for input nodes whose corresponding variables have been decided as $b \in \{0,1\}$, their simulation vectors consist of 2^K bits, all set to b.

Example 3. *In Example 2, we locate the corresponding sub-circuit as Ckt_3 (depicted in Fig. 2(b)) for the CNF formula $g = (\mathcal{C}_{15} \wedge \mathcal{C}_{16} \wedge \mathcal{C}_{17} \wedge \mathcal{C}_{18}) \big| (v_6 = 0, v_8 = 1, v_{17} = 0, v_{18} = 1)$. Ckt_3 has 6 inputs, $i_5, i_6, \ldots, i_{10}$, corresponding to the variables $v_5, v_6, \ldots, v_{10}$ in f, respectively. The variables v_5, v_7, v_9, and v_{10} have not been decided, resulting in $2^4 = 16$ input patterns. The 16-bit simulation vectors for these input nodes are P_5, P_7, P_9, and P_{10}, respectively, in Table II. Besides, variables v_6 and v_8 have been decided as 0 and 1, respectively, and their simulation vectors are P_6 and P_8, respectively, in Table II.*

Step 2. Updating patterns on G's internal nodes: After initializing input nodes, we traverse each gate n_k in $\mathcal{G}$ in topological order, updating n_k's simulation vector based on n_k's functionality and the simulation vectors of n_k's fanins. For example, consider Ckt_3 depicted in Fig. 2(b). We traverse gates n_{15}, n_{16}, n_{17}, and n_{18} in the topological order and update their simulation vectors, shown as P_{15}, P_{16}, P_{17}, and P_{18}, respectively, in Table II. For instance, P_{15} is updated using the bitwise-XOR operation as $P_{15} = P_5$ *BITWISE-XOR* P_6.

TABLE II. SIMULATION VECTORS FOR NODES OF Ckt_3 IN FIG. 2(B). THE GATES MARKED WITH "CHECKING" ARE CHECKING GATES. THE SHADED INPUT PATTERNS ARE "CONSISTENT PATTERNS".

Circuit node		16-bit simulation vector
Input i_5	P_5 =	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Input i_7	P_7 =	0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
Input i_9	P_9 =	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
Input i_{10}	P_{10} =	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
Input i_6	P_6 =	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Input i_8	P_8 =	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Gate n_{15}	P_{15} =	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Gate n_{16}	P_{16} =	1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
Gate n_{17} (checking)	P_{17} =	**0** 0 0 **0** 1 1 1 1 1 1 1 1 **0** 0 0 **0**
Gate n_{18} (checking)	P_{18} =	**1** 0 0 **1** 1 0 0 1 1 0 0 1 **1** 0 0 **1**

Step 3. Counting consistent patterns in $\mathcal{G}$: Now, we have obtained simulation vectors of all nodes in $\mathcal{G}$. However, it is

[1]This algorithm is presented with recursion to make it easy to understand. Real implementation is a non-recursive version, including advanced techniques like clause learning, component caching, and variable ordering.

important to recognize that for a certain gate n_k in $\mathcal{G}$, its corresponding variable in the CNF formula f, denoted as v_k, may have already been decided as 0 or 1 by the #SAT solver. In this case, the simulation vector of n_k may contradict the decision made for v_k. We refer to gates with decided variables as *checking gates*. For instance, in *Ckt*$_3$ of Fig. 2(b), gate n_{17} has a simulation vector P_{17} = 0000 1111 1111 0000 (see Table II). This means n_{17} can be either 0 or 1 for different input patterns. Yet, its corresponding variable, v_{17}, has been decided as 0, indicating that n_{17} must be always 0 for all input patterns. Hence, n_{17} is a checking gate. Based on a similar analysis, n_{18} is also a checking gate.

We define a *consistent pattern* as an input pattern in $\mathcal{G}$, under which each checking gate n_k's simulation value is consistent with v_k's decision. For instance, the input pattern $i_5i_7i_9i_{10}i_6i_8 = 111101$ is a consistent pattern in *Ckt*$_3$. It is because under this input pattern, all checking gates' simulation values, $n_{17}=0$ and $n_{18}=1$, are consistent with the decisions $v_{17} = 0$ and $v_{18} = 1$ (see the rightmost shaded column in Table II). Based on this definition, we have the following obvious proposition for simulation-based *#SAT* solving:

Proposition 1. *#SAT(f) equals the number of consistent patterns in its corresponding circuit, $\mathcal{G}$.*

