DALTA: A Decomposition-based Approximate Lookup Table Architecture

Chang Meng[1], Zhiyuan Xiang[1], Niyiqiu Liu[1], Yixuan Hu[2], Jiahao Song[2], Runsheng Wang[2], Ru Huang[2], Weikang Qian[1,3]
[1]University of Michigan-Shanghai Jiao Tong University Joint Institute, Shanghai Jiao Tong University, Shanghai, China
[2]Institute of Microelectronics, Peking University, Beijing, China
[3]MoE Key Laboratory of Artificial Intelligence, Shanghai Jiao Tong University, Shanghai, China
Email: {changmeng, xzy242215, lnyq10}@sjtu.edu.cn, {yixuanhu, songjiaohao, r.wang, ruhuang}@pku.edu.cn, qianwk@sjtu.edu.cn

***Abstract*—A popular way to implement an arithmetic function is through a lookup table (LUT), which stores the pre-computed outputs for all the inputs. However, its size grows exponentially with the number of input bits. In this work, targeting at computing kernels of error-tolerant applications, we propose DALTA, a reconfigurable decomposition-based approximate lookup table architecture, to approximately implement those kernels with dramatically reduced size. We also propose integer linear programming-based approximate decomposition methods to map a given function to the architecture. Our architecture features with low energy consumption and high speed. The experimental results show that our architecture achieves energy and latency savings by 56.5% and 92.4%, respectively, over the state-of-the-art approximate LUT architecture.**

***Index Terms*—approximate lookup table, approximate decomposition, integer linear programming**

I. Introduction

As the transistor size shrinks into the nano-scale, energy efficiency has become a crucial concern for computing systems [1]. One popular technique for low-energy design is computing with memory [2]. By pre-computing frequently-used functions and storing the results in a lookup table (LUT), the function can be efficiently computed at run-time by reading the LUT with low energy consumption [3]. However, an apparent drawback is that the LUT size grows exponentially with the number of input bits: to store a function with n input bits, a LUT with 2^n entries is needed. Thus, the advantage of computing with memory diminishes for complex functions with many inputs.

To address the drawback, many works turn to another popular low-energy design technique, approximate computing [4], and combine both techniques to implement complex functions in LUTs. Targeting at many error-tolerant applications, approximate computing deliberately introduces a small amount of error into the system. If errors are carefully introduced, the application-level quality is almost unaffected, while the hardware cost is reduced dramatically. Taking advantage of this, approximate computing techniques are applied to simplify the function, and hence reduce the required LUT size.

There are many studies focusing on the design of approximate LUTs. Some early works divide the function's input x into segments and split the original LUT into multiple LUTs. In [5], Schulte and Stine proposed a bipartite LUT to approximate functions based on first-order Taylor approximation. It divides the input x into three segments and uses them to index the two LUTs. The lookup results of all the segments are summed up to approximate the original value $f(x)$. In [6], they further extended the bipartite LUT to the multipartite LUT. Later works further proposed various other multipartite LUT architectures [7]–[9]. However, they all rely on the first-order Taylor approximation of the original functions, and hence they do not support non-continuous functions. With the advent of low-cost associative memristive memories, some recent works proposed to only store part of input-output patterns of the original function, and search the LUT by approximate input pattern matching [10], [11]. Typical pattern matching measures include the Hamming distance [10] and the binary encoding distance [11]. If there is an input pattern $\hat{x}$ in the LUT that is close enough to a query x, it is treated as a successful pattern matching. Then, the corresponding output pattern $f(\hat{x})$ is returned as an approximation to $f(x)$. Otherwise, $f(x)$ is computed by conventional methods. Apparently, additional hardware for accurate computing is required when pattern matching fails. In a recent work [12], Tian *et al.* proposed ApproxLUT, which applies Taylor approximation to the pattern matching-based method. ApproxLUT stores the input-output patterns $(a, f(a))$, as well as the derivative $f'(a)$. When an input x comes, it matches the input x with the nearest input pattern a and then computes the approximate function as $f(a) + f'(a)(x - a)$. Unfortunately, as it is also based on Taylor approximation, it can only deal with continuous function. Furthermore, it requires additional arithmetic circuits to obtain the final result.

To approximate arbitrary functions regardless of their continuity, we propose **DALTA**, a **decomposition-based approximate lookup table architecture**, in this work. Our method is inspired by the conventional Boolean function decomposition theory, which was pioneered by Ashenhurst [13], Curtis [14], Roth and Karp [15]. Later, Lai *et al.* extended the theory using the **binary decision diagram** (**BDD**) and presented how to decompose multiple-output functions [16] with BDDs. Lee *et al.* further proposed to decompose large functions by interpolation and satisfiability solving [17]. In this work, we extend the conventional function decomposition theory and propose several approximate decomposition methods. With these methods, a function with many inputs can be approximately decomposed into two simpler functions with fewer inputs, which are stored into two smaller LUTs. In this way, the LUT size is cut down dramatically, and so is the area, delay, and energy of the LUT architecture. Theoretically, our method enjoys an exponential decrease for LUT size over the accurate LUT as the number of inputs increases.

Our main contributions are as follows:

- We propose DALTA, a decomposition-based approximate lookup table architecture, which improves area, delay, and energy dramatically by introducing a small error. DALTA can implement approximations for both continuous and non-continuous functions.
- To map a function into DALTA, we propose approximate decomposition methods based on **integer linear programming** (**ILP**).
- We also propose an efficient heuristic method to derive approximate decomposition.

This work is supported in part by the National Key R&D Program of China under Grant 2020YFB2205500 and the State Key Laboratory of ASIC & System under Open Research Grant 2019KF004. Corresponding authors: Runsheng Wang and Weikang Qian.

DALTA is a high-speed energy-efficient architecture that is applicable to arbitrary functions. On the one hand, DALTA can calculate continuous functions with 56.5% energy saving and 92.4% latency reduction, compared with the state-of-the-art approximate LUT architecture, ApproxLUT [12]. On the other hand, DALTA works well on non-continuous functions. Compared with the rounding-based approximate LUT architecture, it saves area, energy, and latency by 95.8%, 39.0%, and 98.3%, respectively.

The rest of the paper is organized as follows. Section II provides the preliminaries on disjoint decomposition. Section III explains our basic idea. Section IV elaborates DALTA. Section V presents the approximate decomposition methods. Section VI shows the experimental results. Finally, Section VII concludes the paper.

II. Preliminaries: Disjoint Decomposition

In this section, we present the preliminaries on traditional disjoint decomposition.

x_1x_2 \ x_3x_4	00	01	10	11	
00	0	1	1	0	$\phi \wedge \overline{x_1} \wedge \overline{x_2}$
01	1	0	0	1	$\bar{\phi} \wedge \overline{x_1} \wedge x_2$
10	1	1	1	1	$x_1 \wedge \overline{x_2}$
11	1	0	0	1	$\bar{\phi} \wedge x_1 \wedge x_2$

Fig. 1. A 2D truth table, or Boolean matrix, of a function f.

