RADAR

Oxford Brookes University – Research Archive and Digital Asset Repository (RADAR)

Jabir, A

A technique for representing multiple-output binary functions with applications to verfication and simulation.

Jabir, A, Pradham, D, Singh, A and Rajaprabhu, T L (2007) A technique for representing multiple-output binary functions with applications to verfication and simulation. *IEEE transactions on computers*, 56 (8). pp. 1133-1145.

Doi: 10.1109/TC.2007.1056

This version is available: http://radar.brookes.ac.uk/radar/items/0bda0d6e-8f0c-e9e1-f055-7e57cc4ee51d/1/
Available in the RADAR: September 2010
Copyright © and Moral Rights are retained by the author(s) and/ or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This item cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder(s). The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

This document is the published version of the journal article.

www.brookes.ac.uk/go/radar

OXFORD BROOKES UNIVERSITY

Directorate of **Learning Resources**

A Technique for Representing Multiple-Output Binary Functions with Applications to Verification and Simulation

Abusaleh M. Jabir, *Member*, *IEEE*, Dhiraj K. Pradhan, *Fellow*, *IEEE*,
Ashutosh Kumar Singh, and T.L. Rajaprabhu

Abstract—This paper presents a technique for representing multiple-output binary and word-level functions in $\mathrm{GF}(N)$ (where $N = p^m$, p is a prime number, and m is a nonzero positive integer) based on decision diagrams (DDs). The presented DD is canonical and can be made minimal with respect to a given variable order. The DD has been tested on benchmarks, including integer multiplier circuits, and the results show that it can produce better node compression (more than an order of magnitude in some cases) compared to shared binary DDs (BDDs). The benchmark results also reflect the effect of varying the input and output field sizes on the number of nodes. Methods of graph-based representation of characteristic and encoded characteristic functions in $\mathrm{GF}(N)$ are also presented. Performance of the proposed representations has been studied in terms of average path lengths and the actual evaluation times with 50,000 randomly generated patterns on many benchmark circuits. All of these results reflect that the proposed technique can outperform existing techniques.

Index Terms—Finite or Galois fields, decision diagrams, characteristic and encoded characteristic functions, evaluation, simulation, verification.

1 Introduction

The motivation of this paper is the efficient graph-based representation of multiple-output binary and word-level functions in a finite field for verification and simulation.

1.1 Previous Work

Finite fields have numerous applications in public-key cryptography [19] to encounter channel errors and for protection of information, error control codes [24], and digital signal processing [2]. Finite fields gained significance with the practical lucrativeness of the elliptic-curve cryptosystems. The role of finite fields in error control systems is well established and contributes to many fault-tolerant designs. In the EDA industry, the role of multivalued functions, especially in the form of Multivalued Decision Diagrams (MDDs), is well described in [7], [20]. Word-level diagrams can be useful in high-level verification, logic synthesis [3], [4], and software synthesis [25]. Multivalued functions can also be represented in finite fields, as shown in [15]. Finite fields can represent many arithmetic circuits very efficiently [10]. Also, there are fine-grain FPGA structures for which arithmetic circuits in finite fields seem to be highly efficient. The varied use of finite fields leads to designing high-speed low-complexity systolic VLSI realizations [5]. Fast functional simulation in the design cycles is a key step in all of these applications [14].

Most existing techniques for word-level representation, for example, [12], [16], are not capable of efficiently representing arbitrary combinations of bits or nibbles, that is, subvectors, within a word. The proposed framework for representing circuits can deal with these types of situations by treating each subvector as a word-level function in $\mathrm{GF}(2^m)$, where m denotes the number of bits within a subvector. The word-level functions are then represented as canonic word-level graphs. Hence, the proposed technique offers a generalized framework for verifying arbitrary combinations of bits or words.

Another situation where existing word-level techniques seem to have difficulty is in representing *nonlinear* design blocks such as comparators at the RTL (for example, in the integer domain). The proposed framework does not suffer from this critical shortcoming.

As an example of representing any arbitrary combination of output bits in a multiple-output function, let us consider a 4-input, 8-output binary function. The MSB, $f = \sum m(10, 11, 12, 13, 14, 15)$,[1] and the LSB,

$$g = \sum m(4, 5, 6, 7, 8, 9, 10, 11, 14, 15),$$

- *A.M. Jabir is with the School of Technology, Oxford Brookes University, Wheatley Campus, Oxford OX33 1HX, UK. E-mail: ajabir@brookes.ac.uk.*
- *D.K. Pradhan is with the Department of Computer Science, University of Bristol, Bristol BS8 1UB, UK. E-mail: pradhan@cs.bris.ac.uk.*
- *A.K. Singh is with the Department of Computer Science, Curtin University of Technology, Sarawak Campus Miri, Malayasia. E-mail: ashutosh.s@curtin.edu.my.*
- *T.L. Rajaprabhu is with the School of Electrical and Computer Engineering, Georgia Institute of Technology, 777 Atlantic Drive NW, Atlanta, GA 30332-0250. E-mail: raja@cc.gatech.edu.*

Manuscript received 22 May 2006; accepted 12 Jan. 2007; published online 20 Apr. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0196-0506.
Digital Object Identifier no. 10.1109/TC.2007.1056.

1. The notation $h = \sum m(p_1, p_2, \ldots, p_q)$ is used to represent the truth table of a function, where each p_r $(1 \leq r \leq q)$ is the decimal equivalent of a row in the input part of the table, with an output of 1; that is, each p_r is a $minterm$ from the ON set in its decimal form.

0018-9340/07/$25.00 © 2007 IEEE Published by the IEEE Computer Society

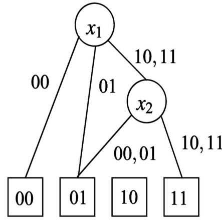

Fig. 1. Representing two bits simultaneously.

can be represented on the same diagram, as shown in Fig. 1 [1]. The BDD-based representation of this circuit will require a larger number of nodes.

Although research has been done on representing circuits in finite fields [10], [17], the theoretical basis was carried out, for example, in the spectral domain for a fixed value, for example, 4 in [17]. Unlike these techniques, this paper presents the generalized framework for the design, verification, and simulation of circuits in finite fields based on the MDD-like graph-based form. The proposed decision diagram (DD) has advantages over other diagrams such as [16] in that, in addition to applications in multiple-valued algebra, it is not restricted to word boundaries, but, instead, can be used to represent and verify any arbitrary combination of output bits. Unlike [6], which is not a DD and, hence, lacks many features present in a DD, the proposed diagram does not have such shortcomings. Also, unlike [7], the proposed DD represents finite fields and extension fields, whereas [7] is based on the MIN-MAX post algebra.

Owing to its canonicity, the proposed technique can be used for verifying circuits at the bit or word level by checking for graph isomorphism, which can be done very quickly.

Fast evaluation times of multiple-output functions is significant in the areas of logic simulation, testing, satisfiability, and safety checking [9], [13]. The proposed DDs also offer much shorter average path lengths (APLs) and, hence, evaluation times [22] compared to shared BDDs, with a varying trade-off between evaluation times and spatial complexity.

1.2 Background and Notation

Let $\mathrm{GF}(N)$ denote a set of N elements, where $N = p^m$, p is a prime number, and m is a nonzero positive integer, with two special elements 0 and 1 representing the additive and multiplicative identities, respectively, and two operators, addition "+" and multiplication "·". $\mathrm{GF}(N)$ defines a finite field, also known as a *Galois field*, if it forms a *commutative ring* with identity over these two operators in which every element has a multiplicative inverse. $\forall a \in \mathrm{GF}(N)$, $\exists -a \in \mathrm{GF}(N)$ such that $a - a = 0$. Similarly, $\forall b \in \mathrm{GF}(N)$ and $b \neq 0$, $\exists b^{-1} \in \mathrm{GF}(N)$ such that $b \cdot b^{-1} = 1$. Here, p is called the characteristic of the field and satisfies the following:

$$\underbrace{1 + 1 + \cdots + 1}_{p \text{ times}} = 0$$

and $pa = 0, \forall a \in \mathrm{GF}(N)$. Also, $\forall a \in \mathrm{GF}(N)$, $a^N = a$, and, for $a \neq 0$, $a^{N-1} = 1$. The elements of $\mathrm{GF}(N)$ can be represented as polynomials over $\mathrm{GF}(p)$ of degree at most $n-1$. Additional properties of $\mathrm{GF}(N)$ can be found in [15], [24].

The following notation is used in this paper:

Let $I_N = \{0, 1, \ldots, N-1\}$ and let $\delta : I_N \to GF(N)$ be a one-to-one mapping, with $\delta(0) = 0$.

Let $f|_{x_k=y}$, called the *cofactor* of f with respect to $x_k = y$, represent the fact that all occurrences of x_k within f are replaced with y, that is, $f|_{x_k=y} = f(x_1, x_1, \ldots, x_k = y, \ldots, x_n)$. The notation $f|_{x_i=y_i, x_{i+1}=y_{i+1}, \ldots, x_{i+j}=y_{i+j}}$ (or just $f|_{y_i, y_{i+1}, \ldots, y_{i+j}}$ when the context is clear) will be used to represent the replacement of variables $x_i, x_{i+1}, \ldots, x_{i+j}$ with the values $y_i, y_{i+1}, \ldots, y_{i+j}$, respectively.

We shall use the notation $|A|$ to represent the total number of nodes in a graph A.

We have the following in $\mathrm{GF}(N)$.

Theorem 1 [15]. *A function $f(x_1, x_2, \ldots, x_k, \ldots, x_n)$ in $GF(N)$ can be expanded as follows:*

$$f(x_1, x_2, \ldots, x_k, \ldots, x_n) = \sum_{e=0}^{N-1} g_e(x_k) f|_{x_k=\delta(e)}, \qquad (1)$$

where $g_e(x_k) = 1 - [x_k - \delta(e)]^{N-1}$.