Example 4. *In Ckt*$_3$ *of Fig. 2(b), there are 4 consistent patterns, i.e., $i_5i_7i_9i_{10}i_6i_8 = \{000001, 110001, 001101, 111101\}$ (shaded columns in Table II). Under these patterns, the simulation values $n_{17} = 0$ and $n_{18} = 1$ are consistent with the decisions $v_{17}=0$ and $v_{18}=1$. Thus, the result for Ckt*$_3$*'s problem, #SAT(g), where $g=(\mathcal{C}_{15}\wedge\mathcal{C}_{16}\wedge\mathcal{C}_{17}\wedge\mathcal{C}_{18})|(v_6=0, v_8=1, v_{17}=0, v_{18}=1)$, is 4.*

3) Dynamic Simulation Controller: In Algorithm 1 Line 1, the *SimulationController*(f) decides when to employ simulation-based #SAT solving and when to use DPLL method. An ideal controller returns *ENABLE_SIM* if the simulation-based method is faster than DPLL method, and *DISABLE_SIM* if it is slower. Given that designing a real-time ideal controller is challenging, we instead develop an efficient controller based on the following analysis.

In simple terms, if f's corresponding circuit, denoted as $\mathcal{G}$, is dense (*e.g.*, with significantly more gates than primary inputs), DPLL method tends to slow down for several reasons:

- Complex variable dependencies reduce the effectiveness of unit propagation, causing a slowdown.
- The complex structure of dense circuits leads to deep and complex search trees with numerous branches. Working with such a tree, the branch prediction on the CPU tends to be inefficient, causing a slowdown.
- Finding a good variable order for DPLL solver is challenging in dense circuits, but a poor choice of order may dramatically slow down the solver.

In constrast, the simulation-based #SAT solving exploits bit-level parallelism and cache-friendly memory, and hence, it is more efficient than DPLL method in dense circuits.

However, when the circuit density of $\mathcal{G}$ is small, the DPLL method tends to be faster, following a similar analysis as described above. Thus, circuit density is a good metric to estimate whether simulation-based method is faster than DPLL method. We propose the following simple-to-compute but effective density metric:

$$\textit{density\_score} = \alpha \times \frac{\textit{gate number of } \mathcal{G}}{(\textit{PI number of } \mathcal{G})^2}, \tag{5}$$

where α is a constant scaling factor. We choose $\alpha = 2$ in our real implementation, based on the experimental results. If *density_score* > 1, then the gate number tends to be much larger than the PI number, indicating that $\mathcal{G}$ is dense. In this case, *SimulationController*($\mathcal{G}$) returns *ENABLE_SIM*. Otherwise, *DISABLE_SIM* is returned. This metric has shown significant acceleration in our experiments, compared to both DPLL and exhaustive simulation methods.

V. Experimental Results

This section presents the experimental results. We develop VACSEM based on GANAK [9], a SOTA open-source #SAT solver. Our experiments are conducted on a single core of an AMD Ryzen9 7945HX processor with 64GB RAM. In our implementation, we represent approximation miters using *AND-inverter graphs* (AIGs), although we can also support other circuit representations. We utilize ABC [15] for logic synthesis on the approximation miters, specifically using the *compress2rs* command to reduce the number of nodes in AIG.

Our experimental benchmarks, listed in Table III, consists of 20 exact circuits. These circuits serve as the basis for generating a series of approximate circuits using an approximate logic synthesis method proposed in [16]. For each benchmark, we randomly select 10 generated approximate circuits, resulting in a total of 200 approximate circuits to be verified. We will verify their average errors with three different methods[2]: VACSEM, the original GANAK, and exhaustive enumeration, as discussed in the following subsections.

TABLE III. Experimental benchmarks.

Type	Name	#PI	#PO	#Node	Name	#PI	#PO	#Node
adders & multipliers	adder32	64	33	337	mult12	24	24	1213
	adder64	128	65	695	mult14	28	28	1702
	adder128	256	129	1403	mult15	30	30	1951
	mult10	20	20	835	mult16	32	32	2429
EPFL benchmarks	ctrl	7	26	141	barshift	135	128	2688
	cavlc	10	11	1740	sin	24	25	7044
	dec	8	256	694	priority	128	8	1524
	int2float	11	7	585	router	60	30	198
BACS benchmarks	binsqrd	16	18	1562	butterfly	32	34	226
	absdiff	16	8	141	mac	12	8	145

A. Experiments on Adders and Multipliers

This experiment verifies ERs and MEDs for approximate adders and multipliers (see benchmarks at the top of Table III).

