GfXpress: A Technique for Synthesis and Optimization of GF(2^m) Polynomials

Abusaleh M. Jabir, *Member, IEEE*, Dhiraj K. Pradhan, *Fellow, IEEE*, and Jimson Mathew, *Student Member, IEEE*

***Abstract*—This paper presents an efficient technique for synthesis and optimization of the polynomials over GF(2^m), where m is a nonzero positive integer. The technique is based on a graph-based decomposition and factorization of the polynomials, followed by efficient network factorization and optimization. A technique for efficiently computing the coefficients of the polynomials over GF(p^m), where p is a prime number, is first presented. The coefficients are stored as polynomial graphs over GF(p^m). The synthesis and optimization is initiated from this graph-based representation. The technique has been applied to minimize multipliers over the fields GF(2^k), where $k = 2, \dots, 8$, generated with all the 51 primitive polynomials in the 0.18-μm CMOS technology with the help of the Synopsys design compiler. It has also been applied to minimize combinational exponentiation circuits, parallel integer adders and multipliers, and other multivariate bit- as well as word-level polynomials. The experimental results suggest that the proposed technique can reduce area, delay, and power by significant amounts. We also observed that the technique is capable of producing 100% testable circuits for stuck-at faults.**

***Index Terms*—Decision diagrams, decomposition, finite or Galois fields, polynomials, synthesis and optimization, testing, verification.**

I. Introduction

POLYNOMIALS in finite fields and their extensions over the set GF(2^m) are crucial to certain error-control codes [36], public key cryptosystems (e.g., the elliptic curve cryptosystem) [7], [27], [32], VLSI testing [24], and digital signal processing [8]. The varied use of finite fields also leads to designing high-speed low-complexity systolic VLSI realizations [14]. In most cases, the evaluation of the polynomials cannot be justified unless they can be realized in high-speed low-power low-complexity dedicated hardware. A polynomial over GF(2^m) requires addition, multiplication, and exponentiation. While addition over GF(2^m) can be realized with m two-input EXOR gates, multiplication and exponentiation are costly in terms of area, delay, and power requirements. Most of the recent research on realizing these polynomials has focused on individually synthesizing the multipliers [2], [6], [23], [29] and sequential [21], [22] and hybrid [11] exponentiations over GF(2^m). These techniques are not suitable for synthesis and optimization of the multiple-output multivariate polynomials over GF(2^m), constituting addition, multiplication, and exponentiation in different configurations. However, these polynomials can be synthesized in hardware efficiently if appropriate factorization with simplification is first carried out. As our experimental results seem to suggest a significant amount of area (close to two orders of magnitude), delay (more than one order of magnitude), and power (more than one order of magnitude), reduction can be achieved if the polynomials are synthesized and optimized as a preprocessing step before they are passed on to the industrial tools such as the Synopsys design compiler.

Factorization of multiple-output multivariate polynomials over finite fields is known to be a very hard problem. Hence, hand-synthesizing complex polynomials by factorizing common subexpression elimination and algebraic simplification over GF(2^m) is impractical. Instead, an automatic synthesis tool, which can perform all of these, is desirable.

Keeping this in view, we propose a novel synthesis and optimization technique for polynomials over GF(2^m). For $m = 1$, i.e., GF(2), the proposed technique performs gate-level synthesis and optimization. For $m > 1$, it performs word-level synthesis and optimization based on the properties of GF(2^m). The technique is stable both at the gate as well as at the word levels, i.e., it is independent of the starting point and produces the same circuit for functionally equivalent polynomials. This technique is unlike many existing techniques, e.g., the Synopsys design compiler and those in [3] and [19], which can produce structurally different circuits for the same function and depend on the structure of the initial circuit. This property of the proposed technique can be useful for fast verification of hardware minimized with it.

The basic idea behind the proposed technique is shown in Fig. 1. The initial specification can be truth-table- or directed-acyclic-graph (DAG)-based representations of the truth tables or the initial netlists, e.g., DAG-based representations of the initial specifications as AND–OR PLA or VHDL/verilog netlist forms. However, the technique requires that the multivariate polynomials are available in terms of their coefficients, either in a graph-based form or in terms of a network of adders or multipliers over GF(2^m), i.e., in a netlist form. If the graph-based representation of the coefficients is given, then it performs a graph-based decomposition over GF(2^m) to obtain the netlist of adders and multipliers.

Manuscript received March 1, 2007; revised July 23, 2007. The work of D. K. Pradhan was supported in part by the Engineering and Physical Science Research Council (EPSRC), U.K., under Grant GR/S40855/01. An earlier version of this paper was presented at the International Conference on Computer Aided Design (ICCAD'06), San Jose, CA, November 5-9, 2006. This paper was recommended by Associate Editor S. Nowick.

A. M. Jabir is with the School of Technology, Oxford Brookes University, Oxford, OX33 1HX, U.K. (e-mail: ajabir@brookes.ac.uk).

D. K. Pradhan and J. Mathew are with the Department of Computer Science, University of Bristol, Bristol, BS8 1UB, U.K. (e-mail: pradhan@cs.bris.ac.uk; jimson@cs.bris.ac.uk).

Digital Object Identifier 10.1109/TCAD.2008.917586

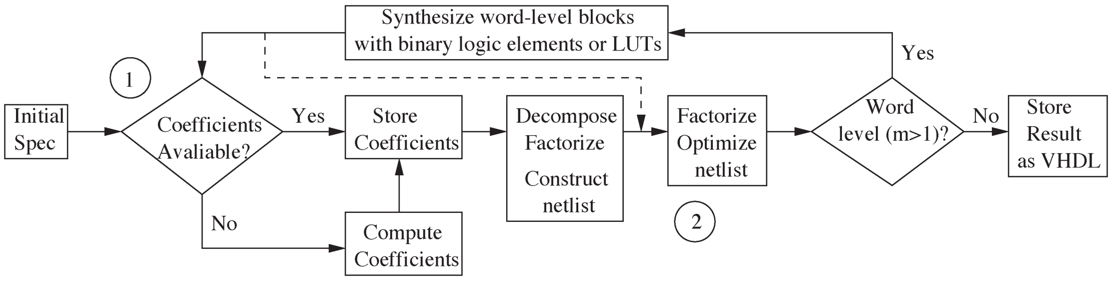
Synthesize word-level blocks with binary logic elements or LUTs
1
Initial Spec
Coefficients Avaliable?
Yes
No
Store Coefficients
Compute Coefficients
Decompose Factorize Construct netlist
Factorize Optimize netlist
2
Word level (m>1)?
Yes
No
Store Result as VHDL

Fig. 1. Basic design flow.

If the coefficients are not available, then they are calculated from the initial DAG and stored in another DAG, which forms the starting point of our synthesis process. We decompose DAGs corresponding to the polynomials into netlists which are further decomposed/factorized and optimized to obtain the final circuits. Fig. 1 shows a two-pass system for $m > 1$. During the first pass, the polynomials are synthesized and optimized at the m-bit word level over GF(2^m). During the second pass, the basic elements such as the adders, multipliers, and exponentiation circuits are resynthesized with gate/logic-level elements, and the whole netlist is optimized again at the gate level over GF(2). Note that we have not assumed anything regarding word sizes during each of the passes, i.e., the same approach works for both gate/bit- and word-level netlists. Hence, it provides with a unified framework. Since the netlists are assumed to constitute adders and multipliers with two inputs only, this type of structure provides with the finest grain optimization possible over a given m.

Once the word-level circuits have been synthesized and optimized, we have the option of taking two routes shown with the solid and broken arrows. Multipliers and exponentiation circuits over GF(2^m) can be realized in the bit parallel fashion with logic gates or with $2m$-input m-output and m-input m-output binary lookup tables (LUTs), respectively. If some of these components of the polynomials are realized with LUTs, then we need to compute the coefficients over GF(2) from the LUTs, i.e., start from (1) in Fig. 1. If these components are realized with presynthesized gate-level macro blocks (cells), then we only need to start from (2), i.e., perform gate-level netlist factorization and optimization only.

Any arbitrary combinations of input and output bits can be logically represented as the elements over finite fields. Hence, this approach can also provide with a wide range of synthesized circuits.

The underlying problem of representing polynomials over GF(p^m), where p is a prime number, is the computation of the coefficients because the polynomials are determined by their coefficients.

A number of techniques exist based on interpolation algorithms for small finite fields (e.g., GF(K) and $K \leq 4$) [1], [18], [37], [38]. Although the technique in [25] seems to be applicable to large fields, it relies heavily on the size of the fields, and also for small fields, a solution only exists if the interpolation points are chosen from a sufficiently large extension field [26]. Most of these techniques (see, e.g., [37] and [38]) are suitable for only the zero-polarity (i.e., only one polarity). However, most practical circuits have different representations in different polarities in finite fields, e.g., certain polarities will require fewer terms or nonzero coefficients and, hence, fewer nodes in a graph than others for the same circuit. In this regard, a technique has been presented in [33] for computing the expressions under the optimal polarity in GF(2) only. Owing to its exhaustive nature, its application is limited to small circuits. Reference [31] presented a technique for functions over GF(4) only without any mechanism for generalizing to higher order fields, although theoretically, it can be generalized. Unlike our technique, this technique does not employ factorization and optimization over the fields. In addition, it does not report any experimental results. Reference [15] also presented another exhaustive search-based technique for finding optimal expressions over GF(4) only from minterms based on an extension of the dual-polarity property over GF(2). Fourier-like matrix-based algorithms (e.g., based on coding theory and the butterfly algorithm) exist for functions in finite fields and also generalized Reed Müller forms [16], [34]. However, the size of the matrices grows exponentially with the size of the inputs and field sizes.