In Theorem 1, $g_e(x_k)$ is called a *multiple-valued literal*[2] in $\mathrm{GF}(N)$. Theorem 1 is known as the *literal-based expansion* of the functions in $\mathrm{GF}(N)$. It can be shown by setting $N = 2$ that Theorem 1 reduces to the Shannon's expansion in GF(2).

The product of literals is called a *product term* or just a *product*.

Two product terms are said to be *disjoint* if their product in $\mathrm{GF}(N)$ equates to zero.

An expression in $\mathrm{GF}(N)$ constituting product terms is said to be disjoint if all of its product terms are pairwise disjoint.

This paper is organized as follows: Section 2 presents the theory behind the graph-based representation and its reduction, with methods for additional node and path optimizations. Section 3 provides the theory behind representing functions in $\mathrm{GF}(N)$ in terms of graph-based characteristic function (CF) and encoded CF (ECF). The proposed methods offer much shorter evaluation times than existing approaches and Section 4 provides a technique for calculation of the APLs for approximating the evaluation times for the proposed representations. The proposed technique has been tested on many benchmark circuits. Finally, in Section 5, we present the experimental results.

2 Graph-Based Representation

Any function in $\mathrm{GF}(N)$ can be represented by means of an MDD-like [7] data structure. However, unlike traditional MDDs, which are used to represent functions in the MIN-MAX post algebra, the algebra of finite fields needs to be considered. Although an MDD type of data structure has been used for representing functions in finite fields in [17], the underlying mathematical framework was considered for GF(4) only: No generalization was proposed for higher order fields and their extensions and no experimental results were reported, even though it was reported that

2. The term "literal" was chosen because, in GF(2), this expression reduces to the traditional Boolean literal, that is, it represents a variable or its complement (inverse).

generalization can be made. Also, no technique seems to exist which can further optimize an MDD-like representation of functions in $\mathrm{GF}(N)$ by zero terminal node suppression and normalization. It should be noted that the technique in [7] has used a type of edge negation based on modular arithmetic. However, modular arithmetic in the form considered in [7] does not naturally comply with extension fields. Since an MDD has been defined in terms of functions in the MIN-MAX post algebra, to distinguish between these two algebras, the MDD-like representation of functions in finite fields will be called *Multiple-Output Decision Diagrams* (MODDs). Hence, traditional MDDs result in a post-algebraic MIN-MAX SOP form, whereas, with the MODD, a canonic polynomial expression in $\mathrm{GF}(N)$ can be obtained. As an example, the MODD in Fig. 5a, which represents a 4-valued function with values in $\{0, 1, \alpha, \beta\}$ (assuming that $\alpha = 2$ and $\beta = 3$), yields the following expression in the MIN-MAX post algebra:

$$\begin{aligned}
&f(x_1, x_2, x_3) = \\
&\beta(x_1^{\beta} x_2^{\beta} \vee x_1^{\{1,\alpha\}} x_2^{\{1,\alpha,\beta\}} x_3^{\beta} \vee x_1^{\beta} x_2^{\{1,\alpha\}} x_3^{\beta} \vee x_1^{0} x_2^{\{1,\alpha\}} x_3^{\beta}) \vee \\
&\alpha(x_1^{\beta} x_2^{\{1,\alpha\}} x_3^{\alpha} \vee x_1^{\{1,\alpha\}} x_2^{\{1,\alpha,\beta\}} x_3^{\alpha} \vee x_1^{0} x_2^{\{1,\alpha\}} x_3^{\alpha}) \vee \\
&(x_1^{\beta} x_2^{\{1,\alpha\}} x_3^{1} \vee x_1^{\{1,\alpha\}} x_2^{\{1,\alpha,\beta\}} x_3^{1} \vee x_1^{0} x_2^{\{1,\alpha\}} x_3^{1}).
\end{aligned}$$

Here, the symbol "$\vee$" has been used to denote MAX. MIN is denoted by the product-like notation. The expression x_i^S, where $S \subseteq \{0, 1, \alpha, \beta\}$, is a literal defined in the MIN-MAX post algebra as $x_i^S = MAX_VALUE$, where $MAX_VALUE = \beta$ in this case if $x_i \in S$; otherwise, $x_i^S = 0$. In contrast, $x_i^S = 1$ if $x_i \in S$; otherwise, $x_i^S = 0$ in $\mathrm{GF}(N)$ (Theorem 1). The following multivariate polynomial results from the MODD in GF(4) by application of Theorem 1 followed by expansion and rearranging the terms:

$$\begin{aligned}
f(x_1, x_2, x_3) = {} & \beta x_1^3 x_2^3 + \alpha x_1^2 x_2^3 + x_1 x_2^3 + \alpha x_1^3 x_2^2 + x_1^2 x_2^2 + \beta x_1 x_2^2 \\
& + x_3 x_2 + \beta x_1^2 x_2 + \alpha x_1 x_2 + \beta x_2^2 x_3 \\
& + \alpha x_2 x_3 + \beta x_1^2 x_2^3 x_3 + \alpha x_1 x_2^3 x_3 + \alpha x_1^2 x_2^2 x_3 \\
& + x_1 x_2^2 x_3 + x_1^2 x_2 x_3 + \beta x_1 x_2 x_3.
\end{aligned}$$

Definition 1 (Decision Diagram). *A decision diagram in* $\mathrm{GF}(N)$ *is a rooted directed acyclic graph with a set of nodes* V *containing two types of nodes: One is a set of* N *terminal nodes or leaves with outdegree zero, each one labeled with a* $\delta(s)$ *and* $s \in I_N$. *Each terminal node* u *is associated with an attribute* $value(u) \in \mathrm{GF}(N)$. *The other is a set of nonterminal nodes, with outdegree of* N. *Each nonterminal node* v *is associated with an attribute* $var(v) = x_i$ *and* $1 \le i \le n$ *and another attribute* $child_j(v) \in V$, $\forall j \in I_N$, *which represents each of the children (direct successor) of* v.

The correspondence between a function in $\mathrm{GF}(N)$ and a MODD in $\mathrm{GF}(N)$ can be defined as follows:

Definition 2 (Recursive Expansion). *An MODD in* $\mathrm{GF}(N)$ *rooted at* v *denotes a function* f^v *in* $\mathrm{GF}(N)$ *defined recursively as follows: 1) If* v *is a terminal node, then* $f^v = value(v)$, *where* $value(v) \in \mathrm{GF}(N)$. *2) If* v *is a nonterminal node, with* $var(v) = x_i$, *then* f^v *is the function:*

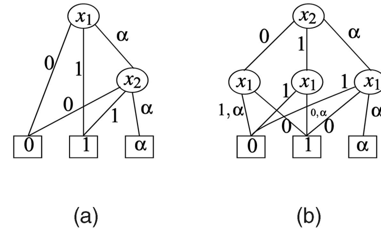

Fig. 2. Effect of variable ordering.

$$f^v(x_1, x_2, \ldots, x_i, \ldots, x_n) = \sum_{e=0}^{N-1} g_e(x_i) f^{child_e(v)},$$

where $g_e(x_i) = 1 - [x_i - \delta(e)]^{N-1}$.

Each variable x_i and $1 \le i \le n$ in an MODD is associated with one or more nodes which appear at the same *level* in the MODD. More precisely, the nodes associated with variable x_i correspond to level $(i-1)$ and vice versa. Therefore, level i, corresponding to variable x_{i+1}, can contain at most N^i nodes. Hence, the root of the MODD contains exactly one node and the level before the external nodes can contain at most N^{n-2} nodes.

Example 1. Let us consider the MODD shown in Fig. 2a. This MODD represents the function in GF(3), $f(x_1, x_2) = g_1(x_1) + g_2(x_1) g_1(x_2) + \alpha \cdot g_2(x_1) g_2(x_2)$, where $g_r(x_s) = 1 - [x_s - \delta(r)]^2$.

Here, both levels 0 and 1, corresponding to the variables x_1 and x_2, respectively, contain exactly one node each.

Lemma 1. *Theorem 1 results in a disjoint expression, that is, the product terms in (1) are mutually (pairwise) disjoint.*

Proof. In (1), $g_e(x_k) = 1$ iff $x_k = \delta(e)$. For all other values of x_k, $g_e(x_k) = 0$. Let us consider any two literals $g_r(x_k)$ and $g_s(x_k)$ such that $r \neq s$. Two cases may arise.

Case 1. $x_k \neq \delta(r)$ and $x_k \neq \delta(s)$. In this case, both $g_r(x_k)$ and $g_s(x_k)$ will equate to 0. Therefore, $g_r(x_k) \cdot g_s(x_k) = 0$.

Case 2. Either $x_k = \delta(r)$ or $x_k = \delta(s)$, but not both. If $x_k = \delta(r)$, then $g_r(x_k) = 1$ and $g_s(x_k) = 0$; otherwise, $g_r(x_k) = 0$ and $g_s(x_k) = 1$. Therefore, $g_r(x_k) \cdot g_s(x_k) = 0$.

Hence, the proof follows. □

Lemma 1 yields the following:

Theorem 2. *Each path from the root node to a nonzero terminal node in a MODD represents a disjoint product term in* $\mathrm{GF}(N)$.

Example 2. Let us consider the MODD in Fig. 2a representing a function in GF(3). The path $\alpha, 1$ represents the product term $X = g_\alpha(x_1) g_1(x_2)$. The path α, α represents the product term $Y = g_\alpha(x_1) g_\alpha(x_2)$. Clearly, X and Y are disjoint because $X \cdot Y = 0$ as $g_1(x_2) \cdot g_\alpha(x_2) = 0$.

2.1 Reduction

We have the following from Theorem 1:

Corollary 2.1. *In (1), if* $\forall i, j \in I_N$ *and* $i \neq j$ $f|_{x_k=\delta(i)} = f|_{x_k=\delta(j)}$, *then* $f = f|_{x_k=\delta(0)} = f|_{x_k=\delta(1)} = \cdots = f|_{x_k=\delta(N-1)}$.