Definition 1 *Let f be a Boolean function of n variables and $X = \{x_1, \ldots, x_n\}$ be its inputs. Let $\{A, B\}$ be a partition on X. The function f has a **disjoint decomposition** with **free set** A and **bound set** B if there exist functions ϕ and F such that $f(X) = F(\phi(B), A)$. The function ϕ and F are called the **bound-set function** and the **free-set function**, respectively. If the function f has a disjoint decomposition, the function is said to be **decomposable**.*

Not every Boolean function is decomposable. Ashenhurst gives a necessary and sufficient condition for the existence of a disjoint decomposition with a given partition on the input variables [13]. It is based on a **2-dimensional (2D) truth table** representation of the Boolean function, in which some variables define the columns and the remaining variables define the rows; an example is shown in Fig. 1. In what follows, we will also call this representation a **Boolean matrix**. The necessary and sufficient condition is given by the theorem below.

Theorem 1 *Let $\{A, B\}$ be a partition on X. A Boolean function f is decomposable with free set A and bound set B, if and only if the Boolean matrix with the variables in A and B defining the rows and the columns, respectively, has at most four distinct types of rows that can be classified into the following categories: 1) a pattern of all 0s; 2) a pattern of all 1s; 3) a fixed pattern p of 0's and 1's; 4) the complement of the pattern p.*

A proof to the above theorem can be found in [18]. We use the following example to illustrate how to obtain the disjoint decomposition once the condition in Theorem 1 is satisfied.

Example 1 *Fig. 1 shows a Boolean matrix of a Boolean function $f(x_1, x_2, x_3, x_4)$ with variables x_1, x_2 defining the rows and variables x_3, x_4 defining the columns. It satisfies the condition described in Theorem 1: row 1 falls into Category 3, rows 2 and 4 fall into Category 4, and row 3 falls into Category 2. Therefore, function f is decomposable with free set as $\{x_1, x_2\}$ and bound set as $\{x_3, x_4\}$. We can set the truth table of the function $\phi(x_3, x_4)$ as the pattern in Category 3. For this example, the truth table is "0110" and correspondingly, $\phi(x_3, x_4) = \bar{x}_3 x_4 + x_3 \bar{x}_4$. The functions for the 1st, 2nd, 3rd, and 4th row of the Boolean matrix are shown on the right of the Boolean matrix in Fig. 1. The final expression of f is $f = \phi \bar{x}_1 \bar{x}_2 + \bar{\phi} \bar{x}_1 x_2 + x_1 \bar{x}_2 + \bar{\phi} x_1 x_2 = F(\phi, x_1, x_2)$.*

III. Key Idea: Approximate Disjoint Decomposition

To facilitate later description, we present the key idea in this section, which is **approximate disjoint decomposition**. As we will show later, if a function is decomposable, we can implement it by a more efficient LUT structure. However, an arbitrary function may not be decomposable. Nevertheless, if we deliberately change the outputs of some input patterns, we can obtain a decomposable function. We call such a technique approximate disjoint decomposition, or **approximate decomposition** in short.

x_1x_2 \ x_3x_4	00	01	10	11
00	0	1	0	0
01	1	0	0	1
10	0	0	1	0
11	1	0	1	1

(a)

x_1x_2 \ x_3x_4	00	01	10	11
00	0	0	0	0
01	1	0	1	1
10	0	0	0	0
11	1	0	1	1

(b)

Fig. 2. Boolean matrices of (a) a non-decomposable function and (b) a decomposable function that approximates the function in (a).

Example 2 *Fig. 2(a) shows a single-output Boolean function represented as a Boolean matrix. From the matrix, we can see that the function is not decomposable with the free set $\{x_1, x_2\}$ and the bound set $\{x_3, x_4\}$, since the condition in Theorem 1 is not satisfied. However, by flipping the outputs of some input patterns, we can construct a decomposable function; an example is shown in Fig. 2(b) with the changed outputs shown in red.*

As there are many ways to change the outputs, we look for one that minimizes the error. In order to evaluate the error induced by a change, the occurrence probability of each input pattern should be known. This can be obtained by analyzing the typical input data set. Thus, in what follows, we assume that such probabilities are known.

It should be noted a previous work [19] also considers the approximate decomposition of a function. In [19], Yao *et al.* proposed a technique to find approximate functions of maximum fanout-free cones (MFFCs), which is a special region in a circuit. However, our method is quite different from theirs. First, their method is limited to single-output functions, while ours can deal with multiple-output functions. Second, their method is a heuristic one that finds local optimum, while we formulate the approximate decomposition problem into ILP and try to find the global optimum. Finally, their method aims at designing logic circuits, while we aim at finding patterns to be stored in the proposed approximate LUT architecture.

In the following sections, we will first present DALTA, a LUT architecture supporting the computation based on decomposition. Then, we will elaborate how to find the closest approximate decomposition for a given function so that it can be computed by DALTA.

IV. Decomposition-based Approximate LUT Architecture

In this section, we present DALTA, the proposed decomposition-based approximate LUT architecture.

We consider implementing an n-input m-output Boolean function $Y = G(X)$, where $X = (x_1, \ldots, x_n)$ with each $x_i \in \{0, 1\}$, $Y = (y_1, \ldots, y_m)$ with each $y_i \in \{0, 1\}$, and $G = (g_1, \ldots, g_m)$ with each g_i as an n-input single-output Boolean function. DALTA implements an approximation of G, denoted as $\hat{G} = (\hat{g}_1, \ldots, \hat{g}_m)$, where $\hat{g}_i$ ($1 \leq i \leq m$) is an n-input single-output Boolean function approximating g_i. In what follows, we call G and $\hat{G}$ **group functions** and call g_i and $\hat{g}_i$ **component functions**.

DALTA is shown in Fig. 3. It consists of m identical **approximate single-output LUTs**, each implementing an n-input single-output Boolean function. These LUTs share the n-bit input X and the configuring signal *config* as inputs, and their outputs $\hat{y}_1, \ldots, \hat{y}_m$ are combined into an m-bit output $\hat{Y}$.

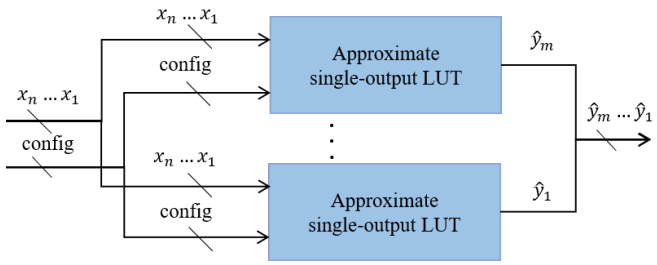

Fig. 3. Overall decomposition-based approximate LUT architecture.

Fig. 4 illustrates the structure of an approximate single-output LUT. It has three components: a LUT pair, a routing box and a configuring box.