Keeping these in view, we consider an efficient DAG-based technique for computing the coefficients of polynomials over GF(p^m) in any polarity. The coefficients are stored as DAGs also as they are computed, which is the starting point of our synthesis technique.

The polynomial decomposition and factorization algorithms proposed in this paper are unlike the existing decomposition and factorization algorithms (see, e.g., [3], [10], and [19]). In this regard, Files and Perkowski [10] have proposed a multiple-valued decision diagram (MDD)-based Boolean-like disjoint decomposition technique for the MIN–MAX postalgebra in its literal form. In contrast, our technique is meant for polynomials over finite fields as polynomial DAGs and netlists.[1] In addition, our technique is inherently capable of performing both disjoint and nondisjoint decompositions on these polynomials, depending on which ever is likely to produce better results.

The technique in [3] is a generalization of the cube-kernel factorization (extraction) algorithms considered in SIS [19], [20] for applications to the polynomials over integers. In contrast to our technique, which is applicable to efficient canonic DAGs and netlists over finite fields, this technique is applicable to sets (arrays) of vectors or cubes, where each

[1]The finite fields are fundamentally different from the MIN–MAX postalgebra. For example, each nonzero element in finite fields has a unique inverse, but this is not true for MIN–MAX postalgebra.

$+$	0	1	α	β
0	0	1	α	β
1	1	0	β	α
α	α	β	0	1
β	β	α	1	0

(a)

$\times$	0	1	α	β
0	0	0	0	0
1	0	1	α	β
α	0	α	β	1
β	0	β	1	α

(b)

Fig. 2. Addition and multiplication over GF(4).

vector represents a term in the polynomials. These sets of vectors are known to be exponential unless the number of vectors is reduced by expensive minimization techniques. Our technique is independent of the starting point owing to the canonicity of the initial DAGs. In addition, our technique is applicable to bit (gate) as well as the word levels under a unified framework.

A. Background

Let GF(N) denote a set of N elements, where N is a power of a prime number, with two special elements 0 and 1 representing the additive and multiplicative identities, respectively, and two operators addition "$+$" and multiplication "$\cdot$". GF(N) defines a finite field, also known as Galois field, if it forms a commutative ring with identity over these two operators in which every nonzero element has a multiplicative inverse. Additional properties of finite fields can be found in [30] and [36]. Finite fields over GF(2^m) and $m \geq 2$ can be generated with primitive polynomials (PPs) of the form $p(x) = x^m + \sum_{i=0}^{m-1} c_i x^i$, where $c_i \in$ GF(2) [36]. For any α, $\beta \in$ GF(2^m), if α and β are in their polynomial basis as $\alpha(x) = \sum_{i=0}^{m-1} \alpha_i x^i$ and $\beta(x) = \sum_{i=0}^{m-1} \beta_i x^i$, where $\alpha_i, \beta_i \in \{0, 1\}$ and $0 \leq i < m$, then multiplication over GF(2^m) can be defined as $w(x) = \alpha(x) \cdot \beta(x) \bmod p(x)$, where $p(x)$ represents the PP used to generate the fields [6], [30], [36]. $\alpha + \beta$, i.e., addition over GF(2^m), is the pairwise EXOR of the terms in the polynomials or the bitwise EXOR of the bit vectors corresponding to $\alpha(x)$ and $\beta(x)$.

Fig. 2 shows the addition and multiplication tables over GF(4). Here, α and β are assumed to be elements in GF(4). For example, assume that $\alpha(x) = x$ and $\beta(x) = x + 1$. $\alpha + \beta = \alpha(x) + \beta(x) = x + x + 1 = 1$. There is only one PP for generating GF(4), which is $p(x) = x^2 + x + 1$. Now, $\alpha \cdot \beta = \alpha(x) \cdot \beta(x) \bmod p(x) = x \cdot (x + 1) \bmod x^2 + x + 1 = 1$.

In this paper, PPs will be considered in their decimal notation. For example, the PP $p(x) = x^3 + x + 1$ in GF(8) can be represented by the bit vector [1, 0, 1, 1], which is 11 in decimal.

The following notation is used in this paper. Let $I_N = \{0, 1, \ldots, N-1\}$, and $\delta : I_N \to$ GF(N) be a one-to-one mapping with $\delta(0) = 0$, $\delta(1) = 1$ and, without loss of generality, $\delta(2) = \alpha$, $\delta(3) = \beta$, etc.

Given a variable x_i in GF(N) and $\pi_i \in I_N$ $(1 \leq i \leq n$, where n is the number of inputs), x_i can appear in one of the N polarities denoted by $x_{i,\pi_i} = x_i + \delta(\pi_i)$. Certain polarities require more resources to represent a polynomial than others. As an example, let $f(x_1, x_2) = \alpha x_1 x_2$ represent a function in zero-polarity in GF(3), i.e., both x_1 and x_2 are in zero-polarity (note that $x_{i,0} = x_i$). In another polarity with x_1 in one-polarity and x_2 in two-polarity, replacing x_1 with $x_{1,1} - 1$ and x_2 with $x_{2,2} - \delta(2) = x_{2,2} - \alpha$ and simplifying them, we get $f(x_1, x_2) = 1 + x_{2,2} + \alpha x_{1,1} + \alpha x_{1,1} x_{2,2}$, which contains more terms in this case. Finding an optimal polarity is a very hard problem.

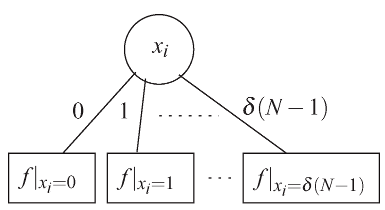

Fig. 3. MODD-A literal-based representation over GF(N).

B. Graph-Based Representation

Any function over GF(N) can be expanded in its literal-based form as $f(x_1, \ldots, x_i, \ldots, x_n) = \sum_{e=0}^{N-1} g_e(x_i) f|_{x_i=\delta(e)}$, where $g_e(x_i) = 1 - [x_i - \delta(e)]^{N-1}$ [30]. Here, $g_e(x_i)$ is called a literal over GF(N). This can be viewed as generalization of Shanon's expansion to GF(N). Such a literal-based form allows MDD-like canonic graph-based representations as shown in Fig. 3 [4], [5].

Here, the internal nodes represent the variables of expansion, x_i in this case, whereas the terminal nodes represent the values of the function corresponding to each value of x_i represented by the edges. However, since the MDDs are defined in MIN–MAX postalgebra [17], this representation will be called multiple-output decision diagram (MODD) to distinguish it from the MDDs. There are two MODD reduction rules [4]: 1) If all 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, and 2) share equivalent subgraphs. A reduced and shared MODD will be denoted as shared MODD (SMODD). An SMODD can be further optimized and normalized based on additional two rules mentioned in [5]. An SMODD, which is reduced by all the four rules, will be called a zero suppressed and normalized MODD (ZNMODD) [5].

This paper is organized as follows. Section II presents a technique for computing and storing coefficients over GF(p^m). Section III presents a synthesis and optimization technique based on the DAG-based representation of Section II. Finally, Section IV presents the experimental results.

II. Computation of Coefficients

Given a function $f(x_n, x_{n-1}, \ldots, x_1)$ in GF(N) (N is a power of a prime number) and a polarity number t, f can be represented in the following canonical form [30]:

$$\begin{aligned} f(x_n, x_{n-1}, \ldots, x_1) &= \kappa_{0,t} + \kappa_{1,t}\tilde{x}_1 + \kappa_{N,t}\tilde{x}_2 + \cdots + \kappa_{i,t}\tilde{x}_{j_k}^{i_k}\tilde{x}_{j_{k-1}}^{i_{k-1}}\cdots\tilde{x}_{j_1}^{i_1} \\ &\quad + \cdots + \kappa_{N^n-1,t}\tilde{x}_n^{N-1}\tilde{x}_{n-1}^{N-1}\cdots\tilde{x}_1^{N-1}. \end{aligned} \tag{1}$$

Here, $\kappa_{i,t}$ represents the coefficient of the ith term containing k number of variables in polarity number t, where both i and t are defined in the radix-N number system as $i = i_k N^{j_k - 1} + i_{k-1} N^{j_{k-1}-1} + \cdots + i_1 N^{j_1 - 1}$ and $t = \pi_n N^{n-1} + \pi_{n-1} N^{n-2} + \cdots + \pi_1$.