Proof. The proof is by perfect induction. Let

$$h = f|_{x_k=\delta(0)} = f|_{x_k=\delta(1)} = \cdots = f|_{x_k=\delta(N-1)}.$$

Then, (1) becomes $f = h\sum_{e=0}^{N-1} g_e(x_k)$.

For any $a \in \mathrm{GF}(N)$ such that $a \neq 0$, $a^{N-1} = 1$. Now, in Theorem 1, if $x_k = \delta(r)$, then $g_r(x_k) = 1$ for $r \in I_N$. Furthermore, $g_s(x_k) = 0$, $\forall s \in I_N$, and $s \neq r$. Therefore, $\sum_{e=0}^{N-1} g_e(x_k)$ becomes 1, which implies that $f = h$ and, hence, the proof. □

Based on the above, an MODD can be reduced as outlined in the following.

Reduction rules. There are two reduction rules: 1) If all of the N children of a node v point to the same node w, then delete v and connect the incoming edge of v to w (this follows from Corollary 2.1). 2) Share equivalent subgraphs.

A DD in $\mathrm{GF}(N)$ is said to be ordered if the expansion in (1) is recursively carried out in a certain linear variable order such that, on all of the paths throughout the graph, the variables also respect the same linear order.

A DD in $\mathrm{GF}(N)$ is said to be reduced iff 1) there is no node $u \in V$ such that, $\forall i$, $j \in I_N$, and $i \neq j$, with $\mathrm{child}_i(u) = \mathrm{child}_j(u)$, and 2) there are no two distinct nodes $u, v \in V$ that have the same variable names and the same children, that is, $var(u) = var(v)$ and $\mathrm{child}_i(u) = \mathrm{child}_i(v)$ $\forall i \in I_N$ implies $u = v$.

We have the following from the definition of the MODD.

Lemma 2. *For any node v in a reduced DD in $\mathrm{GF}(N)$, the subgraph rooted at v is itself reduced.*

Canonicity. A reduced ordered MODD in $\mathrm{GF}(N)$ based on the expansion of Theorem 1 is canonical up to isomorphism. This is presented in the following, which can be proved by generalizing the canonicity of the BDD [3].

Theorem 3. *For any n variable function $f(x_1, x_2, \ldots, x_n)$ in $\mathrm{GF}(N)$, there is exactly one reduced ordered DD in $\mathrm{GF}(N)$ with respect to a specific variable order, which is minimal with respect to the reduction rules.*

A reduction algorithm. A reduction algorithm for MODD appears in Fig. 3. In order to share equivalent subgraphs, subgraphs already present in the MODD are placed in a table. In lines 4, 6, 12, and 14, we have assumed that checking for membership and addition of an element to such a table (HT) can be carried out in constant times, for example, by using a hash table. Hence, the complexity of the algorithm is $O(|G|)$ since each node can be made to be visited just once during the reduction process, where G is an MODD of a function before the reduction.

However, an algorithm for the creation of MODDs from the functional description in $\mathrm{GF}(N)$ will have a complexity $O(N^n)$ in the worst case. The algorithm for the creation of MODDs can be derived from (1). The efficiency of the algorithm can be improved, in general, by noting that, during the recursive expansion with respect to each variable, certain variables may not appear for further recursive calls. Therefore, the recursion tree need not be expanded in the direction of a variable which does not appear. The efficiency can be further improved by

```
1   Algorithm ReduceDD(v : node) : node;
2   begin
3       if v is a terminal node, then
4           if IsIn(v, HT), then return LookUp(HT, v)
5           else begin
6               HT := insert(v, HT); (* Initially HT = ∅ *)
7               return v
8           end;
9       else begin
10          v'_i := ReduceDD(child_i(v)), ∀i ∈ I_N;
11          if v'_j = v'_k, ∀j, k ∈ I_N and j ≠ k, then return v'_0
12          else if IsIn(v, HT), then return LookUp(HT, v)
13          else begin
14              HT := insert(v, HT);
15              return v
16          end
17      end
18  end;
```

Fig. 3. A reduction algorithm for MODDs in $\mathrm{GF}(N)$.

incorporating Lemma 3 based on a dynamic programming-like approach, as discussed in Section 2.3, which has been done for the experimental results in Section 5. In this case, the network in $\mathrm{GF}(N)$ is traversed in topological order from the inputs to the outputs and Lemma 3 is applied iteratively.

Note that the size of a reduced MODD heavily depends on the variable ordering, as in any other DD [3], [7]. The depth of an MODD is $O(n)$ in the worst case since each variable appears once at each level in the worst case.

2.2 Variable Reordering

The size, that is, the number of nodes, of an MODD depends on the order of the variables during its construction. For example, the MODD in Fig. 2a represents a function in GF(3) under the variable order (x_1, x_2). Fig. 2b shows the same function in GF(3), but it is under the variable order (x_2, x_1). Clearly, Fig. 2a contains fewer nodes than Fig. 2b. Given an n variable function in $\mathrm{GF}(N)$, the size of the solution space for finding the best variable order is $O(n!)$, which is impractical for large values of n. Hence, a heuristic level-by-level swap-based algorithm is considered in this paper.

The theory behind variable reordering in $\mathrm{GF}(N)$ is based on Theorem 1 as follows: Without loss of generality, let us assume that variables x_1 and x_2 are to be swapped. From Theorem 1, we have

$$\begin{aligned} &f(x_1, x_2, \ldots, x_n) = \\ &g_0(x_1)\Big(g_0(x_2)f|_{0,0} + g_1(x_2)f|_{0,1} + \cdots + g_{N-1}(x_2)f|_{0,N-1}\Big) \\ &+ g_1(x_1)\Big(g_0(x_2)f|_{1,0} + g_1(x_2)f|_{1,1} + \cdots + g_{N-1}(x_2)f|_{1,N-1}\Big) \\ &+ \cdots \\ &+ g_{N-1}(x_1)\Big(g_0(x_2)f|_{N-1,0} + g_1(x_2)f|_{N-1,1} + \cdots \\ &+ g_{N-1}(x_2)f|_{N-1,N-1}\Big). \end{aligned}$$

If x_1 and x_2 are swapped, then, again from Theorem 1, the function f_s results in

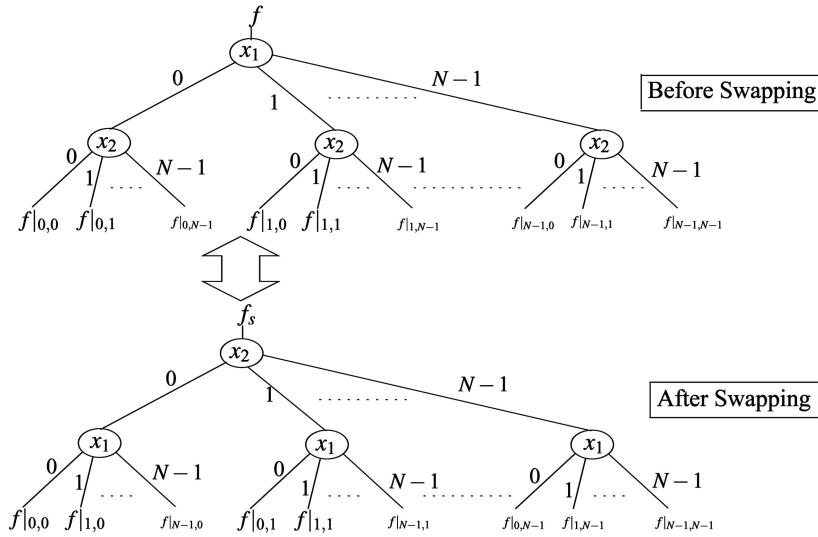

Fig. 4. The theory behind reordering.

$$
\begin{aligned}
&f_s(x_1, x_2, \ldots, x_n) = \\
&g_0(x_2)\Big(g_0(x_1)f|_{0,0} + g_1(x_1)f|_{1,0} + \cdots + g_{N-1}(x_1)f|_{N-1,0}\Big) \\
&+ g_1(x_2)\Big(g_0(x_1)f|_{0,1} + g_1(x_1)f|_{1,1} + \cdots + g_{N-1}(x_1)f|_{N-1,1}\Big) \\
&+ \cdots \\
&+ g_{N-1}(x_2)\Big(g_0(x_1)f|_{0,N-1} + g_1(x_1)f|_{1,N-1} + \cdots \\
&+ g_{N-1}(x_1)f|_{N-1,N-1}\Big),
\end{aligned}
$$

where $f \equiv f_s$. It can be noted by comparing f and f_s that, in f_s, each literal $g_e(x_1)$ $(g_e(x_2))$ is swapped with $g_e(x_2)$ $(g_e(x_1))$ for $e \in I_N$ and each cofactor $f|_{r,s}$ $(f|_{s,r})$ is swapped with $f|_{s,r}$ $(f|_{r,s})$. Fig. 4 shows the MODDs corresponding to f (top MODD) and f_s (bottom MODD). In the top MODD variables, x_1 and x_2 appear in levels 0 and 1, respectively, whereas the cofactors appear as external nodes. This can be a more general case: The two variables may appear in any arbitrary but consecutive levels in a larger MODD, for example, variables x_i and x_{i+1} and $2 < i \leq n$. Clearly, to swap two variables, all we have to do is 1) swap the contents of the nodes (that is, the variables) in the two levels and 2) swap each cofactor $f|_{r,s}$ $(f|_{s,r})$ with $f|_{s,r}$ $(f|_{r,s})$. However, care must be exercised when a level has one or more missing nodes due to reduction. If this happens, then the missing nodes may have to be recreated in the swapped version. Also, some nodes may become redundant after the swap, in which case, the redundant nodes must not appear in the final result (refer to Example 3). However, if a node at level i $(0 \leq i < n)$, which is to be swapped with level $i+1$, does not have any children at level $i+1$, then it can be moved to level $i+1$ directly. By similar reasoning, if a node at level $i+1$ does not have any parent nodes at level i, then that node can be moved up to level i directly. Note that swapping two levels i and $i+1$ does not affect the other levels, that is, those in the range $0 \leq j < i$ (if $i > 1$) and $i+1 < k < n$ (if $i < n-1$).