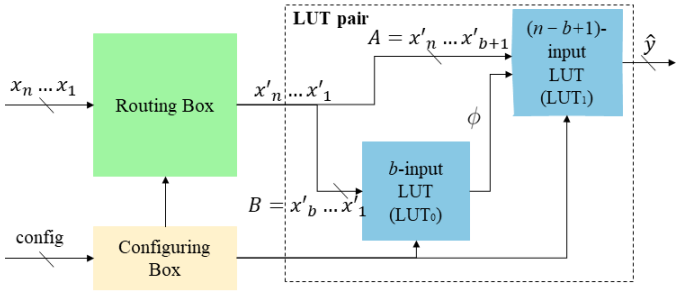

Fig. 4. Approximate single-output LUT.

A LUT pair, as shown in Fig. 4, consists of a b-input LUT (LUT_0) and an $(n - b + 1)$-input LUT (LUT_1). A routing box shuffles the input $X = (x_1, \ldots, x_n)$ into $X' = (x'_1, \ldots, x'_n)$. The first b inputs of X', $x'_1, \ldots, x'_b$, are fed into LUT_0, while LUT_1 has its inputs as the remaining inputs $x'_{b+1}, \ldots, x'_n$ together with the output of LUT_0. The routing box can be easily implemented by n n-to-1 multiplexers (MUXs). The jth ($1 \leq j \leq n$) MUX is controlled by a $\lceil \log_2(n) \rceil$-bit selection signal to decide the routing from X to the jth output x'_j.

Based on the routing box and structure of the LUT pair, we can easily see that an approximate single-output LUT computes a single-output decomposable function

$$f(X) = F(\phi(B), A), \tag{1}$$

where A and B are the free set of size $(n - b)$ and the bound set of size b, respectively. If we can find an approximate decomposition of the function g_i, $\hat{g}_i$, satisfying the form shown in Eq. (1), we can implement $\hat{g}_i$ by the approximate single-output LUT.

Notably, implementing functions with the LUT pair reduces the storage cost dramatically. Storing the original function g_i requires a LUT of 2^n entries, In contrast, storing its approximation $\hat{g}_i$ using the LUT pair only needs $2^b + 2^{n-b+1}$ LUT entries. A typical configuration sets b as $\lceil \frac{n}{2} \rceil$ so that the costs of LUT_0 and LUT_1 are balanced. In this case, the LUT pair structure only requires $\frac{3}{2^{n/2}}$ and $\frac{1}{2^{(n-3)/2}}$ of entries compared to the accurate LUT structure for even and odd n, respectively.

The routing box provides design flexibility. It allows different approximate single-output LUTs to have different partitions of the free and bound sets. An alternative architecture is one with a single routing box shared by all the LUT pairs. This reduces the circuit area and power, but the configuration space is also reduced, which may cause an increase in approximation error.

Each approximate signal-output LUT is configured by a configuring box. It takes configuring signals *config* as input, including the free-set and bound-set functions to be stored in the LUT pair and the routing box configuration signals.

V. Approximate Decomposition Methods

As mentioned above, the proposed LUT architecture implements an approximation of the given group function G through approximate decomposition of each component function g_i of G. In this section, we elaborate how to derive a close approximation for G. We first focus on finding approximate decomposition by considering each component function separately (see Section V-A). We then describe how to jointly decompose component functions by taking their significance into account (see Section V-B).

A. Separate Decomposition of Component Functions

In this section, we present a method that finds the approximate decomposition for each component function g_k separately. We first introduce our method by assuming that g_k's free and bound sets are given. Later, we will describe how to explore different partitions of the free and bound sets.

We focus on a particular component function g_k ($1 \leq k \leq m$). The task is to find an approximate decomposition with the given variable partition that is closest to g_k. The proposed method is based on analyzing a Boolean matrix with variables in the free and bound sets defining the rows and the columns, respectively. We will use the following notations in this and later sections.

- We denote the variable partition of the component function g_k as $\Omega_k = \{A_k, B_k\}$, where A_k and B_k are the free set and the bound set, respectively. Note that $|A_k| = n - b$ and $|B_k| = b$.
- We define $r = 2^{n-b}$ and $c = 2^b$. By the proposed architecture, r and c equal the numbers of rows and columns of the Boolean matrix, respectively.
- We denote the uth ($1 \leq u \leq rc$) input pattern of the component function g_k as X_{ku}. The outputs of g_k and $\hat{g}_k$ under X_{ku} are denoted as y_{ku} and $\hat{y}_{ku}$, respectively.
- We denote the input pattern corresponding to the cell at row i ($1 \leq i \leq r$) and column j ($1 \leq j \leq c$) of the Boolean matrix under the variable partition of g_k as I_{kij}. I_{kij} occurs with probability p_{kij}. The outputs of g_k and $\hat{g}_k$ under I_{kij} are denoted as o_{kij} and $\hat{o}_{kij}$, respectively.
- We define the mapping from the index (i, j) of the 2D Boolean matrix under the variable partition of g_k to the index u of the 1D

truth table as H. Specifically, we write $H(k,i,j)=u$, where k corresponds to the kth component function g_k. Then, we have

$$\begin{aligned} I_{kij} &= X_{k(H(k,i,j))}, \\ o_{kij} &= y_{k(H(k,i,j))}, \\ \hat{o}_{kij} &= \hat{y}_{k(H(k,i,j))}. \end{aligned} \tag{2}$$

A component function is a single-output function. A natural error measure for the approximation of a single-output function is error rate (ER), which is defined as the ratio of input patterns with a wrong output for the approximate function. Given the above notations, ER can be calculated as

$$ER = \sum_{i=1}^{r}\sum_{j=1}^{c} p_{kij}|o_{kij} - \hat{o}_{kij}|.$$

Our task is to find a decomposable function $\hat{g}_k$ that minimizes the ER. By Theorem 1, when the free and bound sets are given, a decomposable function can be characterized by two variables of the Boolean matrix with the free and bound sets. The first is the row pattern in Category 3 of Theorem 1. We represent it as a vector $V = (v_1, \ldots, v_c) \in \{0,1\}^c$, which we call a **pattern vector**. The second is the collection of the category indices of all the rows. We represent it as a vector $T = (t_1, \ldots, t_r) \in \{1,2,3,4\}^r$, which we call a **row-type vector**. The entry t_i represents one of the four categories listed in Theorem 1 that the ith row belongs to. For example, for the decomposable function shown in Fig. 2(b), its pattern vector is $V = (1,0,1,1)$ and its row-type vector is $T = (1,3,1,3)$. Thus, in order to find the optimal decomposable function, we only need to find its corresponding pattern and row-type vectors.