Furthermore, each π_q represents the polarity of the variable x_q and $1 \le q \le n$, i.e., $x_{q,\pi_q} = x_q + \delta(\pi_q)$. Here, $\tilde{x}_l$ represents the fact that x_l is in any one of the polarities $\{0, 1, \ldots, N-1\}$, as determined by t. For example, the term associated with the coefficient $\kappa_{i,t}$ is $\tilde{x}_{j_k}^{i_k} \tilde{x}_{j_{k-1}}^{i_{k-1}} \cdots \tilde{x}_{j_1}^{i_1}$, where the polarity of each variable is determined by t.

In many cases, the SMODD may be available as a part of existing resources during a synthesis, verification, or simulation process, in which case, the coefficients can be derived from the graphs by means of an efficient path-oriented technique given in [5]. We call this technique "CompByPath."

We have the following theorem.

Theorem 1 [5]: Let $f(x_n, x_{n-1}, \ldots, x_1)$ represent an n variable function in GF(N). Its ith coefficient corresponding to a term containing k number of variables in the polarity number t is

$$\kappa_{i,t} = (-1)^k \sum_{x_{j_k}=\delta(0)}^{\delta(N-1)} \sum_{x_{j_{k-1}}=\delta(0)}^{\delta(N-1)} \cdots \sum_{x_{j_1}=\delta(0)}^{\delta(N-1)} \theta_{i,t}(\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n) \times f\left(x_{j_k}, x_{j_{k-1}}, \ldots, x_{j_1}, -\delta(\pi_l), -\delta(\pi_{l-1}), \ldots, -\delta(\pi_1)\right) \tag{2}$$

where $\delta(\pi_l), \delta(\pi_{l-1}), \ldots, \delta(\pi_1)$ are the polarities of the variables $x_l, x_{l-1}, \ldots, x_1$ as determined by t.

Here, $\theta_{i,t}(\tilde{x}_n, \tilde{x}_{n-1}, \ldots, \tilde{x}_1) = \tilde{x}_{j_k}^{N-1-i_k} \tilde{x}_{j_{k-1}}^{N-1-i_{k-1}} \cdots \tilde{x}_{j_1}^{N-1-i_1}$, as defined in [30]. $\theta_{i,t}$ has the following properties: 1) $\theta_{i,t} = 1$ if $i = 0$ or $i_k = i_{k-1} = \cdots = i_1 = N-1$; 2) only those literals present in $\tilde{x}_{j_k}^{i_k} \tilde{x}_{j_{k-1}}^{i_{k-1}} \cdots \tilde{x}_{j_1}^{i_1}$ appear in $\theta_{i,t}$; and 3) those literals with exponents equal to $N-1$ do not appear in $\theta_{i,t}$ at all.

The complexity of algorithm CompByPath is associated with the following: 1) the number of paths which do not terminate into a terminal node 0 in the SMODD or ZNMODD and 2) the number of missing unassigned variables in the path. It has been observed that for most benchmarks, the algorithm found the coefficients quickly. However, the application of this algorithm for determining all the coefficients in large fields can render the process slow. A far more efficient technique is presented next.

1) Speeding Up in Large Fields: Let the partial coefficient (PC) be defined as $\mathcal{PC}_{i,t}(\tilde{x}_{j_k}^{i_k} \tilde{x}_{j_{k-1}}^{i_{k-1}} \cdots \tilde{x}_{j_1}^{i_1}) = \theta_{i,t}(\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n)\, f\left(x_{j_k}, x_{j_{k-1}}, \ldots, x_{j_1}, -\delta(\pi_l), -\delta(\pi_{l-1}), \ldots, -\delta(\pi_1)\right)$.

Then, (2) becomes

$$\kappa_{i,t} = (-1)^k \sum_{x_{j_k}=\delta(0)}^{\delta(N-1)} \sum_{x_{j_{k-1}}=\delta(0)}^{\delta(N-1)} \cdots \sum_{x_{j_1}=\delta(0)}^{\delta(N-1)} \mathcal{PC}_{i,t}\left(\tilde{x}_{j_k}^{i_k} \tilde{x}_{j_{k-1}}^{i_{k-1}} \ldots \tilde{x}_{j_1}^{i_1}\right). \tag{3}$$

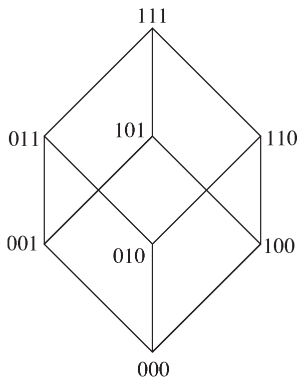

Fig. 4. Boolean lattice for $n = 3$.

Lemma 1: Let $\kappa_{i,t}$ and $\kappa_{i',t}$ be the coefficients corresponding to the two terms $\tilde{x}_{j_k}^{i_k} \tilde{x}_{j_{k-1}}^{i_{k-1}} \cdots \tilde{x}_{j_1}^{i_1}$ and $\tilde{x}_{j_k}^{i'_k} \tilde{x}_{j_{k-1}}^{i'_{k-1}} \cdots \tilde{x}_{j_1}^{i'_1}$, respectively. Also, let $i > i'$ and $i_l \ge i'_l$ for $1 \le l \le k$. Then $\kappa_{i',t}$ can be computed from $\kappa_{i,t}$ by multiplying the PCs of $\kappa_{i,t}$ by each value of $\tilde{x}_{j_k}^{i_k - i'_k} \tilde{x}_{j_{k-1}}^{i_{k-1} - i'_{k-1}} \cdots \tilde{x}_{j_1}^{i_1 - i'_1}$ and then adding them together without revisiting the ZNMODD (SMODD), i.e.,

$$\kappa_{i',t} = (-1)^k \sum_{x_{j_k}=\delta(0)}^{\delta(N-1)} \sum_{x_{j_{k-1}}=\delta(0)}^{\delta(N-1)} \cdots \sum_{x_{j_1}=\delta(0)}^{\delta(N-1)} \times \mathcal{PC}_{i,t}\left(\tilde{x}_{j_k}^{i_k} \tilde{x}_{j_{k-1}}^{i_{k-1}} \ldots \tilde{x}_{j_1}^{i_1}\right) \cdot \tilde{x}_{j_k}^{i_k - i'_k} \tilde{x}_{j_{k-1}}^{i_{k-1} - i'_{k-1}} \ldots \tilde{x}_{j_1}^{i_1 - i'_1}. \tag{4}$$

The proof follows directly from the definition PC and Theorem 1.

Lemma 1 gives us an opportunity to compute coefficients incrementally without having to visit the ZNMODD each time, i.e., without resorting to algorithm CompByPath each time.

Let us represent each unassigned variable with one and each assigned variable with zero. This results in a set of binary vectors of dimension N. Each position in the binary vector corresponds to whether that particular input variable is assigned or not. The set of all such binary vectors defines a lattice over the Boolean algebra. The greatest lower bound (GLB) of this lattice is $[00, \ldots, 0]$ (i.e., all variables assigned), whereas the least upper bound (LUB) is $[11, \ldots, 1]$ (i.e., all variables unassigned). Clearly, an N-variable function in GF(N) will have 2^n points in the lattice. For GF(2), each point in the lattice will correspond to a single coefficient. However, for GF(N) and $N > 2$, each point will correspond to a range of coefficients. For each 1 in the vector corresponding to a point in the lattice, apart from the GLB, the corresponding variable will have N different exponents in the term, ranging from 1 to $N-1$. Assume that each point in the lattice represents a term having exponents equal to $N-1$ for all the unassigned variables. If the coefficients of all such terms are known, then the coefficients of all the remaining terms having unassigned variables in the same place can be determined by Lemma 1, without visiting the SMODD.

The following example best describes this.

Example 1: Fig. 4 shows an example lattice for a three-input function in GF(4). The GLB corresponds to

```
1   Algorithm GfCoeffHs1{
2       do{
3           Get a point pt in the lattice in ascending order;
4           if(pt is the GLB)
5               Compute by Theorem 1;
6           else{
7               Consider the $\kappa_{i,t}$ with the highest exponent, i.e.
8               with $i_k = i_{k-1} = \cdots = i_1 = N-1$ corresponding
9               to the 1s in pt;
10              Compute $\kappa_{i,t}$ by Algorithm CompByPath;
11              For all $i'$ which differ in one exponent and by one
12                  Compute $\kappa_{i',t}$ by Lemma 1 from its previous
13                  coefficient;
14          }
15      }while(all points in the lattice are not considered);
16  }
```

Fig. 5. Fast algorithm for large finite fields.

the coefficient $\kappa_{0,t}$. The LUB corresponds to the terms $x_{1,\pi_1}x_{2,\pi_2}x_{3,\pi_3}, x_{1,\pi_1}x_{2,\pi_2}x_{3,\pi_3}^2, \ldots, x_{1,\pi_1}^3x_{2,\pi_2}^3x_{3,\pi_3}^3$, i.e., the coefficients $\kappa_{21,t}, \kappa_{22,t}, \ldots, \kappa_{64,t}$. Similarly, the point 001 corresponds to the terms x_{3,π_3}, x_{3,π_3}^2, and x_{3,π_3}^3, i.e., the coefficients $\kappa_{1,t}$, $\kappa_{2,t}$, and $\kappa_{3,t}$. Considering the point 001, the coefficient for x_{3,π_3}^2 can be computed as $\kappa_{2,t} = \sum_{x_3=0}^{\beta}(x_3 + \delta(\pi_3))f(\delta(\pi_1), \delta(\pi_2), x_3)$ (Theorem 1) using the algorithm CompByPath. Here, $\mathcal{PC}_{2,t} = (x_3 + \delta(\pi_3))f(\delta(\pi_1), \delta(\pi_2), x_3)$. If during this computation $\mathcal{PC}_{2,t}$ is stored for the relevant values of x_3 from the SMODD, then the coefficient of x_{3,π_3} can be computed as $\kappa_{1,t} = \sum_{x_3=0}^{\beta}\mathcal{PC}_{2,t} \cdot (x_3 + \delta(\pi_3))^{2-1} = \sum_{x_3=0}^{\beta}(x_3 + \delta(\pi_3))^2 f(\delta(\pi_1), \delta(\pi_2), x_3)$ by multiplying the values of $x_3 + \delta(\pi_3)$ by the stored values of $\mathcal{PC}_{2,t}$ and then adding them together, without having to revisit the SMODD (Lemma 1). In this way, other coefficients with smaller exponents can be determined from those with larger ones by Lemma 1.