The heuristic swap-based variable reordering algorithm presented in the following is based on this (Theorem 1). A swap-based variable reordering algorithm exists for BDD [18], but it is not suitable for MODD reordering.

The algorithm uses an array of hash tables, where the array indexes correspond to the levels in an MODD for direct access to each of the nodes within a level. It proceeds by sifting a selected level (that is, a variable) up or down by swapping it with a previous or a next level. The level with the largest number of nodes is considered first and then the one with the next largest node count, and so forth, that is, the array of hash tables is sorted in descending order of the hash table sizes. Once the level to be sifted first is considered, it is sifted up if it is closer to the root or down if it is closer to the external nodes. If it lies in the middle, then the decision to sift either up or down is made arbitrarily. The algorithm stops after a complete sift up and down operation. The complexity of the algorithm can be argued to be $O(n^2)$ [18]. Various heuristics have been considered to limit the sift and swap operations for speed up. For example, if a sift-up (down) operation doubles the node count, then no more sift-up (down) operations are carried out.

Example 3. Fig. 5 shows the basic idea behind the swap algorithm. Fig. 5a shows the original MODD for a function $f(x_1, x_2, x_3)$ in GF(4) generated by recursive expansion of Theorem 1. Here, variables x_1, x_2, and x_3 appear in levels 0, 1, and 2, respectively. Level 1 (that is, variable x_2) contains the largest number of nodes. Hence, this is considered to be the starting point of the sift operation. Level 1 is equidistant from level 0 and level 2. Hence, a sift up is chosen arbitrarily. A swap between levels 0 and 1 results in Fig. 5b. The nodes shown with broken lines are redundant, owing to the fact that all of their children point to the same node (Corollary 2.1). Hence, these nodes are not considered in the final result in Fig. 5c. For example, considering the paths with the edge $x_2 = 1$ in Fig. 5b, all of the paths with edge $x_2 = 1$ leading to node x_3 in Fig. 5a have edges with variable $x_1 = i$ for $i = 0, 1, \alpha, \beta$. Therefore, node x_1 becomes redundant by Corollary 2.1 if node x_1 appears after

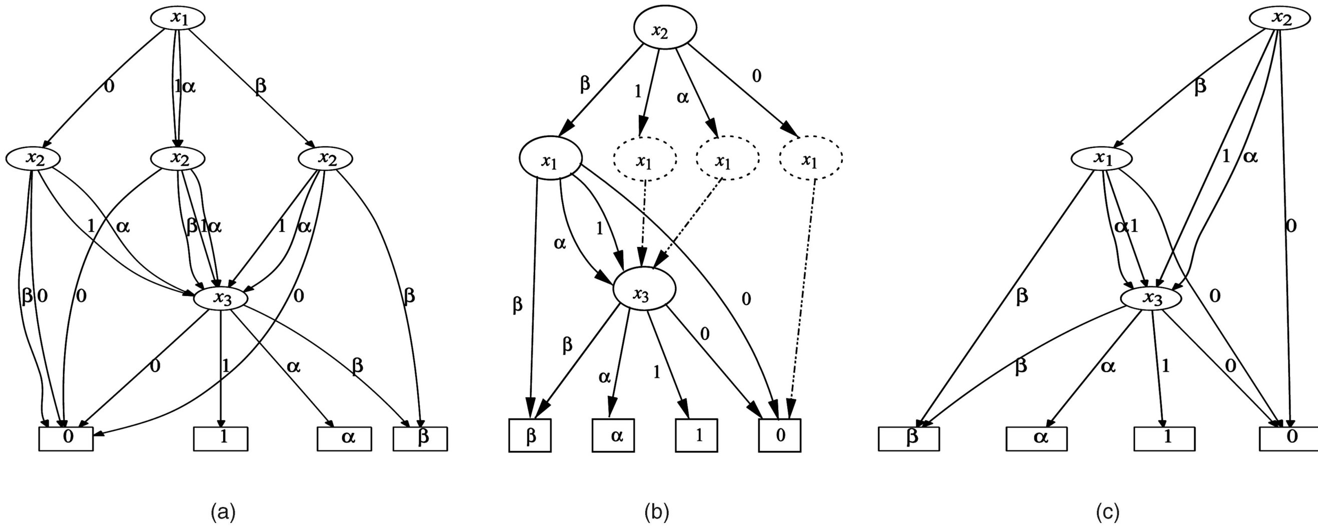

Fig. 5. Reordering example. (a) Before reordering. (b) Intermediate stage. (c) After reordering.

node x_2 under this circumstance. The same reasoning applies to the other two nodes with variable x_1, shown with the broken lines.

It can be shown that further sift operations do not yield additional node reduction. The original node count was 5 and the new node count is 3.

2.3 Operations in $\mathrm{GF}(N)$

Algebraic operations. Algebraic operations such as addition, multiplication, subtraction, and division in $\mathrm{GF}(N)$ can be carried out between two MODDs. Consider the following lemma, which can be shown to hold by perfect induction.

Lemma 3. *Let $f(x_1, \dots, x_i, \dots, x_n)$ and $h(x_1, \dots, x_i, \dots, x_n)$ be two functions in $\mathrm{GF}(N)$ and let "$\odot$" represent an algebraic operation in $\mathrm{GF}(N)$. Then,*

$$f \odot h = \sum_{e=0}^{N-1} g_e(x_i)\Big(f|_{x_i=\delta(e)} \odot h|_{x_i=\delta(e)}\Big), \tag{2}$$

where $g_e(x_i) = 1 - [x_i - \delta(e)]^{N-1}$.

Lemma 3 can be implemented recursively to perform algebraic operations between MODDs. However, the application of Lemma 3 directly will almost certainly be explosive in terms of the search space. Two things can be done to eliminate this. First, while the resulting DD in $\mathrm{GF}(N)$ is being constructed, it can be reduced at the same time. Second, intermediate results can be stored in a cache (dynamic programming), thus eliminating many operations which would otherwise have to be repeated.

Let G_f and G_h be the reduced MODDs for f and h, respectively. The complexity of such an operation can be reasoned about by considering the case for BDDs [3]. Assuming that insertion and deletion from the cache can be carried out in constant times, owing to the dynamic programming nature, the number of recursive calls can be limited to $O(|G_f| \cdot |G_h|)$.

Composition. We have the following, which can be shown to hold by perfect induction.

Lemma 4. *Let $f(x_1, x_2, \dots, x_i, \dots, x_n)$ and $h(x_1, x_2, \dots, x_n)$ be two functions in $\mathrm{GF}(N)$. Then,*

$$f|_{x_i=h} = \sum_{e=0}^{N-1} \Big[1 - (h - \delta(e))^{N-1}\Big] f|_{x_i=\delta(e)}. \tag{3}$$

An algorithm for the composition of two functions in $\mathrm{GF}(N)$ can be formed based on Lemma 4 in a manner similar to that for a BDD [3]. For this operation, we require a *restrict operation* in $\mathrm{GF}(N)$, similar to that for a BDD, and the algebraic operations presented previously. The restrict algorithm can be constructed for a reduced ordered DD in $\mathrm{GF}(N)$ in a manner similar to that for a BDD and is not shown here for brevity.

Multiple-valued SAT. Given a function $f(x_1, x_2, \dots, x_n)$ in $\mathrm{GF}(N)$ and $T \subseteq I_N - \{0\}$, the idea is to find an assignment for x_i, $\forall i \in \{1, 2, \dots, n\}$, such that the value of f is in $\{\delta(s)|s \in T\}$. If such an assignment exists, then f is said to be satisfiable (MV-SAT); otherwise, it is unsatisfiable.

The MV-SAT problem finds applications in bounded model checking, simulation, testing, and verification. An algorithm for *any* such satisfying assignment will have a complexity $O(|G_f|)$, where G_f is the reduced MODD for the function f in $\mathrm{GF}(N)$. An algorithm for *all* such assignments would have an exponential complexity. However, this process can be sped up by considering CF and ECF in $\mathrm{GF}(N)$ and their evaluation times, as discussed in Sections 3 and 4 [9], [13].

2.4 Multiple-Output Functions in $\mathrm{GF}(N)$

For multiple-input, multiple-output binary functions, the inputs or outputs can be arbitrarily grouped into m-bit chunks and each m-bit chunk can be represented in $\mathrm{GF}(2^m)$ with a single MODD. Further node reduction can be obtained by sharing the nodes between each of the MODDs representing a chunk of bits. Such an MODD will be called a *shared MODD* (SMODD). The general idea is shown in Fig. 6. The SMODD is basically a single diagram with multiple root nodes, which is also canonic. The canonicity

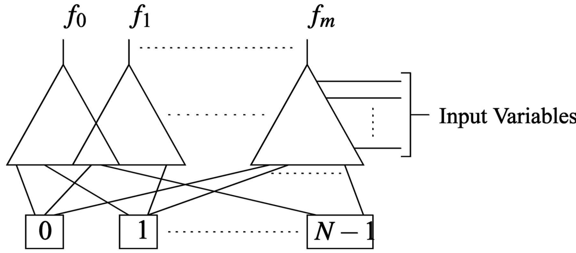

Fig. 6. General structure of a shared MODD.

of the SMODD can be argued in a similar manner as in a single MODD.

Similar reasoning can be carried over to higher order fields. Given any multiple-output function in $\mathrm{GF}(R)$, where R is a power of a prime, the inputs and outputs can be arbitrarily grouped into m R-valued chunks and each chunk can be represented in $\mathrm{GF}(R^m)$ by means of an MODD. Then, an SMODD will represent all of the chunks simultaneously.