Actually, for this problem, we only need to solve for the pattern vector. We can formulate an optimization problem as follows with the unknowns to be solved as v_j $(1 \le j \le c)$.

$$\min \textstyle\sum_{i=1}^{r} E_i, \tag{3}$$

$$\begin{aligned} s.t. \quad & E_i = \min\{e_{i1}, e_{i2}, e_{i3}, e_{i4}\}, \text{ for } 1 \le i \le r, \\ & e_{i1} = \textstyle\sum_{j=1}^{c} p_{kij} o_{kij}, \text{ for } 1 \le i \le r, \\ & e_{i2} = \textstyle\sum_{j=1}^{c} p_{kij}(1 - o_{kij}), \text{ for } 1 \le i \le r, \\ & e_{i3} = \textstyle\sum_{j=1}^{c} p_{kij}|v_j - o_{kij}|, \text{ for } 1 \le i \le r, \\ & e_{i4} = \textstyle\sum_{j=1}^{c} p_{kij}|1 - v_j - o_{kij}|, \text{ for } 1 \le i \le r, \\ & v_j \in \{0,1\}, \text{ for } 1 \le j \le c. \end{aligned} \tag{4}$$

In the above formulation, e_{id} $(1 \le i \le r,\ 1 \le d \le 4)$ is the ER of row i of the Boolean matrix of $\hat{g}_k$ if it is in Category d. In order to minimize the ER of $\hat{g}_k$, row i of the Boolean matrix of $\hat{g}_k$ should belong to the category that minimizes the ER of that row. Correspondingly, the minimum ER for row i, E_i, is given by Eq. (4). The optimization target is the sum of E_i over all the rows (see Eq. (3)).

The above formulation can be further transformed into an ILP problem. The non-linear terms include the absolute value function $|f|$ and the minimum function. The former can be transformed into a maximum function as $|f| = \max\{f, -f\}$. Then, we can transform the minimum and maximum functions further into linear constraints [20]. Furthermore, we can relax the constraint $v_j \in \{0,1\}$ to a linear constraint $0 \le v_j \le 1$ for further acceleration. Our experiment results show that under the uniform input distribution, such a relaxation does not affect the optimality.

Up to now, we have obtained an approximate decomposition for a single-output component function g_k with a fixed bound and free set. Since different partitions of the bound and free sets can result in different errors between g_k and $\hat{g}_k$, different partitions should be explored. However, the total number of partitions grows exponentially with the input number. To limit the runtime, we set a limit P as the maximum number of partitions to be explored. If the total number of partitions exceeds P, then we will randomly explore P partitions. Otherwise, all possible partitions are evaluated.

B. Joint Decomposition of Component Functions

The separate decomposition presented in Section V-A has a drawback that it ignores the different significance of the outputs $y_1, \ldots, y_m$. In many applications, the outputs encode a numerical value as $\sum_{k=1}^{m} 2^{k-1} y_k$. The ignorance of output significance can cause a large error. For example, consider an accurate output $(y_3, y_2, y_1) = (1,0,0)$ and its two approximations, $(0,0,0)$ and $(0,1,1)$. The separate decomposition prefers the first approximation over the second, as the last two bits of the first are still correct, while those of the second are both incorrect. However, in terms of the numerical value, the second should be preferred, as it is closer to the accurate numerical value than the first.

To address the issue of the separate decomposition, we propose a joint decomposition of all the component functions in this section. The same notations in Section V-A are used here. We also assume the outputs encode a numerical value in the binary radix format. We use the **normalized mean error distance (NMED)** to measure the error, which is calculated as follows:

$$NMED = \frac{1}{2^m - 1}\sum_{i=1}^{r}\sum_{j=1}^{c} p_{kij}\left|\sum_{k=1}^{m} 2^{k-1}\left(o_{kij} - \hat{o}_{kij}\right)\right| \tag{5}$$

Our task is to find the joint approximate decomposition that minimizes the NMED. Next, we present two ILP-based solutions with different variable partition strategies and an efficient heuristic solution.

1) Decomposition with Shared Variable Partition: The variable partition strategy used here is that all the component functions have the same partition. In this case, all the approximate single-output LUTs can share one routing box, thus, reducing the area and power.

As mentioned in Section V-A, each approximate function $\hat{g}_k$ is fully determined by its pattern and row-type vectors. We denote the pattern vector of $\hat{g}_k$ as $V_k = (v_{k1}, \ldots, v_{kc})$ and the row-type vector as $T_k = (t_{k1}, \ldots, t_{kr})$. To facilitate the ILP formulation, we introduce binary indicator variables s_{kid} $(1 \le k \le m,\ 1 \le i \le r,\ 1 \le d \le 4)$ to encode a row-type vector. Specifically, $s_{kid} = 1$ if and only if $t_{ki} = d$, i.e., the ith row in the Boolean matrix of $\hat{g}_k$ belongs to Category d. With the variables s_{kid}, we can formulate the optimization problem as follows.

$$\min \tfrac{1}{2^m-1}\textstyle\sum_{i=1}^{r}\sum_{j=1}^{c} p_{kij}\left|\sum_{k=1}^{m} 2^{k-1}\left(o_{kij} - \hat{o}_{kij}\right)\right| \tag{6}$$

$$\begin{aligned} s.t. \quad & \hat{o}_{kij} = s_{ki0}\cdot 0 + s_{ki1}\cdot 1 + s_{ki2} v_{kj} + s_{ki3}(1 - v_{kj}), \\ & \text{for } 1 \le k \le m, 1 \le i \le r, 1 \le j \le c, \end{aligned} \tag{7}$$

$$\textstyle\sum_{d=1}^{4} s_{kid} = 1, \text{ for } 1 \le k \le m, 1 \le i \le r, \tag{8}$$

$$s_{kid} \in \{0,1\}, \text{ for } 1 \le k \le m, 1 \le i \le r, 1 \le d \le 4, \tag{9}$$

$$v_{kj} \in \{0,1\}, \text{ for } 1 \le k \le m, 1 \le j \le c.$$

In the above formulation, Eq. (6) is same as Eq. (5), which calculates the NMED. Eq. (7) gives the output of the approximate function $\hat{g}_k$ under the input pattern I_{kij}, based on the definition of the binary indicator variables. Eqs. (8) and (9) are the basic constraints on the binary indicator variables s_{kid}. The above formulation includes

two non-linear computations, absolute value function and product of two binary variables. Fortunately, both of them can be transformed into linear constraints [20]. Thus, the above formulation can be transformed into an ILP problem. Similar to Section V-A, we explore at most P variable partitions for runtime concern.

2) Decomposition with Non-shared Variable Partition: The variable partition strategy used here is that all the component functions can have different partitions. Compared with the strategy in Section V-B1, it explores more on the variable partitions and may achieve much smaller error.