1) Verification of ER: We use VACSEM, the original GANAK, and the enumeration method to verify the ERs of these circuits, which have ER values ranging from 3×10^{-6} to 0.2. Table IV shows the results. Columns 2-4 list the geometric mean runtime over 10 approximate versions of each benchmark using the three methods. Columns 5-6 show the speedup of VACSEM compared to GANAK and enumeration, respectively. Note that VACSEM is always faster than GANAK

[2]We do not use DD-based methods to verify average errors due to their limited scalability. DD-based methods can only support circuits up to 32-bit adders and 8-bit multipliers [3]–[6], impractical for most circuits in Table III.

TABLE IV. VERIFYING ERS OF ADDERS AND MULTIPLIERS WITH DIFFERENT METHODS. THE RUNTIME LIMIT IS 14400S. **BOLD** ENTRIES DENOTE VACSEM IS FASTER THAN GANAK AND ENUMERATION.

Benchmark	Geomean runtime/s			Speedup	Speedup
	VACSEM	GANAK	Enum.	than GANAK	than enum.
adder32	**0.004**	0.005	>14400	1.246	$>3.7\times10^6$
adder64	**0.022**	0.025	>14400	1.127	$>6.4\times10^5$
adder128	**0.413**	0.458	>14400	1.655	$>1.1\times10^4$
mult10	**0.012**	0.113	0.017	9.546	1.445
mult12	**0.255**	53.864	0.342	210.8	1.339
mult14	**3.311**	>14400	8.246	>4349	2.490
mult15	**16.18**	>14400	39.487	>889.7	2.440
mult16	**116.0**	>14400	175.81	>124.2	1.516
GEOMEAN of speedup				**>35×**	**>161×**

and enumeration, and it can verify these adders within a second and multipliers in minutes. In contrast, GANAK cannot verify ERs of *mult14*, *mult15*, and *mult16* within 14400 seconds, since these circuits are dense, in which the node number significantly larger than the PI number. Besides, enumeration is impractical for verifying the adders because they have numerous input patterns. On average, VACSEM is more than 35× faster than GANAK, and more than 161× faster than the enumeration method.

2) Verification of MED: Similarly, we use VACSEM, original GANAK, and enumeration method to verify the MEDs of the approximate adders and multipliers, which have MED values ranging from 0.25 to 1.6×10^{26}. The results are shown in Table V, with columns indicating the geomean runtime and speedup. While VACSEM generally outperforms GANAK and enumeration, we also notice that GANAK performs similarly to VACSEM for *adder64* and *adder128*. Moreover, GANAK cannot verify the MEDs of the dense circuits *mult15* and *mult16*, and enumeration is impractical on the adders with numerous input patterns. In contrast, VACSEM can verify these adders within a few milliseconds and multipliers in minutes. On average, VACSEM is more than 10× faster than GANAK, and more than 633× faster than the enumeration method.

TABLE V. VERIFYING MEDS OF ADDERS AND MULTIPLIERS WITH DIFFERENT METHODS. RUNTIME LIMIT IS 14400S. **BOLD** ENTRIES DENOTE VACSEM IS FASTER THAN GANAK AND ENUMERATION.

Name	Runtime/s			Speedup	Speedup
	VACSEM	GANAK	Enum.	than GANAK	than enum.
adder32	**0.007**	0.007	>14400	1.009	$>2.0\times10^6$
adder64	0.0289	0.0288	>14400	0.996	$>5.0\times10^5$
adder128	0.098	0.097	>14400	0.989	$>1.5\times10^5$
mult10	**0.007**	0.029	0.098	4.020	13.36
mult12	**0.230**	56.66	1.769	246.7	7.700
mult14	**0.162**	0.941	32.77	5.811	202.3
mult15	**41.35**	>14400	158.12	>348.2	3.824
mult16	**361.7**	>14400	810.88	>39.81	2.242
GEOMEAN of speedup				**>10×**	**>633×**

B. Experiments on Other Types of Circuits

In addition to adders and multipliers, to show VACSEM's wide applicability, we test other types of approximate circuits from BACS [17] and EPFL [18] benchmarks (see Table III).