The concept of levels is applicable to SMODDs across all of the outputs simultaneously by trivial reasoning. Therefore, the theory behind variable reordering, as discussed in Section 2.2, applies equally well to SMODDs. In this case, when levels i and $i+1$ $(0 \leq i < n)$ are swapped, all of the nodes in levels i and $i+1$ across all of the outputs have to be considered simultaneously. Therefore, the swap-based sift reordering algorithm discussed in Section 2.2 works equally well for SMODDs as it does for MODDs.

2.5 Further Node Reduction

Further node reduction can be obtained by means of two rules, in addition to the two rules presented in Section 2:

- *Zero suppression.* Suppress the 0-valued terminal node, along with all of the edges pointing to it.
- *Normalization.* Move the values of the nonzero terminal nodes as weights to the edges and ensure that 1) the weight of a specific valued edge (for example, that with the highest value) is always 1 and 2) assuming P represents the set of all of the paths, $\forall z \in P$, the $\mathrm{GF}(N)$ product of all of the weights along z is equal to the value of the function corresponding to z.

Note that the zero-suppression rule is unlike the reduction rule for the zero-suppressed BDD [11]. It can be argued that the above two rules will also maintain the canonicity if the weights are assigned in a fixed order throughout the graph during normalization. A reduced graph obtained using the above four reduction rules in $\mathrm{GF}(N)$ will be called a *Zero-suppressed Normalized MODD* (ZNMODD). The values of the terminal nodes in an MODD are *distributed* as weights over each path in the ZNMODD. To read off a value of a function from a ZNMODD, first, the path corresponding to the inputs is determined. Then, all of the weights along that path are multiplied in $\mathrm{GF}(N)$, which corresponds to the value of that function. In the rest of the paper, the weight of the highest valued edge will be normalized to 1 unless otherwise stated.

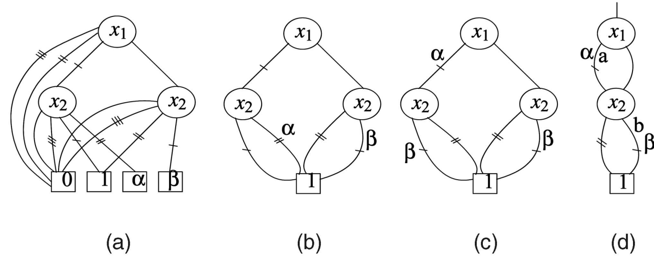

Fig. 7. Example of ZNMODD reduction.

Example 4. Let us consider the function $f(x_1, x_2) = [0\beta1001\alpha000000000]$ in GF(4), where $\{0, 1, \alpha, \beta\}$ are the elements of GF(4). Fig. 7a shows this function realized by means of a reduced MODD. Figs. 7b, 7c, and 7d show the gradual conversion to ZNMODD. Here, the lines with zero, one, two, and three cuts represent the values 0, 1, α, and β, respectively. Note how the weights are moved around and adjusted.

In Fig. 7b, the terminal node with 0 value is suppressed, along with all of the edges pointing to it. Also, the nonzero values of the terminal nodes are moved as weights associated with the terminal edges. Let us normalize with respect to the highest valued edge, that is, make the weight of the highest valued edges, β in this case, equal to 1. The α-edge of the left subgraph rooted at x_2 has a weight of α. Therefore, in order to make its weight equal to 1, α is moved up one level, whereas the 1-edge is assigned a weight of β to maintain the correctness of the underlying function. This results in the ZNMODD in Fig. 7c. Clearly, in Fig. 7c, the two subgraphs rooted at x_2 are isomorphic, which can be shared, resulting in the ZNMODD in Fig. 7d.

Now, let us find the value of $f(1,1)$. This should yield a 1. In Fig. 7d, this corresponds to the path ab. The value of this function is therefore $1 \cdot \alpha \cdot \beta = \alpha \cdot \beta = 1$. Similarly, $f(1, \alpha)$ yields $1 \cdot \alpha \cdot 1 = \alpha$, and so on.

Note that the total number of paths in Fig. 7a is 10, whereas that in Fig. 7d is only 4.

3 Representing Characteristic Functions in $\mathrm{GF}(N)$

The characteristic function (CF) defines a relation over inputs and outputs such that CF $= 1$ if, for a specific input combination, the output is valid; otherwise, CF $= 0$.

Let us consider a multiple-output function defined over finite fields $f(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_m)$. Let $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_m)$. Then, the $(n+m)$-input 1-output CF is defined as

$$\phi(X, Y) = \begin{cases} 1 & \text{if } f(X) = Y \\ 0 & \text{otherwise.} \end{cases}$$

An SMODD can be constructed from the above, which will constitute the n input variables and m *auxiliary* variables (AVs) corresponding to each of the outputs. Such an SMODD will be called CF-SMODD. Given an input combination of f, the nodes corresponding to the AVs in the CF-SMODD decide the outputs of f. For each node corresponding to an AV, except for only one edge, all of the

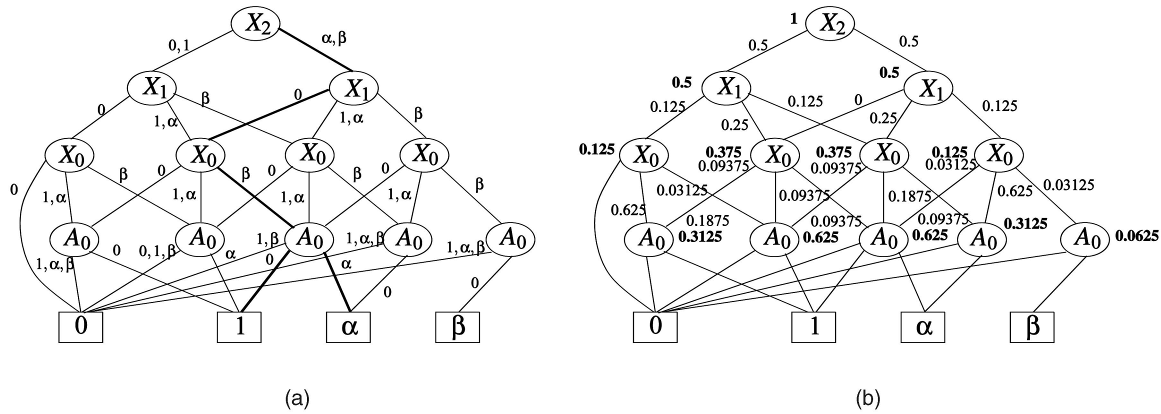

Fig. 8. (a) ECF-SMODD. (b) Computation of the APL.

other edges lead to the terminal node 0. The edge leading to the nonzero terminal node determines the output of the function. Examples of the CF can be found in [21].

The concept of CF can be extended by allowing output encoding since there is only one possible set of outputs for a given input combination. The resulting function can be represented by a mapping $\eta : G^n \times G^l \rightarrow G$, where $l = \lceil \log_N(m) \rceil$, and will be called *encoded* CF (ECF). The ECF has l AVs. Each output is defined by one of the N^l input combinations in an l AV function. As with the CF-SMODD, an ECF can be represented by means of an SMODD, which we shall call the ECF-SMODD. The following example illustrates the key points.

Example 5. Let us consider a 5-input 3-output binary function defined as follows, with the inputs denoted by the variables $(x_0, x_1, x_2, x_3, x_4)$ and the outputs denoted by (f_0, f_1, f_2):

$$f_0 = \sum m(15, 23, 26, 29, 30, 31),$$
$$f_1 = \sum m(1, 2, 4, 8, 11, 13, 14, 16, 21, 22, 26, 31),$$
$$f_2 = \sum m(3, 5, 6, 7, 8, 9, 14, 11, 12, 13, 17, 19, 20, 21, 22, 23, 24, 25, 26, 28).$$

Let us assume that the function is encoded in GF(4), with the inputs and outputs grouped as $X_0 = (x_0, x_1)$, $X_1 = (x_2, x_3)$, $X_2 = x_4$, $F_0 = (f_0, f_1)$, and $F_1 = f_2$. The CF of this function is $\phi(X_0, X_1, X_2, F_0, F_1)$. We have assumed that the binary combinations $10 = \alpha$ and $11 = \beta$. The variables F_0 and F_1 can be encoded as $A_0 = 0$ for F_0 and $A_0 = \alpha$ for F_1, resulting in the ECF $\eta(X_0, X_1, qX_2, A_0)$. Note that we can encode four functions by using one AV. Here, we have only two functions. The resulting ECF-SMODD appears in Fig. 8a.

4 Evaluation of Functions

The evaluation of the SMODDs is required for finding satisfying assignment corresponding to an input pattern. The path corresponding to the given input pattern is traced from the root node to one of the terminal nodes and the value of the terminal node gives the satisfying assignment, if it exists. This is an $O(n)$ operation, which can become a bottleneck, especially when the number of inputs is large and there are many input patterns that require evaluating. However, fast evaluation is highly desirable for applications in simulation, testing, and safety checking [9], [13].

In the case of a CF, the outputs are evaluated at the AVs. As we know, all of the outgoing edges except for only one edge lead to the zero terminal node. The remaining edge indicates the value of the function. With the ECF, once we reach the node corresponding to the AV, the paths corresponding to each of the output encodings is traced to find the value.

For example, let us consider an input pattern $(x_0 = 1, x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1)$ for the ECF-SMODD in Fig. 8. The pattern is $(X_2 = \alpha, X_1 = 0, X_0 = \beta)$ if it is encoded in GF(4) (Example 5). The path traced by the evaluation is shown in boldface. After reaching node A_0 in the path, the path corresponding to the given encoding for each output has to be taken into account. In this case, the encoding was defined as $A_0 = 0$ for F_0 and $A_0 = \alpha$ for F_1. Hence, if we take the path corresponding to $A_0 = 0$, then we end up at terminal node 1, thus giving us $F_0 = 1$. Similarly, $F_1 = \alpha$. This results in $(f_0 = 0, f_1 = 1, f_2 = 1)$, which is the required evaluation.