The method finds decomposition of each component function sequentially from the **most significant bit** (**MSB**) to the **least significant bit** (**LSB**). When decomposing the kth component function g_k, we assume that the rest component functions $\hat{g}_\lambda$ $(1 \leq \lambda \leq m, \lambda \neq k)$ have been known. In other words, we have already known the outputs of the rest component functions $\hat{y}_{\lambda u}$ $(1 \leq \lambda \leq m, \lambda \neq k, 1 \leq u \leq rc)$. Then, under a given variable partition Ω_k, in order to find the optimal decomposable function $\hat{g}_k$, we can formulate the optimization problem as follows with the unknowns v_j $(1 \leq j \leq c)$.

$$\min \textstyle\sum_{i=1}^{r} E_i, \tag{10}$$

$$s.t. \quad E_i = \tfrac{1}{2^m-1} \min\{e_{i1}, e_{i2}, e_{i3}, e_{i4}\}, \text{for } 1 \leq i \leq r, \tag{11}$$

$$u_{ij} = H(k, i, j), \text{for } 1 \leq i \leq r, 1 \leq j \leq c, \tag{12}$$

$$Y_{ij} = \textstyle\sum_{\lambda=1}^{m} 2^{\lambda-1} y_{\lambda u_{ij}}, \text{for } 1 \leq i \leq r, 1 \leq j \leq c, \tag{13}$$

$$\hat{Y}_{ij} = \textstyle\sum_{\lambda=1}^{k-1} 2^{\lambda-1} \hat{y}_{\lambda u_{ij}} + \sum_{\lambda=k+1}^{m} 2^{\lambda-1} \hat{y}_{\lambda u_{ij}}, \quad \text{for } 1 \leq i \leq r, 1 \leq j \leq c, \tag{14}$$

$$D_{ij} = \hat{Y}_{ij} - Y_{ij}, \text{for } 1 \leq i \leq r, 1 \leq j \leq c, \tag{15}$$

$$e_{i1} = \textstyle\sum_{j=1}^{c} p_{kij} \left| D_{ij} \right|, \text{for } 1 \leq i \leq r, \tag{16}$$

$$e_{i2} = \textstyle\sum_{j=1}^{c} p_{kij} \left| D_{ij} + 2^{k-1} \right|, \text{for } 1 \leq i \leq r, \tag{17}$$

$$e_{i3} = \textstyle\sum_{j=1}^{c} p_{kij} \left| D_{ij} + 2^{k-1} v_j \right|, \text{for } 1 \leq i \leq r, \tag{18}$$

$$e_{i4} = \textstyle\sum_{j=1}^{c} p_{kij} \left| D_{ij} + 2^{k-1}(1 - v_j) \right|, \text{for } 1 \leq i \leq r, \tag{19}$$

$$v_j \in \{0, 1\}, \text{for } 1 \leq j \leq c.$$

In the above formulation, Eq. (12) converts the index (i, j) of the 2D Boolean matrix to the index u_{ij} of the 1D truth table. Eq. (13) calculates the accurate output encoded by $g_1, \ldots, g_m$ under the input pattern I_{kij}. Eq. (14) computes the approximate output encoded by $\hat{g}_1, \ldots, \hat{g}_{k-1}, 0, \hat{g}_{k+1}, \ldots, \hat{g}_m$ under the input pattern I_{kij}. Eq. (15) gives the difference between the above two output values.

In Eqs. (16)–(19), e_{id} $(1 \leq i \leq r,\ 1 \leq d \leq 4)$ denotes the mean error distance (MED) over the input patterns on row i of $\hat{g}_k$'s Boolean matrix if the row is in Category d. For example, consider e_{i3}. Since $d = 3$, the pattern vector $(v_1, \ldots, v_c)$ is selected as the output for the ith row of $\hat{g}_k$'s Boolean matrix. Thus, we have $\hat{o}_{kij} = v_j$ $(1 \leq j \leq c)$. By Eqs. (2) and (12), we further have $\hat{y}_{ku_{ij}} = v_j$. Combining this equation with Eqs. (13), (14), and (15), we have

$$D_{ij} + 2^{k-1} v_j = \sum_{\lambda=1}^{m} 2^{\lambda-1} (\hat{y}_{\lambda u_{ij}} - y_{\lambda u_{ij}}).$$

Combining the above equation with Eq. (16), we can conclude that e_{i3} indeed gives the MED over the input patterns on row i of $\hat{g}_k$'s Boolean matrix if the row is in Category 3.

In order to minimize the NMED, row i of the Boolean matrix should belong to the category that minimizes the MED of that row. Correspondingly, the minimum MED for row i, E_i, is given by Eq. (11). The optimization target is the sum of E_i over all the rows (see Eq. (10)). Similar to Section V-A, the above formulation can be transformed into an ILP problem.

Since the error measure is NMED, the MSBs contribute more to the error. Therefore, we iteratively solve the above optimization problem from the MSB to the LSB, as shown in Algorithm 1. In other words, the approximate functions on the MSB, the second MSB, up to the LSB are solved sequentially. The inputs of Algorithm 1 include the iteration round R, the variable partition limit P, and the accurate component functions $Y = \{y_{ku} | 1 \leq k \leq m, 1 \leq u \leq rc\}$. The outputs include the best variable partitions for the component functions $\Omega^* = (\Omega_1, \ldots, \Omega_m)$, the corresponding pattern vectors $V^* = (V_1^*, \ldots, V_m^*)$, and the row-type vectors $T^* = (T_1^*, \ldots, T_m^*)$.

Note that the above optimization problem assumes that all the approximate component functions except for $\hat{g}_k$ are known, which clearly is not held initially. To solve this issue, we use the accurate component function as an estimate to the approximate component function (Line 1). Then, the algorithm iterates for R rounds, and for each round, it decomposes each component functions from MSB to LSB. When finding the kth approximate component function $\hat{g}_k$, a set Φ of variable partition candidates of size no more than P (similar to Section V-A) is generated (Line 5). Each partition candidate ω is evaluated by solving the optimization problem in Eq. (10) (Lines 6–8). The minimal NMED E_k^* under those partitions is kept, as well as the corresponding variable partition Ω_k^*, pattern vector V_k^*, and row-type vector T_k^* (Lines 9–10). After checking all the variable partitions in set Φ, we update the approximate function estimation $\hat{y}_{ku}$ according to the kept Ω_k^*, V_k^*, T_k^* (Line 11). Finally, the best variable partitions, pattern vectors, and row-type vectors of all the component functions are put together and returned (Lines 12–13).

Algorithm 1: *Iterative_Approximate_Solution*(R, P, Y)

Input: Iteration round R, variable partition limit P, accurate function $Y = \{y_{ku} | 1 \leq k \leq m, 1 \leq u \leq rc\}$.
Output: Best variable partitions Ω^*, best pattern vectors V^*, best row-type vectors T^*.