2) Fast Algorithm: A fast algorithm for computing coefficients can be derived based on Lemma 1. Assume that the exponent of $\kappa_{i,t}$ and that of $\kappa_{i',t}$ differ only in place s, and also that $i_s - i'_s = 1$. Then, $\kappa_{i',t}$ can be computed from $\kappa_{i,t}$ by multiplying with the PCs of $\kappa_{i,t}$ the values of $\tilde{x}_{j_s}$ and adding them together, if the values of the PCs are stored while computing $\kappa_{i,t}$. Similarly, another coefficient whose exponent differs from $\kappa_{i',t}$ in one place only and by one can be computed, and so on.

An algorithm for computing all the coefficients can be formulated as shown in Fig. 5. It is assumed that the information about unassigned variables represented by the point of the lattice under consideration is stored while executing Line 10 so that this information can be used to compute the other coefficients in Lines 11–12. One way of implementing Lines 11–13 is based on optimally storing the PCs in hash tables. We call this algorithm "CompBy1Diff."

3) Increased Efficiency for Large Fields: An N-input function in GF(2) has 2^n lattice points. However, if this same function is encoded in GF(2^m), i.e., at the word level with m-bit word size, then the number of lattice points would come down to $2^{\lceil n/m \rceil}$. This is the total number of times that algorithm CompByPath will be executed. Hence, with large m, the number of lattice points would be small, implying that the total number of times the Algorithm CompByPath is executed will also be small. However, as m approaches N, the number of terms whose exponents differ by one also increases. This implies that algorithm CompBy1Diff will also be executed more frequently.

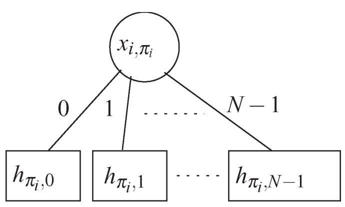

Fig. 6. Representing and storing coefficients.

We noticed a significant speedup by applying the aforementioned improvements, as detailed in Section IV.

4) Storing the Coefficients: Coefficients can be stored in an array of vectors, where each vector corresponds to a single term in the polynomial. Clearly, this method is exponential, unless expensive optimization routines are carried out. Instead, we store the coefficients as they are computed in a reduced and shared Galois polynomial decision diagram (SGPDD). Any function in GF(N) can be represented (expanded) based on the following polynomial form: $f(x_1, \ldots, x_i, \ldots, x_n) = \sum_{r=0}^{N-1} x_{i,\pi_i}^r h_{\pi_i,r}$. Here, $h_{\pi_i,j}$ is the coefficient corresponding to the jth term in π_i polarity. This can be represented by means of the GPDD as shown in Fig. 6. Here, the terminal nodes represent the coefficients over GF(N), whereas the edges represent the exponents of x_i. The SGPDD can be reduced as follows. Given two nodes v and w such that $\text{child}_i(v) = w$ and $0 \leq i \leq N-1$, 1) if all the nonzero edges of w point to the terminal node 0, then reconnect $\text{child}_i(v)$ to $\text{child}_0(w)$ and delete w, and 2) share equivalent subgraphs. A GPDD, which is reduced based on these two rules, can be shown to be canonic and minimal under a fixed variable ordering. A GPDD reduction algorithm, with linear time complexity in the number of nodes, has been developed for GF(N) and any polarity, with N being any power of a prime number. This algorithm is analogous to the BDD reduction algorithm [9], with the exception that the reduction rule has been replaced with the aforementioned rule 1) while the sharing rule remains the same and, obviously, the number of children has been extended to N. Note that the SGPDD reduces to the functional decision diagram over GF(2) [35]. It also has integer interpretation as TED [28].

Fig. 7(a) shows a reduced and shared GPDD for the polynomial $f(a, b, c) = ac + b^2c$ in GF(4) in zero-polarity. Each internal node has four children. The edges are labeled 0–3. However, edges terminating at the terminal node 0 are not labeled for brevity. Fig. 10(a) shows another reduced SGPDD for the function $f(a, b, c) = a + \beta bc^3 + \beta a^2c^3$.

Note that the algorithms for computing and representing the coefficients are independent of the variable ordering of the SMODDs. This follows from the fact that, owing to the commutative property of finite fields, the term $\theta_{i,t}$ in Theorem 1 is independent of the order of the variables in which the

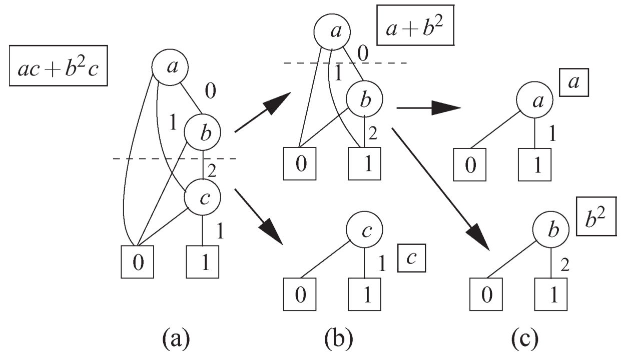

Fig. 7. Factorizing from SGPDD.

calculations are carried out. By similar reasoning, this is also true for Lemma 1 and the algorithms based on these.

III. Synthesis and Optimization

Once the SGPDD is obtained, circuits are synthesized by decomposing and factoring the SGPDD based on finding cuts within the SGPDD. A cut is a partitioning of the nodes in the SGPDD into two sets T and B, where T contains internal nodes and the root and B contains external, internal, and the last nodes, i.e., internal nodes which have the external nodes as their children. A cut passes through an SGPDD horizontally and does not cross an edge more than once. Effectively, a cut can factorize an SGPDD realizing a function f in GF(2^m) as $f = D \cdot Q + R$. Cut-based algorithms have been used for synthesis in the Boolean domain (see, e.g., [12]). In this paper, we quickly factorize a polynomial over GF(2^m) based on cuts on their SGPDDs to construct an expression DAG-based multiple-output shared netlist. The netlist constitutes two types of nodes: internal nodes which can either be GF(2^m) adders or multipliers, or external nodes which can only be constants and variables in GF(2^m). The internal nodes can have two children. Fig. 8(e) shows the netlist for the expression $ab^2 + bc$. The netlist is further synthesized based on additional factorization and optimization. Being in multiple-valued GF(2^m) domain, finding proper cuts is a much more difficult problem than in the Boolean domain. We found out that structural factorization of the netlist constituting only two input adders and multipliers works well in conjunction with the cut-based decomposition.

A. Decomposing from SGPDD

Given a variable x and an SGPDD for a function f, there are two types of decomposition possible: 1) multiplicative, which represents f as $f = X \times Y$, and 2) additive, which represents f as $f = W + Z$. Here, X, Y, W, and Z are SGPDDs. To perform a decomposition, a cut is performed above the nodes representing x. Let v_x be a node representing x. To obtain the multiplicative decomposition, all the paths in the original SGPDD from nodes above the cut leading to v_x are reconnected to the terminal node 1, and the result is reduced. This gives X. Y is simply the SGPDD rooted at v_x. To obtain the additive decomposition, all the paths in the original SGPDD from nodes above the cut leading to v_x are reconnected to the terminal

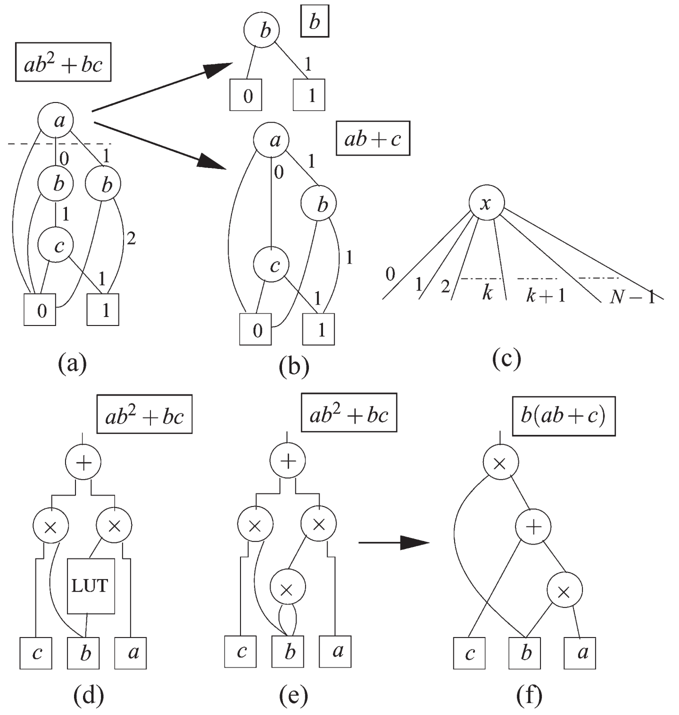

Fig. 8. Constructing the expression tree.

node 0, and the result is reduced. This gives W. Z is obtained by reconnecting all the paths from nodes above the cut that do not pass through v_x to the terminal node 0 and reducing the result. The proof for this reasoning is straightforward and has been left out for brevity. The following example demonstrates the basic idea.