Comparison of evaluation times. Functions can be represented and evaluated by means of SMODD, CF-SMODD, or ECF-SMODD. This section provides a mechanism for comparing the evaluation times for each of the cases. We have tested our theory on many benchmarks and the results appear in Section 5. A good estimation of evaluation times can be obtained by computing the *APL*, which we define below:

1. *Node traversing probability* $(P(V_i))$: This is the probability of traversing the node V_i when an MODD is traversed from the root node to a terminal node.
2. *Edge traversing probability* $(P(e_{j,v_i}))$: This is the probability of traversing the edge $j = 0, 1, \ldots, p^{m-1}$ from the node V_i, that is, $P(e_{j,v_i}) = \frac{P(V_i)}{p^m}$, for nodes corresponding to the input variables.
3. The edge traversing probability for edges emanating from a node corresponding to an AV is $P(e_{j,v_i}) = P(V_i)$ since all of the edges have to be traversed to determine all the outputs of the function.

```
Algorithm ENC_PROB(int NODE, float PROB)
{  // NODE is the current node, and PROB is its probability
   if (NODE = Terminal Node) return 0;
   Add PROB to NODE's probability; // Each node stores its probability
   Decrease the Reference count of NODE by 1;
   if (Reference count of NODE > 0) return 0;
   float TOT = 0;
   for(int i = 0; i < N_o; i++){ // N_o is the output field size
      T = child_i(NODE);
      if(NODE = auxiliary node){
         clear reference counts for all nodes of the subtree T;
         make new reference count only for T;
         clear probability variable in all nodes of T;
         APL for T = ENC_PROB(T, NODE's probability);
         TOT = TOT + APL for T + NODE's probability;
      }else{
         APL for T = ENC_PROB(T, NODE's probability/N_o);
         TOT = TOT + APL for T;
      }
   }
   if(NODE != auxiliary node) TOT = TOT + NODE's probability;
   return TOT;
}
```

Fig. 9. Algorithm for calculation of APL in ECF-SMODD.

4. The node traversing probability is equal to the sum of all of the edge traversing probabilities incident on it.
5. *Average Path Length (APL)*: For an SMODD, the APL is equal to the sum of the node traversing probabilities of the nonterminal nodes. For a CF-SMODD and an ECF-SMODD, the APL is equal to the sum of the node traversing probabilities of those nodes above the AVs and the APLs for each subgraph rooted at the AVs.
6. The APL of a shared MODD is the sum of the APLs of the individual MODDs.

An algorithm for computing the APL for ECF-SMODD appears in Fig. 9. Algorithms for computing the APLs for SMODD and CF-SMODD can be formulated from this algorithm, which we have implemented in Section 5, but the details have been left out for brevity.

For example, the node and edge traversing probabilities of the ECF-SMODD in Fig. 8a appear in Fig. 8b. The first three levels in the tree correspond to the input variables. Hence, the probabilities are computed using Definitions 1, 2, and 4. However, since all of the outputs have to be considered, the APL for each subtree is separately computed at the auxiliary nodes. The probabilities at the auxiliary nodes correspond to the sums of the APLs of each subtree of the outputs. The APL is

$$1 + (0.5 + 0.5) + (0.125 + 0.375 + 0.375 + 0.125) + (0.3125 + 0.625 + 0.625 + 0.3125 + 0.0625) = 4.9375.$$

5 Experimental Results

The techniques in this paper have been applied to a number of benchmarks, including integer multiplier circuits. The program was developed in C++ (Gnu C++ 3.2.2) and tested on a Pentium 4 machine with 256-Mbyte RAM running RedHat Linux 9 (kernel 2.4).

Performance. Table 1 shows the results from the standard IWLS '93 and MCNC benchmark sets. Columns "I/P" and "O/P" represent the total number of inputs and outputs, column "SBDD" shows the number of nodes obtained using the shared reduced ordered BDD (ROBDD) representation, whereas column $GF(2^r)$ represents the number of nodes obtained based on the proposed DD for a field size of 2^r. In column $GF(2^r)$ r, adjacent bits are grouped together for each variable in $GF(2^r)$. The nodes of the proposed DD are also shared across the outputs. The same notation is used for the other tables.

The columns with the headings "W/o reord" and "Reord" present the node counts without and with variable reordering, respectively. A first-come, first-served variable ordering is considered for the "W/o reord" columns. For the reordering, the variable-by-variable swap-based sifting algorithm from Section 2.2, has been employed. Significant node reduction is apparent for many circuits, for example, misex3c, and so forth. However, in some cases, the node count has increased due to the lack of sharing. Note that a successive swap operation in a particular direction (up or down) is only carried out if a swap operation does not increase the size of the MODD by 2 or more. This restriction

TABLE 1
Same Field Size for Input and Output

Benchmark	I/P	O/P	SBDD	$GF(2^2)$		$GF(2^3)$		$GF(2^4)$	
				W/o reord	Reord	W/o reord	Reord	W/o reord	Reord
5xp1	7	10	88	48	42	43	35	27	16
9sym	9	1	33	17	17	10	10	-	-
apex4	9	19	1021	536	515	324	324	241	136
b12	15	9	91	78	47	48	45	61	51
bw	5	28	118	76	72	59	47	59	21
clip	9	5	254	136	89	88	41	49	31
misex3c	14	14	844	505	279	396	181	586	156
duke2	22	29	366	793	507	783	445	601	453
table5	17	15	685	751	678	636	636	499	348
e64	65	65	194	943	569	858	601	624	495
cordic	23	2	75	29	28	21	20	15	15
misex2	25	18	100	93	81	50	42	45	41
pdc	16	40	596	433	310	384	384	212	212
spla	16	46	628	352	339	261	261	155	155
ex1010	10	10	1402	662	654	346	344	670	207

TABLE 2
I/P Field Varying with Constant Output Field Size of 2

Benchmark	I/P	O/P	SBDD	GF(2^2)	GF(2^3)	GF(2^4)
5xp1	7	10	88	57	51	43
9sym	9	1	33	18	16	22
apex4	9	19	1021	509	346	240
b12	15	9	91	69	56	60
bw	5	28	118	71	47	42
clip	9	5	254	149	106	69
misex3c	14	14	844	450	300	246

TABLE 3
O/P Field Size Varying with Constant Input Field Size of 2

Benchmark	I/P	O/P	SBDD	GF(2^2)	GF(2^3)	GF(2^4)
5xp1	7	10	88	87	77	87
9sym	9	1	33	-	-	-
apex4	9	19	1021	1031	989	983
b12	15	9	91	102	84	126
bw	5	28	118	154	148	140
clip	9	5	254	227	211	195
misex3c	14	14	844	903	978	1333

can sometimes be relaxed (for example, by considering the maximum allowable size increase to be 1.5 times) to obtain better results. However, this also increases the execution time as more swaps are carried out. Variable reordering algorithms exist for MDDs [8]. However, they cannot be directly compared to the presented technique because the presented technique has been applied to varying input and output field sizes, whereas [8] seems to have ignored this aspect.

Apart from the benchmark 9sym, all of the circuits have been tested up to GF(16). 9sym is a single-output circuit and, hence, its testing in higher order fields seemed to be unjustified. In Table 1, the input and output field sizes are kept the same. Clearly, as we move to a higher order field, the number of nodes is reduced for the majority of the cases. Also, as we move to a higher order field, node reduction owing to reordering seems to be more effective.

For the rest of the tables, apart from Table 5, reordering has not been done to illustrate the other properties of the MODD more effectively, for example, the effect on the node count when the input and output field sizes are varied, as well as when the MODDs are ordered based on the evaluation times.

Table 2 represents the results for the same set of benchmarks under the same variable ordering. However, the input field size is varied, whereas the output field size is kept at constant 2. Clearly, the number of nodes has reduced further for many benchmarks as compared to Table 1. The reason seems to be the improved sharing of nodes between the different outputs.

Table 3 shows the result with the input field size kept at constant 2 and the output field size varied, again under the same variable ordering. In general, the number of nodes seems to have increased due to the lack of sharing between the different outputs. In other words, higher output field size seems to hamper sharing of nodes between different outputs even though the number of nodes in each output may reduce. This observation seems to be consistent with the conclusion drawn from Table 2.

Table 4 shows the results for $n * n$ integer multipliers for $n = 2, 3, 4, 5, 6$. The input and output field sizes are kept the same in this table. Clearly, a substantial reduction in the number of nodes is noticeable for the majority of the benchmarks. In some cases, reordering has produced further improvement, for example, the $4 * 4$ and $6 * 6$ multipliers in GF(8), and so forth. Although we have not explicitly shown the results in $\mathrm{GF}(2^5)$ and $\mathrm{GF}(2^6)$, MODD reported only 63 nodes as opposed to the 471 nodes for SBDD for the $5 * 5$ multiplier in $\mathrm{GF}(2^5)$. Also, for the $6 * 6$ multiplier, the number of nodes reported by the MODD in $\mathrm{GF}(2^6)$ is only 127, as opposed to 1,348 for the SBDD, that is, more than an order-of-magnitude reduction. Also, this table suggests that the node reduction seems to improve as we consider larger and more practical integer multipliers.

Table 5 shows the results with varying input field size and a constant output field size of 2. Again, considerable improvement has been observed for some benchmarks due to the improved sharing of the nodes across the outputs.

Table 6 shows the results for the multipliers with fixed input field size and varying output field size. As anticipated, the number of nodes has increased due to the possible lack of sharing.

As we move to a higher order field from a lower order field, the number of nodes usually decreases. This decrease is also associated with the smaller number of levels and shorter path lengths compared to conventional BDDs and their variants.