1 Approximate function estimation $\hat{Y} = \{\hat{y}_{ku} | 1 \leq k \leq m, 1 \leq u \leq rc\} \leftarrow Y$;
2 **for** *round* l *from* 1 *to* R **do**
3 **for** k *from* m *downto* 1 **do**
4 Best NMED at the kth bit $E_k^* \leftarrow \infty$;
5 Variable partitions $\Phi \leftarrow$ *GenerVarPart*(P);
6 **for** *each variable partition* $\omega \in \Phi$ **do**
7 Pattern vector $V_k \leftarrow$ *SolveILP*$(Y, \hat{Y}, \omega)$;
8 Get corresponding row-type vector T_k and NMED E_k;
9 **if** $E_k < E_k^*$ **then**
10 $(E_k^*, \Omega_k^*, V_k^*, T_k^*) \leftarrow (E_k, \omega, V_k, T_k)$;
11 $\hat{Y}_k = \{\hat{y}_{ku} | 1 \leq u \leq rc\} \leftarrow$ *GetComponentFunction*$(\Omega_k^*, V_k^*, T_k^*)$;
12 $(\Omega^*, V^*, T^*) \leftarrow ((\Omega_1^*, \ldots, \Omega_M^*), (V_1^*, \ldots, V_M^*), (T_1^*, \ldots, T_M^*))$;
13 **return** (Ω^*, V^*, T^*);

3) Heuristic Acceleration Technique: Unfortunately, the variable number of the previous ILP-based methods increases exponentially with the bound set size, as $c = 2^b$. For a target function with many inputs, it is challenging to find an optimal approximation for it by the ILP-based methods. In this section, we propose a heuristic method to search for a good local optimal solution.

Same as the decomposition method with non-shared variable partition, we focus on finding an approximate component function $\hat{g}_k$ given that the rest component functions have been known. Thus,

equivalently, we work on the Boolean matrix and try to find a pair of pattern and row-type vectors (V, T) giving the smallest NMED. The basic idea of the heuristic method is that if one of the pattern vector V and the row-type vector T is fixed, we can identify the optimal choice for the other efficiently. If the pattern vector V is fixed, we update the row-type vector element-wise. To decide t_i $(1 \leq i \leq r)$, we try the 4 categories on the ith row, and set t_i as the category giving the smallest MED over the input patterns on row i of the Boolean matrix for $\hat{g}_k$, under the current V and current choice of the rest component functions. If the row-type vector T is fixed, we update the pattern vector element-wise. To decide v_j $(1 \leq j \leq c)$, we try the value $v_j = 0$ and $v_j = 1$, and set v_j as the value that gives the smallest MED over the input patterns on the jth column of the Boolean matrix for $\hat{g}_k$, under the current T and current choice of the rest component functions.

Thus, instead of searching for V and T simultaneously, our method will update V and T alternatively by fixing one and optimizing the other until they converge to fixed values. Then, we reach a local optimal solution. Since the final solution highly depends on the initial choice for V, our method tries different initial V's for Z times and finally returns the best local optimum.

After elaborating all the four approximate decomposition methods, we compare the complexity of them, as is shown in Table I. The joint decomposition method with shared variable partition has the most variables and constraints, so it suffers from a long runtime. The separate decomposition method, the joint decomposition method with non-shared variable partition, and the heuristic joint decomposition method have the same c unknown variables, and hence, they are more efficient.

TABLE I
COMPARISON OF DIFFERENT APPROXIMATE DECOMPOSITION METHODS.

Method	Separate	Joint (shared)	Joint (non-shared)	Joint (heuristic)
Objective	Eq. (3)	Eq. (6)	Eq. (10)	Eq. (10)
#variables	c	$m(c+4r)$	c	c
#constraints	$c+5r$	$m(c+5r+rc)$	$c+5r+4rc$	$-$

VI. EXPERIMENTAL RESULTS

In this section, we present the experimental results on DALTA.

A. Experimental Setup

All the LUT architectures are implemented in Verilog Hardware Description Language, where LUTs are implemented by RAMs consisting of D filp-flops. We use Synopsys Design Compiler [21] for synthesizing the architectures, mapping them into Nangate 45nm standard cell library [22], and measuring their areas and delays. The function of the architectures is verified by Synopsys VCS [21], and the power is evaluated by Synopsys PrimeTime [21]. In addition, we implement the approximate decomposition methods in C++. We use Gurobi [23] as the ILP solver. All the experiments are performed on a computer with an 8-core 3.8GHz AMD 3700X processor and a 16GB RAM.

DALTA is compared with two other architectures. The first is a rounding-based approximate LUT architecture. A q-bit rounding-based LUT architecture just rounds the q LSBs of the outputs and keeps the $(m-q)$ MSBs. The second is a state-of-the-art architecture, ApproxLUT [12]. For ApproxLUT, we re-implement the pattern selection algorithm proposed in [12] to decide the stored patterns in it. Its hardware part, consisting of an LUT, a pattern matching unit, and arithmetic units, is reproduced by Verilog.

We test 6 continuous functions and 4 non-continuous functions listed in Table II, where the continuous functions are from [12] and the non-continuous ones include 4 arithmetic circuits from AxBench [24]. To compute continuous functions with n-input and m-output LUTs, we quantize their inputs into n bits and outputs into m bits. Since some circuits in AxBench have too many inputs that are not suitable for LUT-based computing, we re-quantize their inputs into 16 bits and adjust the number of outputs correspondingly. Actually, the four functions from AxBench take two 8-bit operands x_1 and x_2 as inputs, and we stitch them together to form a 16-bit input x. In this way, the output function $f(x)$ is a piecewise function with many segments, and hence it is non-continuous. For all the experiments, we assume that the inputs are uniformly distributed.

The important parameters of our method are listed below. The variable partition limit is $P = 1000$. The iteration round in Algorithm 1 is $R = 5$. The number of different initial pattern vectors for the heuristic joint decomposition method is $Z = 30$. For the ILP solver Gurobi, we set the runtime limit to solve a single ILP problem as 3600s. If the limit is reached, Gurobi returns the current best solution with the smallest error.

TABLE II
BENCHMARKS USED IN THE EXPERIMENTS.

Continuous function	Domain	Range	Application
$\cos(x)$	$[0, \frac{\pi}{2}]$	$[0, 1]$	Geometry
$\tan(x)$	$[0, \frac{2\pi}{5}]$	$[0, 3.08]$	Geometry
$\exp(x)$	$[0, 3]$	$[0, 20.09]$	Financial
$\ln(x)$	$[1, 10]$	$[0, 1.61]$	Financial
$\mathrm{erf}(x)$	$[0, 3]$	$[0, 1]$	Statistics
$\mathrm{denoise}(x)$	$[0, 3]$	$[0, 0.81]$	Medical
Non-continuous function	**#input**	**#output**	**Application**
Brent-Kung	16	9	Arithmetic
Forwardk2j	16	16	Robotics
Inversek2j	16	16	Robotics
Multiplier	16	16	Arithmetic

B. Comparison of Different Decomposition Methods

Section V proposes four approximate decomposition methods: the separate decomposition method, the joint shared-partition decomposition method, the joint non-shared-partition decomposition method, and the joint heuristic method. We compare their performance on the 6 continuous functions listed in Table II. In this experiment, we quantize the functions into $n = 9$ inputs and $m = 9$ outputs. The free set size is 4 and the bound set size is $b = 5$, so the LUT sizes of the free-set and the bound-set functions are both 32. For the shared-partition method, there are so many variables (see Table I) that exploration of different variable partitions for it is impractical. Therefore, we only apply the shared-partition method once with the partition found by a heuristic approach.[1]

For the other three methods, we explore different variable partitions for them, and the number of partitions is controlled by the parameter $P = 1000$.