In the SGPDD of Fig. 7(a), the cut is shown with the horizontal broken line above the node c. Note that cuts can also be made above node b. However, the cut above c has been made because of a specific reason. Here, the node c is a dominator node [12], i.e., all the paths ending in a particular terminal node, the terminal node 1 in this case, pass through node c. Given a dominator node v, v can be trivially factored out of an SGPDD by reconnecting all the paths leading to v to the terminal node 1 and detaching the subgraph rooted at v as another SGPDD. This results in a multiplicative decomposition over GF(2^m). After cutting the SGPDD, each of the components is reduced. While the bottom SGPDD in Fig. 7 cannot be decomposed further, the top one can be decomposed as two SGPDDs: one rooted at a, and the other rooted at b [Fig. 7(c)]. Here, the cut is performed above node b, which results in an additive decomposition over GF(2^m). The additive decomposition is performed as follows: all the paths leading to node b are reconnected to terminal node 0, which results in the SGPDD rooted at a after reduction. Again, all the paths passing around b are reconnected to the terminal node 0, which results in the SGPDD rooted at b after reduction. These nodes are not further decomposed but, instead, added to the netlist as $c \times (a + b^2)$, which requires one multiplier and one adder over GF(2^m). The exponent b^2 can be implemented in two ways: either using an additional multiplier for the exponentiation or using an m-input m-output LUT since we are dealing with GF(2^m). Our tool can do either way depending on which option is given.

```
1   Algorithm DecompGpdd{
2       Let NodeQ be a queue initially empty;
3       Let root be the root of the SGPDD;
4       Let EtRoot be the root of the netlist;
5       Add root to the netlist and push root on to NodeQ;
6       do{
7           Remove head of NodeQ and assign to root;
8           Construct parent list of root;
9           if(ParentList > 0){
10              Q = node with the min/max parents;
11              D = AssignPassAroundZ(Q);
12              Reconnect all edges in D leading to Q to 1;
13              R = AssignPassThroughZ(Q);
14              D = reduce(D);
15              R = reduce(R);
16              Add Q, D, and R to the netlist;
17              Push Q, D, and R on to NodeQ;
18          }
19      }while(NodeQ is not empty);
20      EtRoot = optimize(EtRoot);
21  }
```

Fig. 9. SGPDD decomposition algorithm.

A more general case appears in Fig. 8. Here, the function considered is $ab^2 + bc$. Decomposition of expressions like this cannot be done based on the approach in [12], owing to the presence of the exponent. Fig. 8(b) shows how a multiplicative decomposition is carried out if the cut is performed above the nodes b. First, the exponent of the term to be factored out is determined, which, in this case, is 1 (since $b = b^1$). Now, the exponents corresponding to the edges of the nodes representing b are reduced by one, which, in effect, implies that the edge 1 of the left node b is replaced with an edge 0, whereas the edge 1 is reconnected to the terminal node 0. The edge 2 of the right node b is replaced with an edge 1, whereas the edge 2 is reconnected to the terminal node 0. Reduction of this GPDD results in the bottom SGPDD of Fig. 8(b), whereas the top SGPDD corresponds to the factored-out variable b.

In general, given an SGPDD rooted at node x, as shown in Fig. 8(c), if we want to factor out the term x^k, we first need to perform an additive decomposition about x^k to obtain two SGPDDs as $(1 + x + x^2 + \cdots + x^{k-1}) + (x^k + x^{k+1} + \cdots + x^{N-1})$: one representing $1 + x + x^2 + \cdots + x^{k-1}$ and the other representing $x^k + x^{k+1} + \cdots + x^{N-1}$. The term x^k can be factored out by using a multiplicative decomposition from the second SGPDD as outlined in the previous paragraph.

The synthesis techniques may depend on the variable ordering of the SGPDDs. However, the dependence is not straightforward because factorizable terms, such as those shown in Fig. 8, can be determined irrespective of the variable reordering. In contrast, the reordering may result in an explosion in the node complexity without revealing any factorizable term. To avoid such potential node explosions in the SGPDDs and to retain the ability to determine all factorizable terms, in this paper, we perform factorization of the exponents on the netlist, which we obtain by using a fast greedy heuristic algorithm as outlined in Fig. 9.

Much of the algorithm is self-explanatory. The algorithm decomposes a function f as $f = D \cdot Q + R$. In Line 8, the list of parents is computed for the SGPDD rooted at root. Lines 10–17 are executed for nodes with at least one parent. Line 10 selects the node with the least or most number of parents, depending on which option is given.

Observation 3.1: We have observed that if the minimum number of parents is selected, then, for multiplier circuits, the proposed technique produces results that are close to hand-optimized multipliers in GF(2^m), e.g., the Mastrovito multipliers [2], [6]. In addition, it produces good results for polynomials with repeated multiplication and exponentiation. For other circuits such as parallel integer adders, selecting the maximum number of parents produces better results. This heuristic is based on the fact that, with integer adders, significant factorization is possible, which benefits from initially factoring out nodes with a high degree of sharing.

The cut is performed above the node Q. Lines 11 and 13 perform additive decomposition by reconnecting all the paths coming from nodes above the cut not leading to (Line 11) and leading to (Line 13) Q to the terminal node 0. Line 12 performs multiplicative decomposition. These steps are repeated for each of the components Q, D, and R. Line 20 performs an efficient netlist optimization as outlined in Section III-C. Note that no assumption is made regarding the word size or the polarity.

Fig. 7 shows the decomposition obtained from the SGPDD of Fig. 7(a) using this algorithm. Fig. 8(d) (exponent realized with LUT) and (e) (exponent realized with repeated multiplication) shows the resulting netlist after decomposing the SGPDD of Fig. 8(a).

Fig. 10 shows how this algorithm would proceed in decomposing an example function $f(a, b, c) = a + \beta bc^3 + \beta a^2 c^3$ over GF(4). Fig. 10(a) shows the initial SGPDD. The cut is performed above node c because it has the largest number of parents. First, an additive decomposition is performed to separate a and $\beta bc^3 + \beta a^2 c^3$, as shown in Fig. 10(b). The top SGPDD in Fig. 10(b) is not decomposed further, whereas the bottom one is decomposed further by performing a cut above node c. After reduction, this results in a multiplicative decomposition as shown in Fig. 10(c). While the bottom SGPDD in Fig. 10(c) is not further decomposed, the top one is decomposed, resulting in an additive decomposition of a^2 and b, thus completing the process. Note that with each step of the decomposition process, the netlist is constructed, which initially stores the address of the root of the SGPDD in Fig. 10(a). The result is $f(a, b, c) = a + \beta c^3 (b + a^2)$.

If algorithm DecompGpdd is unable to find any factors, then the resulting netlist will have the form of (2), which will be factorized and optimized based on the algorithms presented in the following, assuming that the polynomials can be factorized at all.

B. Factorizing Netlists

Common factors are determined by walking through chains of multipliers following chains of adders. A chain of adders is a nonempty ordered list of adders $\langle A_i, A_{i+1}, \ldots, A_{i+r} \rangle$ such that A_i forms a chain if $r = 0$; otherwise, A_{j+1} appears as one of the inputs to A_j for $i \leq j < i + r$. A chain of multipliers is an ordered list of multipliers $\langle M_k, M_{k+1}, \ldots, M_{k+s} \rangle$ such that M_{j+1} appears as one of the inputs to M_j for $k \leq j < k + s$. A

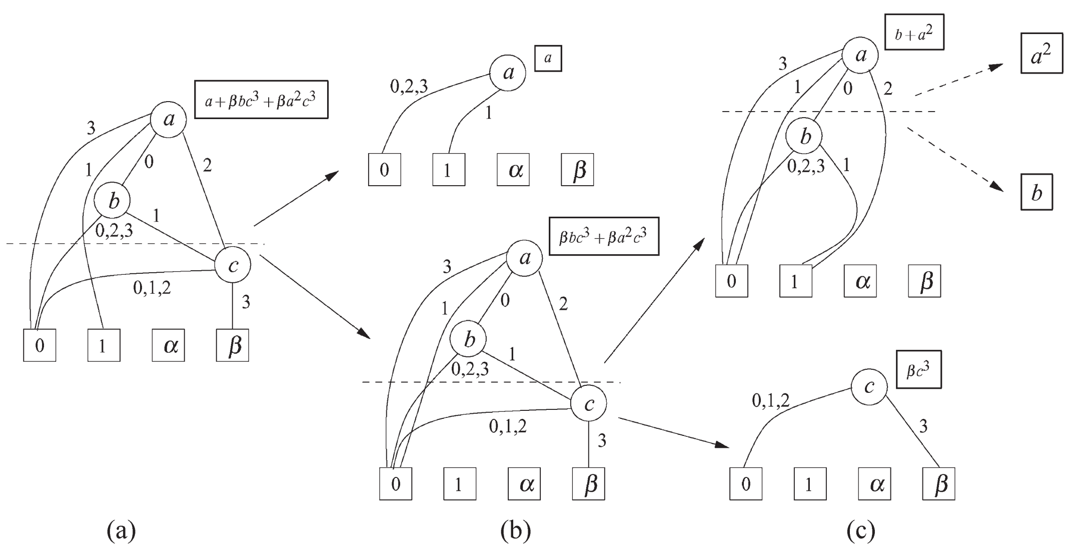

Fig. 10. Decomposition—an example.