Table VI presents the ER verification results for these circuits using different methods, showing VACSEM's runtime and the speedup achieved by VACSEM over GANAK. Notably, VACSEM shows significant speedup over GANAK for several circuits, such as *cavlc* with a speedup of 3.6× and *binsqrd* with an impressive speedup of 1246.914×. For the *sin* circuit, GANAK exceeds the time limit (N/A), indicating that it cannot verify the ER within 14400 seconds. Overall, these results highlight VACSEM's efficiency and effectiveness in verifying the ERs of EPFL and BACS circuits.

TABLE VI. VERIFYING ERS OF EPFL AND BACS CIRCUITS WITH DIFFERENT METHODS. N/A MEANS CANNOT BE SOLVED IN 14400S. **BOLD** ENTRIES DENOTE VACSEM IS FASTER THAN GANAK.

Name	VACSEM runtime/s	Speedup than GANAK	Name	VACSEM runtime/s	Speedup than GANAK
ctrl	0.002	0.887×	barshift	0.945	**1.281×**
cavlc	0.008	**3.600×**	sin	115.959	**N/A**
dec	0.002	0.917×	priority	0.110	**1.060×**
int2float	0.004	**2.069×**	router	0.029	**1.114×**
binsqrd	0.010	**1246.914×**	butterfly	0.011	**2.034×**
absdiff	0.002	**1.123×**	mac	0.002	**2.113×**

VI. CONCLUSION

In this paper, we introduce VACSEM, a formal method for verifying average errors in circuits through simulation-enhanced model counting. VACSEM seamlessly integrates efficient circuit simulation into DPLL-based #SAT solvers, resulting in remarkable speedup, especially for high-density circuits. We also devise a dynamic controller that adaptively between circuit simulation and the DPLL method as needed. Compared to state-of-the-art average error verification methods, VACSEM significantly enhances scalability, enabling rapid verification of 128-bit adders within a second and 16-bit multipliers in minutes. In future work, we will expand VACSEM's capabilities to accommodate non-uniform input distributions.

REFERENCES

[1] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *ETS*, 2013, pp. 1–6.

[2] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, pp. 1–33, 2016.

[3] R. Venkatesan *et al.*, "MACACO: Modeling and analysis of circuits for approximate computing," in *ICCAD*, 2011, pp. 667–673.

[4] Y. Wu and W. Qian, "ALFANS: Multi-level approximate logic synthesis framework by approximate node simplification," *IEEE TCAD*, vol. 39, no. 7, pp. 1470–1483, 2019.

[5] V. Mrazek, "Optimization of BDD-based approximation error metrics calculations," in *ISVLSI*, 2022, pp. 1–6.

[6] S. Froehlich, D. Große, and R. Drechsler, "One method-all error-metrics: A three-stage approach for error-metric evaluation in approximate computing," in *DATE*, 2019, pp. 284–287.

[7] A. Biere *et al.*, *Handbook of satisfiability*. IOS press, 2009, vol. 185.

[8] M. Thurley, "SharpSAT–counting models with advanced component caching and implicit bcp," in *SAT*, Springer, 2006, pp. 424–429.

[9] S. Sharma *et al.*, "GANAK: A scalable probabilistic exact model counter.," in *IJCAI*, vol. 19, 2019, pp. 1169–1176.

[10] M. Soos and K. S. Meel, "BIRD: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting," in *AAAI*, vol. 33, 2019, pp. 1592–1599.

[11] J. K. Fichte *et al.*, "Parallel model counting with CUDA: Algorithm engineering for efficient hardware utilization," in *CP*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[12] A. T. Goldberg, *On the complexity of the satisfiability problem*. New York University, 1979.

[13] M. Soeken *et al.*, "BDD minimization for approximate computing," in *ASPDAC*, 2016, pp. 474–479.

[14] L. Zhang *et al.*, "Efficient conflict driven learning in a boolean satisfiability solver," in *ICCAD*, 2001, pp. 279–285.

[15] A. Mishchenko *et al.*, *ABC: A system for sequential synthesis and verification*, http://people.eecs.berkeley.edu/~alanmi/abc/, 2023.

[16] C. Meng *et al.*, "ALSRAC: Approximate logic synthesis by resubstitution with approximate care set," in *DAC*, 2020, pp. 1–6.

[17] Brown University Scale Lab, *BACS: Benchmarks for approximate circuit synthesis*, https://github.com/scale-lab/BACS, 2023.

[18] EPFL Integrated Systems Laboratory, *The EPFL combinational benchmark suite*, https://github.com/lsils/benchmarks, 2023.