Table III compares the performance of different decomposition methods. All numbers are the averages over the 6 benchmarks in Table II. The last column reports the runtime of solving a single

[1]Specifically, it is a modified version of the heuristic method described in Section V-B3. The modification is letting the approximate component functions $\hat{g}_k$'s share the same row-type vector T. The variable partition returned by the modified method is used here.

TABLE III
PERFORMANCE OF VARIOUS DECOMPOSITION METHODS.

Method	NMED	Area	Route box area	Latency	Power	Runtime
Separate	1.93%	$7502\mu m^2$	$3013\mu m^2$	0.16ns	0.30mW	2.21s
Shared	**0.49%**	**4823**μm^2	**335**μm^2	0.16ns	**0.25mW**	3600s
Non-shared	0.68%	$7502\mu m^2$	$3013\mu m^2$	0.16ns	0.30mW	4.38s
Heuristic	0.70%	$7502\mu m^2$	$3013\mu m^2$	0.16ns	0.30mW	**0.6ms**

ILP problem under a fixed variable partition. There are three observations from the table. First, the shared-partition method produces approximate functions with the smallest NMED. The reason is that the shared-partition method directly minimizes NMED by taking all the outputs into account, which can avoid inducing errors on the most significant output bits. Second, the shared-partition method requires smaller area and power than the others, and all the methods have the same latency. It is reasonable since the shared-partition method allows sharing a routing box over different LUT pairs in DALTA. However, the other methods need different partitions on the 9 outputs, and 9 copies of the routing boxes are needed. Third, the heuristic method is the most efficient and does not induce a large error, so it is preferred especially for more complex functions with larger input and output numbers.

C. Experiments on Continuous Functions

The benchmarks used in this experiment are the 6 continuous functions in Table II. Different from Section VI-B, we quantize the inputs and outputs into $n = m = 16$ bits. In this case, only the heuristic joint decomposition method can obtain solutions in a reasonable time (i.e., within a few hours), and we choose it to generate approximate decompositions. The free set size is 7 and the bound set size is $b = 9$. Thus, the LUT sizes of the free-set and the bound-set functions are 256 and 512, respectively.

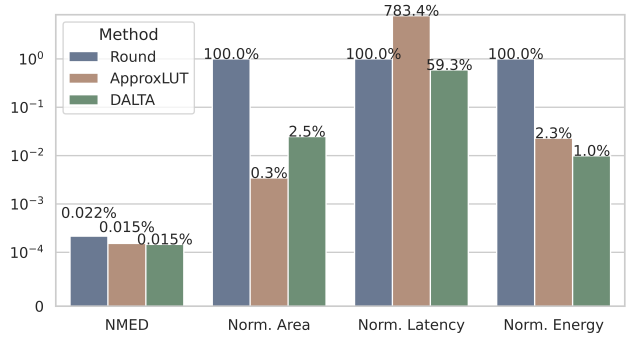

Fig. 5. Comparison of different approximate LUT architectures on continuous functions.

We compare DALTA with ApproxLUT and the rounding-based LUT architecture. For ApproxLUT, we tune the number of stored patterns to make its NMED the same as that of DALTA. Similarly, for rounding-based LUT architecture, we adjust the number of rounding bits so that its NMED is slightly larger than that of DALTA. Fig. 5 compares the average NMEDs, areas, latencies, and energies over the 6 functions for the rounding architecture, ApproxLUT, and DALTA. We measure the latency and the energy for a single reading operation. For area, latency, and energy, the values are normalized to that of the rounding architecture. As shown in Fig. 5, the average NMEDs of the rounding architecture, ApproxLUT, and DALTA are $0.022\%, 0.015\%$, and 0.015%, respectively. In other words, the errors mainly distribute on the 4 LSBs. Compared to the rounding architecture, DALTA reduces area, latency, and energy by 97.5%, 40.7%, and 99%, respectively. Compared to ApproxLUT, DALTA improves latency and energy by 92.4% and 56.5%, respectively, but requires $8\times$ area.

D. Experiments on Non-continuous Functions

The benchmarks used in this experiment are the 4 non-continuous functions in Table II. They all have $n = 16$ inputs, but their output numbers m are different. The free set size is 7 and the bound set size is $b = 9$. We use the heuristic joint decomposition method to decompose these functions.

First, we show that ApproxLUT is not suitable to approximate non-continuous functions. We tune the number of stored patterns in ApproxLUT so that the LUT sizes of ApproxLUT and DALTA are the same. Notably, DALTA consumes less energy than ApproxLUT when they have the same LUT size, since ApproxLUT requires additional pattern matching and arithmetic units. The LUT size of DALTA is $(2^b+2^{n-b+1})m = 768m$, while that of ApproxLUT is $3sm$, where s is the number of stored patterns. We select $s = 256$, so the LUT size of both architectures are the same. Table IV compares the NMEDs between DALTA and ApproxLUT with the same LUT size for each non-continuous function. We can see that DALTA can approximate non-continuous functions with smaller NMED, while ApproxLUT suffers from extremely large error. The reason is that ApproxLUT utilizes first-order Taylor expansion to approximate functions, and it does not work on non-continuous functions.

TABLE IV
COMPARISON OF NMEDS OF DALTA AND APPROXLUT WITH THE SAME LUT SIZE.

Benchmark	NMED		LUT size in bits	
	DALTA	ApproxLUT	DALTA	ApproxLUT
Brent-Kung	**0.0978%**	84.8722%	6912	6912
Forwardk2j	**0.8955%**	61.3607%	12288	12288
Inversek2j	**0.6068%**	13.4480%	12288	12288
Multiplier	**0.6548%**	41.9141%	12288	12288

Now, we compare the performance of DALTA on non-continuous functions with the rounding architecture. Similar as in Section VI-C, we adjust the number of rounding bits so that the rounding architecture has slightly larger NMED than DALTA. Fig. 6 compares the average NMEDs, areas, latencies, and energies over the 4 functions for the two architectures. Again, we measure the latency and the energy for a single reading operation. For area, latency, and energy, the values are normalized to that of the rounding architecture. As shown in Fig. 6, the average NMEDs over the 4 functions of the rounding architecture and DALTA are 0.81% and 0.56%, respectively. Compared to the rounding architecture, DALTA improves the area, latency, and energy by 95.8%, 39.0%, and 98.3%, respectively.