Evaluation time. Table 7 shows the results for the APLs as compared to SBDDs. For the majority of the cases, the APLs are significantly lower than those in the SBDDs. Also, as we go from GF(4) to GF(8), the APLs reduce further. On the average, the APLs are three times less in GF(4), whereas they are about six times less in GF(8) as compared to the SBDDs. That is, the evaluation time is essentially halved as we go from GF(4) to GF(8). Note that the number of nodes in the SMODDs is almost identical on the average compared

TABLE 4
Same Field Size for Both Input and Output

Multiplier	SBDD	GF(2^2)		GF(2^3)		GF(2^4)	
		W/o reord	Reord	W/o reord	Reord	W/o reord	Reord
2*2	14	7	7	6	5	1	-
3*3	51	28	28	15	15	16	9
4*4	157	98	87	84	60	31	31
5*5	471	254	249	183	183	272	121
6*6	1348	795	731	736	624	431	428

TABLE 5
I/P Field Varying with Constant Output Field Size of 2

Multiplier	SBDD	GF(2^2)	GF(2^3)	GF(2^4)
2*2	14	9	10	4
3*3	51	33	24	28
4*4	157	78	64	55
5*5	471	263	190	127
6*6	1348	695	389	465

TABLE 6
O/P Field Size Varying with Constant Input Field Size of 2

Multiplier	SBDD	GF(2^2)	GF(2^3)	GF(2^4)
2*2	14	15	11	12
3*3	51	50	65	46
4*4	157	178	194	260
5*5	471	490	553	755
6*6	1348	1587	1917	1890

to those in the SBDDs. This is due to the fact that the SMODDs have been ordered based on the APLs, which does not necessarily guarantee reduced node count. The dashes ("-") in the table indicate that results for those circuits (such as, ex1010, ex5, b12, risc, and apex4) are not available for the BDDs.

Tables 8 and 9 present the comparison of the APLs for SMODD, CF-SMODD, and ECF-SMODD for the benchmark circuits modeled in GF(4) and GF(8), respectively. These tables also show the results for 50,000 random vectors. The results for random pattern simulation constitute the net total path lengths for the 50,000 vectors. The spatial

TABLE 7
APL Compared to SBDD

Benchmark	I/P	O/P	BDD		SMODD					
			Nodes	APL	GF(4)			GF(8)		
					Nodes	APL	% Imp.	Nodes	APL	% Imp.
duke2	22	29	366	150.3	793	80.61	186	783	48.87	307
cordic	23	2	-	-	29	4.41	-	21	3.29	-
misex2	25	18	100	75.6	93	16.55	456	50	8.61	877
vg2	25	8	90	48.9	892	24.15	202	867	17.2	284
table5	17	15	685	114.1	751	32.12	355	635	16.02	712
pdc	16	40	596	215.4	433	52.58	409	384	33.13	650
e64	65	65	194	256	943	64.96	394	858	56.27	454
spla	16	46	628	226.6	352	47.33	478	261	30.18	750
ex1010	10	10	-	-	662	24.69	-	346	14.74	-
ex5	8	63	-	-	194	66.72	-	144	36.05	-
b12	15	9	-	-	78	15.34	-	48	8.31	-
x6dn	39	5	235	41.2	212	7.26	567	165	4.09	100
exep	30	63	675	255.7	462	53.86	474	409	33.44	764
x1dn	27	6	139	41	151	10.3	397	120	6.56	624
x9dn	27	7	139	50.4	160	12.6	399	174	10.66	472
mark1	20	31	119	115.7	149	36.95	313	150	26.63	434
mainpla	27	54	1857	277.5	2414	138.26	200	1667	70.74	392
risc	8	31	-	-	58	23.72	-	32	13.33	-
xparc	41	73	1947	304.8	1925	94.46	322	1550	58.22	523
apex4	9	19	-	-	536	36.54	-	324	16.73	-
Average	23.5	29.7	555	155.23	564.35	42.17	368	449.45	25.65	524.5

TABLE 8
APL with Random Pattern Simulation in GF(4)

Benchmark (IP/OP)	SMODD			CF-SMODD			ECF-SMODD		
	Nodes	APL	Random Simulation	Node	APL	Random Simulation	Nodes	APL	Random Simulation
duke2 (22/29)	793	80.61	4023605	445	8.1	405126.5	470	28.03	439206.5
cordic (23/2)	29	4.41	220771.5	30	5.14	256775	28	4.17	83466.8
misex2 (25/18)	93	16.55	825155	131	6.65	317410	83	4.22	84350.6
table5 (17/15)	751	32.12	1605105	555	6.51	326386.5	604	14.67	201920
pdc (16/40)	433	52.58	2630225	1356	9.28	464182	418	63.54	1200000
e64 (65/65)	943	64.96	3251360	66	1.33	66643.5	442	13.24	226700
spla (16/46)	352	47.33	2384420	1454	7.26	356000	345	47.93	961395
ex1010 (10/10)	662	24.69	1234420	555	7.29	343225	655	36.55	600000
ex5 (8/63)	192	66.72	3337715	980	20.77	1083580	205	110.72	2200000
b12 (15/9)	78	15.34	767230	157	7.11	265646.5	62	21.6	400000
x6dn (39/5)	212	7.26	363796.5	201	5.4	265861.5	144	3.08	5012.8
exep (30/63)	462	53.86	2695070	1459	10.05	493531.5	423	23.75	509220
x1dn (27/6)	151	10.3	515495	358	6.73	332395	127	5.92	2.71
x9dn (27/7)	160	12.6	628695	456	7.95	397483	129	6.47	129333.4
mark1 (20/31)	149	36.95	1850300	281	8.69	426828.5	116	18.4	323713.5
mainpla (27/54)	2414	138.26	6914400	2153	9.56	468033.5	811	41.51	1083405
risc (8/31)	58	23.72	1185410	64	4.97	248535	55	14.91	215953.5
xparc (41/73)	1925	94.45	4732915	2537	5.31	260840	983	16.63	337326.5
apex4 (9/19)	536	36.54	1825970	1078	10.81	540455	505	17.7	204093.5

TABLE 9
APL with Random Pattern Simulation in GF(8)

Benchmark (IP/OP)	SMODD			CF-SMODD			ECF-SMODD		
	Nodes	APL	Random Simulation	Node	APL	Random Simulation	Nodes	APL	Random Simulation
duke2 (22/29)	783	48.87	2444800	373	6.53	326456.5	412	29.15	637575
cordic (23/2)	21	3.29	165215	22	3.68	184286.5	20	2.7	90145
misex2 (25/18)	50	8.61	429593.5	88	4.2	204385	43	16.63	400000
table5 (17/15)	635	16.02	801950	379	4.48	224104.5	247	4.28	207866.5
pdc (16/40)	384	33.13	1656360	1096	5.62	281190	387	57.13	1200000
e64 (65/65)	858	56.27	2813180	44	1.14	57121.5	443	3.01	75326.25
spla (16/46)	261	30.18	1509950	1132	3.77	188649	259	31.6	848145
ex1010 (10/10)	346	14.74	737095	337	5.8	274060	345	22.73	400000
ex5 (8/63)	144	36.05	1802605	652	13.28	663865	148	68.05	1600000
b12 (15/9)	48	8.31	415075	120	5.43	265646.5	46	16.05	400000
x6dn (39/5)	165	4.09	206140	232	2.71	134725	151	2.45	81764.33
exep (30/63)	409	33.44	1674265	1073	6.54	327246.5	368	157.18	4536146.5
x1dn (27/6)	120	6.56	327983.5	302	5.23	259120	121	14.56	400000
x9dn (27/7)	174	10.66	532785	216	5.27	263295.5	129	5.16	129103.75
mark1 (20/31)	150	26.63	1850300	196	6.53	332668.5	116	139.87	800400
mainpla (27/54)	1667	70.74	3545165	1593	6.69	329915	537	91.71	2764640
risc (8/31)	32	13.33	666330	44	3.1	154980.5	35	27.16	800000
xparc (41/73)	1550	58.22	2914670	1820	3.8	188025	731	3.22	59733.25
apex4 (9/19)	324	16.73	837530	722	7.46	373047	325	24.73	400000

complexity is reflected by the node count and the speed of evaluation by the APLs and random pattern simulations. These results reflect the speed up over current methods for simulation, such as in [22]. The trade-off between these two factors across the representations is clearly evident from the results shown in these tables. In general, it can be seen that the CF-SMODD clearly wins out in terms of speed, whereas the ECF-SMODD tries to optimize between the speed and node count.

6 Conclusions

This paper focused on a framework for representing multiple-output binary and word-level circuits based on canonic DDs in $\mathrm{GF}(N)$. We showed that such reduced ordered DDs are canonical and minimal with respect to a fixed variable ordering. Techniques for further node and path optimization have also been presented. We also presented the theory for representing functions in $\mathrm{GF}(N)$ in terms of their CF and ECF under the same framework.

The proposed DDs have been tested on many benchmarks with varying input and output field sizes. The results suggest superior performance in terms of node compression and reduced APLs, which implies improved evaluation times. This has also been confirmed by the simulation of 50,000 randomly selected vectors. Overall, the results seem to suggest that the proposed framework can serve as an effective medium for verification, as well as for simulation, testing, and safety checking.

Acknowledgments

The authors would like to thank R. Krishna for his help with the programming. A preliminary version of parts of the paper appeared in [1], [23]. The work of Dhiraj K. Pradhan was partially funded by the Engineering and Physical Science Research Council (EPSRC), United Kingdom, Grant No. GR/S40855/01.

References

[1] A. Jabir and D. Pradhan, "MODD: A New Decision Diagram and Representation for Multiple Output Binary Functions," *Proc. Design, Automation and Test in Europe Conf. (DATE '04)*, pp. 1388-1389, Feb. 2004.

[2] R.E. Blahut, *Fast Algorithms for Digital Signal Processing.* Addison-Wesley, 1984.

[3] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677-691, Aug. 1986.