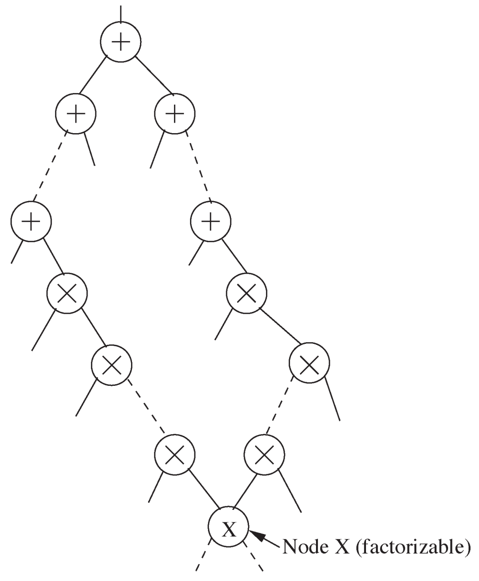

Fig. 11. Generic factorizable node.

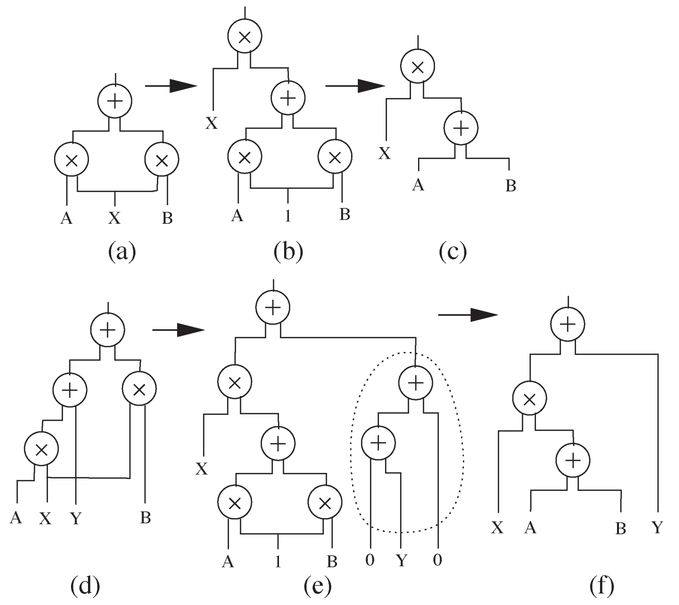

Fig. 12. Factorizing from netlist.

chain of multipliers can be empty, in which case, a term (i.e., a node u or a subgraph rooted at u) is considered as a chain. This is done to ensure uniformity of the underlying implementation, e.g., in situations where we may have an adder node whose inputs are coming from either a variable node or a constant node. In this case, the variable or constant nodes are considered as chains with a single element (singletons).

Theorem 2: Let $C_{A_1} = \langle A_i, A_{i+1}, \ldots, A_{i+r}\rangle$ and $C_{A_2} = \langle A_j, A_{j+1}, \ldots, A_{j+s}\rangle$ be two chains of adders. Let $C_{M_1} = \langle M_k, M_{k+1}, \ldots, M_{k+t}\rangle$ and $C_{M_2} = \langle M_l, M_{l+1}, \ldots, M_{l+w}\rangle$ be two chains of multipliers. Let M_k and M_l be one of the inputs to A_{i+r} and A_{j+s}, respectively. Let X be a subgraph such that X appears as one of the inputs to M_{k+t} and M_{l+w}. If $A_i = A_j$, then X is factorizable.

The proof is straightforward and has been left out for brevity. The situation is shown in Fig. 11.

The factorization is carried out as shown in Fig. 12. Fig. 12(a) shows the structure $(AX + BX)$, which can be factored as $X(A + B)$ as shown in Fig. 12(b) and (c). Fig. 12(d) shows a more general structure of the form $Z = ((AX + Y) + BX)$. Clearly, X is factorizable. To factorize X, Z is restructured as $Z = ((AX + BX) + Y)$ [Fig. 12(e)], and then, the factorization is carried out as $Z = ((X(A + B)) + Y)$ [Fig. 12(f)]. The structure within the circle in Fig. 12(e) is the network of Fig. 12(d) after its pointers have been readjusted. After restructuring the circuits, the netlist optimization technique of Section III-C is applied to obtain the final circuits. This type of restructuring also allows optimization of adders, e.g., given $W = ((X + Y) + X)$ after restructuring and optimization, we get $W = (X(1 + 1)) + ((0 + Y) + 0) = Y$. The factorization is carried out based on unique paths. This allows correct factorization of terms with exponents. For example, factorizing the netlist of Fig. 8(e) yields the network of Fig. 8(f). This result is the same as decomposing the SGPDD shown in Fig. 8(a) and (b).

The algorithm for factorization proceeds by trying out all possible factorizations and only stops when there are no more terms which can be factored out. Factors are selected on a first-come first-served basis. Note that no assumption regarding the word size or the polarity has been made, i.e., the same approach works for bit-level as well as word-level circuits in any polarity.

The original polynomials can be trivially constructed from the netlists obtained after decomposing their SGPDDs or after decomposition followed by factorization: 1) Traverse the netlists from the external (variable or constant) nodes toward the root node, i.e., from the inputs to the outputs of the netlists in the topological order; 2) construct the expressions for each node successively; and 3) the expression corresponding to the root node will correspond to the polynomial from which the netlist was constructed.

For example, considering the netlist in Fig. 8(f), if we visit this netlist from the inputs to the outputs, this would give us the expression $(b \cdot a + c) \cdot b = ab^2 + bc$, which is the original polynomial.

C. Optimizing Netlists

The netlists are optimized using an efficient optimization algorithm as follows. The netlists are processed recursively from the external (i.e., variable or constant) nodes toward the root node, i.e., in the topological order. Each node is visited exactly once. For each node u, the following is done. If u is already processed, then its reference is returned so that it can be shared; otherwise, its information is stored in a hash table for sharing. This step is necessary for sharing of hardware resources. If u is an internal (i.e., GF(2^m) multiplier/adder) node, then let v_0 and v_1 be its two children. If both v_0 and v_1 are constants, then replace u with v_0 op v_1, where op is either addition or multiplication over GF(2^m), i.e., perform constant propagation. Replace u with $v_i (i \in \{0, 1\})$ if $u = v_i \times 1$ or $u = v_i + 0$. Replace u with zero if $u = v_i \times 0$ or $u = v_i + v_i$. Note that this algorithm is analogous to the BDD reduction algorithm in [9], which can be argued to be optimal under a fixed variable ordering. Analogously, this algorithm can be shown to be optimal w.r.t. two input addition and multiplication over GF(2^m) under a fixed netlist transformation. Also note that no assumption is made regarding the polarity or the word sizes, i.e., it can be applied to the bit as well as word levels correctly with either repeated multiplication or LUT-based exponentiation and in any polarity. This algorithm can be applied for single- as well as multiple-output functions.

For example, given the netlists of Fig. 12(b) and (e), this algorithm will yield the netlists of Fig. 12(c) and (f), respectively.

D. Exponentiation

Exponentiation is carried out in two ways: 1) using the repeated multiplication rule and 2) using LUTs depending on which option is given. If rule 1) is specified, then exponents are generated using four different heuristics depending on which option is given: 1) Exp1: balanced tree of multipliers, e.g., given $5x^8$, a balanced tree of multipliers is generated for the complete operation; 2) Exp2: the actual exponent is a balanced tree, but terms like $5x^8$ are carried out using one additional multiplier as $5 \times x^8$; 3) Exp3: square and multiply; and 4) Exp4: square and multiply incrementally, e.g., given x^4, x^5, and x^{11}, generate x^4 using square and multiply, $x^5 = x^4 \times x$ and $x^{11} = x^5 \times x^6$, where $x^6 = x^5 \times x$.