E. Exploration of Different Structures of DALTA

To show the exponential decrease on area and power with the input numbers, we apply the heuristic method to generate approximations for the $\cos(x)$ function targeting at different configurations of DALTA with different input and output numbers. For an n-input function, the free set size is $\lfloor \frac{n}{2} \rfloor$, and the bound set size is $\lceil \frac{n}{2} \rceil$. Table V shows the performance of the heuristic method for different DALTA configurations. The cost (area, power, and delay) improvement is

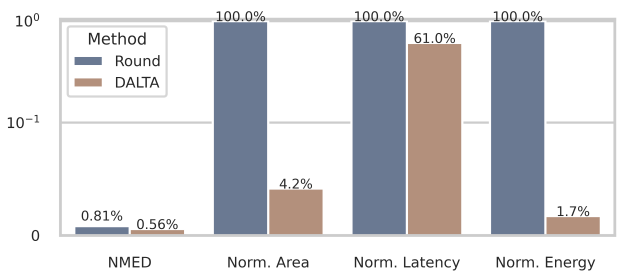

Fig. 6. Comparison of DALTA and the rounding architecture on non-continuous functions.

TABLE V
PERFORMANCE OF HEURISTIC METHOD FOR DIFFERENT CONFIGURATIONS OF DALTA FOR THE $\cos(x)$ FUNCTION.

#In/#Out	NMED	Area impr.	Latency impr.	Power impr.	Runtime/s
8/16	0.230%	3.0×	1.7×	4.5×	3
9/16	0.121%	4.3×	1.9×	6.4×	6
10/16	0.124%	6.2×	1.8×	9.2×	25
11/16	0.052%	9.4×	2.2×	13.9×	225
12/16	0.060%	13.6×	2.4×	20.2×	546
13/16	0.028%	20.7×	2.6×	30.8×	1036
14/16	0.027%	29.2×	2.5×	43.3×	2212
15/16	0.013%	45.0×	2.7×	66.7×	3622
16/16	0.014%	62.4×	1.6×	92.5×	6367

calculated as the cost of accurate LUT over that of DALTA. As the input and output numbers increase, the NMED roughly decreases, the area improvement, power improvement, and the runtime increase exponentially. It is not surprising because our method enjoys exponential decrease of LUT size over the accurate LUT. The reason of the long runtime is that as the input number increases, the numbers of rows and columns in Boolean matrices grow exponentially. From Section V-B3, the row-type and pattern vectors are both updated element-wise. Thus, the runtime shows an exponential growth.

VII. CONCLUSION

This paper proposes DALTA, a decomposition-based approximate LUT architecture. The main idea is to decompose a function approximately into bound-set and free-set functions, and store them with LUTs of smaller sizes. To map an arbitrary function into DALTA, we formulate the approximate decomposition as an optimization problem, and propose ILP-based and heuristic solutions. The experimental results show that DALTA significantly improves the area, delay, and power by introducing a small error.

Our future work will further improve the efficiency of the approximate decomposition approaches by resorting to the BDD-based decomposition method [16], as BDD is a more compact representation than Boolean matrix. Furthermore, we plan to explore the space of free and bound variables more efficiently by the cuts of BDDs, like the approach in [16].

REFERENCES

[1] M. M. Waldrop, "The chips are down for Moore's law," *Nature*, vol. 530, no. 7589, pp. 144–147, 2016.

[2] P. T. P. Tang, "Table-lookup algorithms for elementary functions and their error analysis," in *International Symposium on Computer Arithmetic*, 1991, pp. 232–236.

[3] J. Cong *et al.*, "Energy-efficient computing using adaptive table lookup based on nonvolatile memories," in *International Symposium on Low Power Electronics and Design*, 2013, pp. 280–285.

[4] Q. Xu *et al.*, "Approximate computing: A survey," *IEEE Design and Test*, vol. 33, no. 1, pp. 8–22, 2016.

[5] M. J. Schulte and J. E. Stine, "Symmetric bipartite tables for accurate function approximation," in *International Symposium on Computer Arithmetic*, 1997, pp. 175–183.

[6] J. E. Stine and M. J. Schulte, "The symmetric table addition method for accurate function approximation," *Journal of VLSI Signal Processing*, vol. 21, no. 2, pp. 167–177, 1999.

[7] J.-M. Muller, "A few results on table-based methods," *Reliable Computing*, vol. 5, no. 3, pp. 279–288, 1999.

[8] F. de Dinechin and A. Tisserand, "Multipartite table methods," *IEEE Transactions on Computers*, vol. 54, no. 3, pp. 319–330, 2005.

[9] S. Hsiao *et al.*, "Hierarchical multipartite function evaluation," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 89–99, 2017.

[10] A. Rahimi, "Approximate associative memristive memory for energy-efficient gpus," in *Design, Automation and Test in Europe*, 2015, pp. 1497–1502.

[11] M. Imani *et al.*, "Resistive configurable associative memory for approximate computing," in *Design, Automation and Test in Europe*, 2016, pp. 1327–1332.

[12] Y. Tian *et al.*, "ApproxLUT: A novel approximate lookup table-based accelerator," in *International Conference on Computer-Aided Design*, 2017, pp. 438–443.

[13] R. L. Ashenhurst, "The decompositions of switching functions," in *International Symposium on the Theory of Switching Functions*, 1959, pp. 74–116.

[14] H. A. Curtis, *A New Approach to the Design of Switching Circuits*. Van Nostrand, 1962.

[15] J. P. Roth and R. M. Karp, "Minimization over boolean graphs," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 227–238, 1962.

[16] Y.-T. Lai *et al.*, "BDD based decomposition of logic functions with application to fpga synthesis," in *Design Automation Conference*, 1993, pp. 642–647.

[17] R.-R. Lee, J.-H. R. Jiang, and W.-L. Hung, "Bi-decomposing large boolean functions via interpolation and satisfiability solving," in *Design Automation Conference*, 2008, pp. 636–641.

[18] V. Shen *et al.*, "An algorithm for the disjunctive decomposition of switching functions," *IEEE Transactions on Computers*, vol. 100, no. 3, pp. 239–248, 1970.

[19] Y. Yao *et al.*, "Approximate disjoint bi-decomposition and its application to approximate logic synthesis," in *International Conference on Computer Design*, 2017, pp. 517–524.

[20] B. Kolman and R. E. Beck, *Elementary Linear Programming with Applications*. Academic Press, 1995.

[21] Synopsys, Inc., *Synopsys softwares*, 2021. [Online]. Available: http://www.synopsys.com.

[22] Nangate, Inc., *Nangate 45nm open cell library*, 2021. [Online]. Available: https://si2.org/open-cell-library/.

[23] Gurobi, *Gurobi - the fastest solver*, 2021. [Online]. Available: https://www.gurobi.com/.

[24] A. Yazdanbakhsh *et al.*, "AxBench: A multi-platform benchmark suite for approximate computing," in *IEEE Design and Test*, 2016.