[4] C. Scholl, R. Drechsler, and B. Becker, "Functional Simulation using Binary Decision Diagrams," *Proc. Int'l Conf. Computer-Aided Design (ICCAD '97)*, pp. 8-12, 1997.

[5] C.H. Wu, C.M. Wu, M.D. Sheih, and Y.T. Hwang, "High-Speed, Low-Complexity Systolic Design of Novel Iterative Division Algorithm in $\mathrm{GF}(2^m)$," *IEEE Trans. Computers*, vol. 53, no. 3, pp. 375-380, Mar. 2004.

[6] D.K. Pradhan, M. Ciesielski, and S. Askar, "Mathematical Framework for Representing Discrete Functions as Word-Level Polynomials," *Proc. Eighth IEEE Int'l High-Level Design Validation and Test Workshop (HLDVT '03)*, pp. 135-142, Nov. 2003.

[7] D.M. Miller and R. Drechsler, "On the Construction of Multiple-Valued Decision Diagrams," *Proc. 32nd IEEE Int'l Symp. Multiple-Valued Logic (ISMVL '02)*, pp. 245-253, 2002.

[8] F. Schmiedle, W. Gunther, and R. Drechsler, "Selection of Efficient Re-Ordering Heuristics for MDD Construction," *Proc. 31st IEEE Int'l Symp. Multiple-Valued Logic (ISMVL '01)*, pp. 299-304, 2001.

[9] J.T. Butler, T. Sasao, and M. Matsuura, "Average Path Length of Binary Decision Diagrams," *IEEE Trans. Computers*, vol. 54, no. 9, pp. 1041-1053, Sept. 2005.

[10] K.M. Dill, K. Ganguly, R.J. Safranek, and M.A. Perkowski, "A New Zhegalkin Galois Logic," *Proc. Third Int'l Workshop Applications of the Reed-Muller Expansion in Circuit Design (Reed-Muller '97)*, pp. 247-257, Sept. 1997.

[11] S. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," *Proc. 30th IEEE/ACM Int'l Design Automation Conf. (DAC '93)*, pp. 272-277, June 1993.

[12] M.J. Ciesielski, P. Kalla, Z. Zeng, and B. Rouzeyere, "Taylor Expansion Diagrams: A Compact, Canonical Representation with Applications to Symbolic Verification," *Proc. Design, Automation and Test in Europe Conf. (DATE '02)*, Mar. 2002.

[13] P. Asher and S. Malik, "Fast Functional Simulation Using Branching Programmes," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD '95)*, pp. 408-412, Oct. 1995.

[14] P.C. McGeer, K.L. McMillan, and A.L. Sangiovanni-Vincentell, "Fast Discrete Function Evaluation Using Decision Diagram," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD '95)*, pp. 402-407, Nov. 1995.

[15] D.K. Pradhan, "A Theory of Galois Switching Functions," *IEEE Trans. Computers,* vol. 27, no. 3, pp. 239-249, Mar. 1978.
[16] R.E. Bryant and Y.A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams," *Proc. 32nd IEEE/ACM Int'l Design Automation Conf. (DAC '95),* 1995.
[17] R.S. Stanković and R. Dreschler, "Circuit Design from Kronecker Galois Field Decision Diagrams for Multiple-Valued Functions," *Proc. 27th Int'l Symp. Multiple-Valued Logic (ISMVL '97),* pp. 275-280, May 1997.
[18] R. Rudell, "Dynamic Variable Ordering for Ordering Binary Decision Diagrams," *Proc. Int'l Conf. Computer-Aided Design (ICCAD '93),* pp. 42-47, Nov. 1993.
[19] W. Stallings, *Cryptography and Network Security.* Prentice Hall, 1999.
[20] T. Kam, T. Villa, R.K. Brayton, and A.L. Sangiovanni-Vincentelli, "Multi-Valued Decision Diagrams: Theory and Applications," *Multiple Valued Logic,* vol. 4, nos. 1-2, pp. 9-62, 1998.
[21] T. Rajaprabhu, A. Singh, A. Jabir, and D. Pradhan, "MODD for CF: A Compact Representation for Multiple-Output Functions," *Proc. Ninth IEEE Int'l High-Level Design Validation and Test Workshop (HLDVT '04),* Nov. 2004.
[22] T. Sasao, Y. Iguchi, and M. Matsuura, "Comparison of Decision Diagrams for Multiple-Output Logic Functions," *Proc. 11th IEEE/ACM Int'l Workshop Logic and Synthesis (IWLS '02),* 2002.
[23] T.L. Rajaprabhu, A. Singh, A. Jabir, and D. Pradhan, "GASIM: A Fast Galois Field Based Simulator for Functional Model," *Proc. 10th IEEE Int'l High-Level Design Validation and Test Workshop (HLDVT '05),* Dec. 2005.
[24] S.B. Wicker, *Error Control Systems for Digital Communication and Storage.* Prentice Hall, 1995.
[25] Y. Jiang and R.K. Brayton, "Software Synthesis from Synchronous Specification Using Logic Simulation Techniques," *Proc. 39th IEEE/ACM Int'l Design Automation Conf. (DAC '02),* pp. 319-324, June 2002.

Abusaleh M. Jabir received the BSc degree (with honors) in computer science and the MSc degree (with distinction) in applied physics and electronics, both from the University of Dhaka, Bangladesh, and the DPhil degree in computing from the Computing Laboratory, University of Oxford, United Kingdom, in 2001, where he was with the Hardware Compilation Group, a subdivision of the renowned Programming Research Group. While completing the DPhil degree, he worked with Celoxica Ltd., United Kingdom, as a senior member of their research staff. He is currently a senior lecturer in the School of Technology at Oxford Brookes University, United Kingdom. Prior to that, he served as a lecturer in the Department of Computer Science, University of Dhaka, Bangladesh. His research interests include computer architectures, digital systems design (especially for low-power applications, tests, and verification), efficient hardware design for error control and reliability, and cryptosystems. He is the recipient of the IEE Hartree Premium Award (Best Paper Award) in 2004, with Jon Saul, for his paper "Minimization Algorithm for Three-Level Mixed AND-OR-EXOR/AND-OR-EXNOR Representation of Boolean Functions." He is a member of the IEEE.

Dhiraj K. Pradhan is currently a professor of computer science at the University of Bristol, Bristol, United Kingdom. Previously, he was a professor of electrical and computer engineering at Oregon State University, Corvallis, and held the COE Endowed Chair Professorship in Computer Science at Texas A&M University, College Station, where he also served as the founder of the Laboratory of Computer Systems, and he held a professorship at the University of Massachusetts, Amherst, where he also served as a coordinator of computer engineering. He also worked at the University of California, Berkeley, Oakland University, Rochester, Michigan, and the University of Regina, Saskatchewan, Canada. He was also a visiting professor at Stanford University, Stanford, California. In the past, he worked as a staff engineer at IBM. More recently, he served as the founding CEO of Reliable Computer Technology, Inc. He continues to serve as an editor of prestigious journals, including IEEE transactions. He has also served as the general chair and program chair for various major conferences. He is also the inventor of two patents, one of which was licensed to Mentor Graphics and Motorola. The recently announced verification tool, Formal Pro, by Mentor Graphics is based on his patent. He has contributed to very large scale integrated (VLSI) computer-aided design and test, as well as to fault-tolerant computing, computer architecture, and parallel processing research, with major publications in journals and conferences spanning more than 30 years. During this long career, he has been well funded by various agencies in Canada, the United States of America, and the United Kingdom. He is also the coauthor and editor of various books, including *Fault-Tolerant Computing: Theory and Techniques*, volumes I and II (Prentice Hall, 1986), *Fault-Tolerant Computer Systems Design* (Prentice Hall, 1996; second print, 2003), and *IC Manufacturability: The Art of Process and Design Integration* (IEEE Press, 2000). He is a fellow of the IEEE, the ACM, and the Japan Society for the Promotion of Science. He is the recipient of a Humboldt Prize in Germany. In 1997, he was also awarded the Fulbright-Flad Chair in Computer Science. He received Best Paper awards including the 1996 *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* Best Paper Award, with W. Kunz, on "Recursive Learning: A New Implication Technique for Efficient Solutions to CAD Problems Test, Verification and Optimization."

Ashutosh Kumar Singh received the PhD degree in electronics engineering from Banars Hindu University, India, in 2000. He is a faculty member in the Department of Computer Science at Curtin University, Miri, Malaysia. He worked as a senior lecturer and deputy dean on the Faculty of Information Technology at University Tun Abdul Razak, Kuala Lumpur, Malaysia. Prior to this, he was a postdoctoral research assistant in the Department of Computer Science at the University of Bristol, United Kingdom. He also worked in the Faculty of Information Science and Technology, Multimedia University, Malaysia, for two years and as a senior lecturer in the Department of Electronics and Communication at the National Institute of Science and Technology (INDIA), India. He was a member of the editorial board of the University Tun Abdul Razak (UNITAR) e-journal and has also been involved in the reviewing process of different journals and conferences, such as the *IEEE Transactions on Computers*, IEEE International Test Conference (ITC), International Conference on Advanced Computing and Communication (ADCOM), and so forth. His research interests include verification, synthesis, design, and testing of digital circuits. He has published approximately 36 research papers to date in different conferences and journals in these areas. He is a coauthor of two books, *Digital Systems Fundamentals* and *Computer System Organization and Architecture* (Prentice Hall). He is the recipient of the Merit Award from the Institute of Engineers in 2003, the Best Poster Presenter Award from the 86th Indian Science Congress in 1999, and the Best Paper Presenter from the 23rd National Systems Conference (NSC '99) in India.

T. L. Rajaprabhu is currently completing the MSc degree in the School of Electrical and Computer Engineering at the GeorgiaInsstitute of Technology. Prior to this, he worked as a research assistant in the Department of Computer Science at the University of Bristol, United Kingdom.