If the LUT option is given, then the polynomials of the form $p(x) = \sum_{i=0}^{2^m-1} c_i x^i$, where $c_i \in$ GF(2^m), are generated as a single m-input m-output LUT. For example, given an expression $a^3(\beta b^9 + b) + c(\beta b^9 + b)$ in GF(16), it is first factored as $(\beta b^9 + b)(a^3 + c)$. Then, the result is implemented using two four-input four-output LUTs, one for each $\beta b^9 + b$ and a^3, and one multiplier and one adder over GF(16).

The idea behind the different heuristics for generating the exponents is to allow effective term factoring and sharing from which the factorization and optimization techniques of Sections III-B and C can benefit.

E. Overall Approach

The approach proceeds as follows.

Step 1) Store initial specification as SMODDs.
Step 2) Apply the technique of Section II to compute and store the coefficients as reduced and shared GPDDs.
Step 3) Apply the techniques of Section III-A to decompose the SGPDDs into the netlist.
Step 4) Apply the technique of Section III-B to further decompose and optimize the netlist.

During this process, the technique of Section III-C is applied between Steps 3) and 4) to obtain an initial and, also, after Step 4) to obtain the final result. The best result is retained.

The algorithm can start from any one of the aforementioned steps. If the initial DAG is available, then start from Step 2). If the initial polynomials are available as DAGs in GF(2^m), then start from Step 3). If the netlist constituting elements in GF(2^m) is available, then start from Step 4). Alternatively, if the netlist constituting the elements in GF(2^m) is available, then we can also start from Step 2). In this case, the DAGs constituting the coefficients are constructed from this netlist.

For $m = 1$, this approach performs gate-level synthesis and optimization. For $m > 1$, it performs word-level synthesis and optimization. For word-level synthesis and optimization ($m > 1$), we propose a two-step solution (Fig. 1).

Step 1) Apply this technique at the word level to synthesize and optimize the polynomials. Once the polynomials have been synthesized at the word level, the basic blocks such as the adders, multipliers, and exponentiation circuits can be "filled-up" by these circuits synthesized with the proposed technique at the gate level, or by presynthesized macro blocks based on techniques specifically tailored for multipliers and exponentiation circuits in GF(2^m) [2], [6], [11], [21]–[23], [29].
Step 2) Apply this technique at the gate level on the net result for further minimization based on the properties of GF(2).

No assumption is made regarding the word size or polarity for this approach, i.e., it works for gate as well as word levels in any polarity.

F. Easily Verifiable Hardware

Owing to the canonic nature of the DAGs for storing the coefficients, the proposed technique is stable and independent of the starting point if the designs are initiated from (1) in Fig. 1. This is a major advantage over many industrial tools, which can produce different results for the same polynomial if the initial spec is altered. This has an interesting side effect—the resulting netlist will always be structurally identical for polynomials with identical functionality. This can lead to easily verifiable hardware. For example, assume that we have two netlists N_1 and N_2, both generated with the proposed technique. Also assume that N_1 is generated from polynomial P. To verify that N_2 is functionally equivalent to N_1 and, hence, realizes P, we only need to check if N_2 structurally matches N_1. This can be done on a node-by-node match between the two netlists (i.e., test for graph isomorphism) in linear time in the number of nodes. Alternatively assume that we have two netlists N_1 and N_2 obtained from polynomials P_1 and P_2, respectively. If N_1 is isomorphic to N_2, then we can conclude that $P_1 = P_2$. If the netlists are saved as VHDL files, then the equivalence of N_1 to N_2 can be tested simply by matching their VHDL files line by line, e.g., using the "diff" tool in Linux/UNIX. Hence, the proposed technique can make it easier to verify the hardware designed from the polynomials.

IV. Experimental Results

The algorithms in this paper have been implemented in C++ (Gnu C++ 3.2.2-5) on a computer with 640-MB RAM and a 600-MHz Athlon processor running RedHat Linux with kernel-2.4.20-43.9. The Synopsys design compiler was run on a dual-processor Pentium-4 Linux machine with 2-GB RAM and kernel-2.4.21-20.EL. The benchmarks were stored as two-level AND–OR PLAs to enable us to determine how effective the proposed technique is in optimizing area, power, and delay. In addition, the Synopsys design compiler understands this format. The PLAs were read into SMODDs, and the algorithm of Section III-E was applied from Step 1). The SGPDDs were constructed in the natural order of the variables during the synthesis process. After optimization, the results were saved in the VHDL format, which were passed to the Synopsys design compiler (power was estimated with the Synopsys power compiler). The PLAs were also passed directly to the Synopsys compiler for it to optimize without the aid of the proposed technique. A combination of C++ codes and perl scripts was used to generate the test cases and then execute the tools with the different test cases, word sizes, and PPs.

Unless otherwise specified, the tool assumes the following PPs by default for generating the fields: (2, 7), (3, 11), (4, 19), (5, 37), (6, 67), (7, 137), and (8, 285). Here, the first number m in the ordered pair (m, d) is the bit width and determines the field size, i.e., GF(2^m), and the second number d is the decimal representation of the PP. It is also possible to specify any other word sizes and valid PPs as command line options.

We have used the basic command: `compile` for the Synopsys compilers, i.e., without any constraints. We have also compared with the command: `compile − map_efforthigh− area_efforthigh` with a virtual clock. In the second case, the compiler reported a higher overall chip area without affecting the area improvements brought about by the proposed technique.

1) Multipliers Over GF(2^m): Table I shows the results for GF(2^m) multipliers $(2 \leq m \leq 8)$ for all the 51 PPs in the 0.18-μm CMOS technology. Column 2 represents the PPs, whereas column 5 represents the area, delay, and power reported by the Synopsys design compiler without the aid of the proposed technique. The column with the heading "Proposed Technique" shows the result of applying the proposed technique first, and then applying the Synopsys compiler on the resulting VHDL files. The letters "a" and "m" represent two-input EXOR and AND gates, respectively. Here, area, delay, and power are in 10^{-6} mm^2, nanoseconds, and milliwatts, respectively. Power was estimated at 1.8 V. Significant area, delay, and power reduction is observable (as much as 57 times for polynomial 487). Clearly, the number of AND gates is m^2 for all the cases. The number of EXOR gates varies from approximately $m^2 - 1$ for trinomials, e.g., 19, 25, 37, 67, 131 to $m^2 + k$, where k is a constant, for pentanomials and polynomials with more than five terms. As compared with [29], which reports 2 $m^2 - 1$ two-input AND gates and 2 $m^2 - 3$ $m + 1$ EXOR gates, the proposed technique produced better results. In the 0.18-μm CMOS technology for the 8-bit multiplier, this technique reports a maximum area of 0.002906 mm^2, whereas the technique in [29] reported 0.0128 mm^2, i.e., about 4.4 times better (5.2 times better for polynomial 391). It also reports about two-times reduced delay. For the 4-bit case, the proposed technique reports about 5.1-times better area. The techniques in [2], [6], and [23] have reported only theoretical results, with m^2 two-input AND gates and $m^2 - 1$ (for trinomials) to $m^2 + k$ EXOR gates depending on the number of terms in the polynomials and their positions, without any implementation. However, we were able to compare the examples given in [2] and [6]. For example, the technique in [6] reported 18 EXOR and 16 AND gates for the GF(2^4) multiplier in the PP 25, whereas this technique reports 15 EXOR and 16 AND gates. As compared with [2], the proposed technique reported similar result for the polynomial 171 in GF(2^7). Note that Table I closely approximates the theoretical results reported by these techniques despite the fact that the proposed technique is a heuristic synthesis algorithm for polynomials, whereas these techniques are designed only for hand-synthesizing the multipliers in GF(2^m).

Columns 3 and 4 show the total number of nodes and paths required by the SMODD and SGPDD. The node count in the SMODDs was approximately doubling for each added bit, whereas the SGPDDs were increasing much more slowly. In addition, for the 7- and 8-bit multipliers, the node count in the SMODDs was well over one order of magnitude higher than that in the corresponding SGPDDs.

The last column of Table I shows the total number of test vectors required by the automatic test pattern generation tools

TABLE I
2–8-bit GF(2^m) MULTIPLIERS WITH ALL 51 PPs

Word	Prim	SMODD	SGPDD	Synopsys® only	Proposed Technique		Testing (Num. Vectors)	
size	Poly	(nodes,paths)	(nodes,paths)	(area,delay,power)	(a,m)	(area,delay,power)	Synopsys®	SIS [19]
2	7	(10,8)	(7,5)	(112.9,0.3,0.1)	(3,4)	(112.9,0.3,0.1)	7	7
3	11	(31,39)	(14,12)	(412.9,0.7,0.5)	(8,9)	(277.4,0.5,0.3)	8	9
	13	(31,39)	(14,13)	(403.2,0.9,0.5)	(8,9)	(287.1,0.7,0.4)	9	8
4	19	(82,160)	(23,22)	(1283.8,1.1,1.7)	(15,16)	(532.2,0.6,0.6)	11	10
	25	(82,160)	(23,26)	(977.4,1.4,1.2)	(15,16)	(522.5,1.0,0.7)	11	10
5	37	(201,605)	(35,36)	(2383.7,2.0,3.1)	(25,25)	(838.7,0.9,1.1)	13	13
	41	(201,605)	(35,38)	(2354.7,1.9,2.9)	(25,25)	(841.9,0.9,1.2)	12	13
	47	(201,605)	(42,51)	(5951.2,2.5,7.3)	(32,25)	(970.9,1.2,1.4)	15	14
	55	(201,605)	(41,46)	(5260.9,2.7,6.3)	(31,25)	(967.7,1.1,1.4)	13	14
	59	(201,605)	(41,50)	(4909.3,2.7,6.5)	(31,25)	(993.5,1.4,1.4)	14	14
	61	(201,605)	(42,49)	(5015.8,2.4,6.6)	(32,25)	(977.4,1.3,1.4)	16	15
6	67	(472,2184)	(47,51)	(6606.0,2.9,8.5)	(35,36)	(1206.4,1.0,1.6)	18	13
	91	(472,2184)	(58,83)	(18047.2,5.7,22.9)	(46,36)	(1387.0,1.8,2.1)	16	14
	97	(472,2184)	(47,71)	(5164.2,2.8,7.2)	(35,36)	(1270.9,2.5,1.9)	21	13
	103	(472,2184)	(57,74)	(9886.4,3.6,12.4)	(45,36)	(1377.3,1.5,2.0)	15	12
	109	(472,2184)	(59,73)	(11118.6,3.9,14.5)	(47,36)	(1367.7,1.4,2.0)	14	14
	115	(472,2184)	(57,75)	(15657.0,4.4,20.0)	(45,36)	(1438.6,1.4,2.1)	15	17
7	131	(1079,7651)	(62,70)	(30866.1,7.7,39.1)	(48,49)	(1699.9,1.3,2.4)	20	15
	137	(1079,7651)	(64,73)	(30356.5,8.3,39.9)	(50,49)	(1709.6,1.6,2.5)	19	18
	143	(1079,7651)	(81,118)	(33375.6,8.0,40.3)	(67,49)	(1925.7,1.9,3.0)	18	14
	145	(1079,7651)	(64,76)	(33469.1,8.1,42.3)	(50,49)	(1703.1,1.7,2.5)	20	17
	157	(1079,7651)	(79,120)	(35020.6,7.4,42.7)	(65,49)	(1964.4,2.1,3.1)	26	14
	167	(1079,7651)	(79,118)	(34978.7,7.7,39.7)	(65,49)	(1964.4,1.7,3.2)	20	16
	171	(1079,7651)	(78,102)	(39023.2,8.5,49.1)	(64,49)	(1906.3,1.4,2.9)	23	16
	185	(1079,7651)	(80,104)	(34797.9,8.7,42.1)	(66,49)	(2003.1,1.4,3.0)	23	16
	191	(1079,7651)	(89,131)	(34843.1,8.8,41.0)	(75,49)	(2077.3,1.8,3.2)	20	16
	193	(1079,7651)	(62,105)	(30321.0,8.0,38.8)	(48,49)	(1725.7,3.2,2.7)	21	13
	203	(1079,7651)	(78,114)	(33875.6,8.0,42.0)	(64,49)	(1922.5,1.5,3.0)	20	12
	211	(1079,7651)	(78,120)	(33440.1,8.2,40.3)	(64,49)	(1935.4,1.7,3.1)	18	15
	213	(1079,7651)	(79,129)	(33488.5,8.0,38.9)	(65,49)	(2009.6,1.8,3.3)	21	13
	229	(1079,7651)	(81,123)	(34578.6,8.2,41.3)	(67,49)	(1980.5,1.8,3.2)	17	12
	239	(1079,7651)	(86,114)	(34123.9,9.9,41.5)	(72,49)	(2041.8,1.6,3.1)	19	14
	241	(1079,7651)	(75,99)	(32298.3,8.7,36.7)	(61,49)	(1899.9,1.6,3.0)	20	19
	247	(1079,7651)	(86,118)	(36704.1,7.2,45.0)	(72,49)	(2106.3,1.6,3.2)	19	14
	253	(1079,7651)	(88,109)	(33782.0,8.3,41.3)	(74,49)	(2106.3,2.0,3.2)	20	16
8	285	(2422,26240)	(103,150)	(141689.3,18.8,130.9)	(87,64)	(2628.9,1.6,4.2)	26	14
	299	(2422,26240)	(101,160)	(152603.3,14.6,125.7)	(85,64)	(2609.5,1.7,4.1)	22	17
	301	(2422,26240)	(100,141)	(132595.5,15.0,125.4)	(84,64)	(2580.5,1.5,4.0)	17	15
	333	(2422,26240)	(100,160)	(140896.3,15.2,129.3)	(84,64)	(2574.0,2.0,4.1)	25	18
	351	(2422,26240)	(116,162)	(135107.6,17.9,121.5)	(100,64)	(2757.9,1.5,4.3)	24	17
	355	(2422,26240)	(104,165)	(143195.4,17.4,126.0)	(88,64)	(2612.7,1.8,4.2)	21	18
	357	(2422,26240)	(100,158)	(144472.7,14.8,133.5)	(84,64)	(2548.2,1.5,4.0)	25	18
	361	(2422,26240)	(104,170)	(135227.9,16.3,125.6)	(88,64)	(2628.9,1.7,4.2)	27	17
	369	(2422,26240)	(105,153)	(132405.8,17.6,130.2)	(89,64)	(2722.4,1.9,4.3)	30	16
	391	(2422,26240)	(95,155)	(130577.4,15.9,126.5)	(79,64)	(2441.8,2.2,3.8)	18	16
	397	(2422,26240)	(103,158)	(139992.8,14.4,122.3)	(87,64)	(2532.1,2.0,4.0)	24	16
	425	(2422,26240)	(110,181)	(141879.6,15.6,118.9)	(94,64)	(2690.2,2.0,4.3)	23	15
	451	(2422,26240)	(95,152)	(139612.7,17.6,123.9)	(79,64)	(2493.4,1.8,3.9)	23	13
	463	(2422,26240)	(106,160)	(141228.2,17.8,122.9)	(90,64)	(2661.1,1.9,4.1)	24	16
	487	(2422,26240)	(108,155)	(150703.6,16.7,138.8)	(92,64)	(2632.1,1.9,4.1)	20	16
	501	(2422,26240)	(120,174)	(129935.6,16.0,119.2)	(104,64)	(2906.3,2.3,4.7)	22	17

in the Synopsys compilers (TetraMAX) and SIS to test the multipliers. The fault coverage was always 100% for stuck-at faults. We used the random pattern generation scheme of SIS (command: `atpg − v3`) for determining the number of test patterns with SIS. Note that the Synopsys compilers seemed to map the AND–EXOR-based VHDL formats to a network comprising elements from its technology libraries. For SIS, the circuits were provided in the BLIF format, which constituted LUTs for each two-input AND and EXOR gates.

Note that we do not present any results for adders over GF(2^m) because the proposed technique trivially synthesizes adders optimally, i.e., with m two-input EXOR gates.

2) Exponentiation Over GF(2^m)*:* Fig. 13 shows the result of applying the technique for optimizing the term ab^2 in GF(2^m), where $2 \leq m \leq 8$. The total number of AND gates is exactly m^2. The total EXOR gate count is approximately $m^2 + m$ up to $m = 7$. For $m = 8$, the EXOR gate count worsens. Note that techniques such as in [13], which are tailored for hand-synthesizing ab^2, have reported total EXOR and AND gate counts of $(m+1)^2$ each.

Fig. 14 shows how the total two-input gate count varies with the exponent x^y, where $2 \leq y \leq 255$, over GF(2^8). There are distinct regions noticeable which seem to be repeated with the exponents. For exponents 2, 4, 8, 16, 32, and 128, only two-input EXOR gates were required, with 11 being the minimum (for exponents 2 and 128) and 21 being the maximum (for exponent 32). The hardest exponent was 251, which required 410 and 251 two-input EXOR and AND gates, respectively, i.e.,

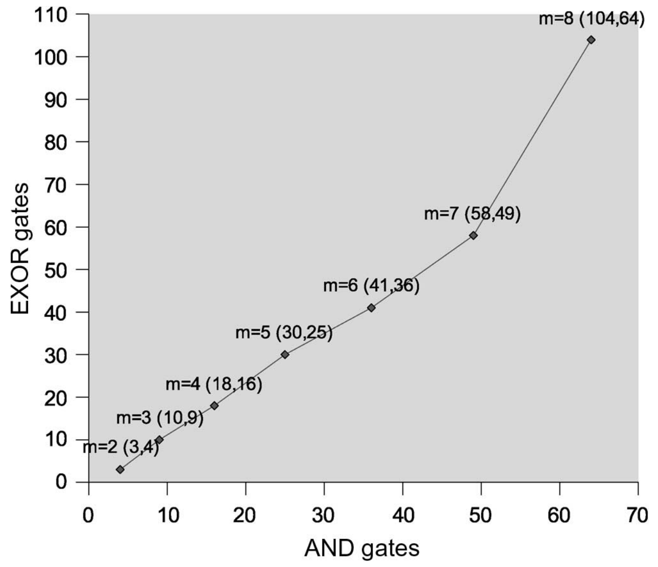

Fig. 13. Gate count for ab^2.

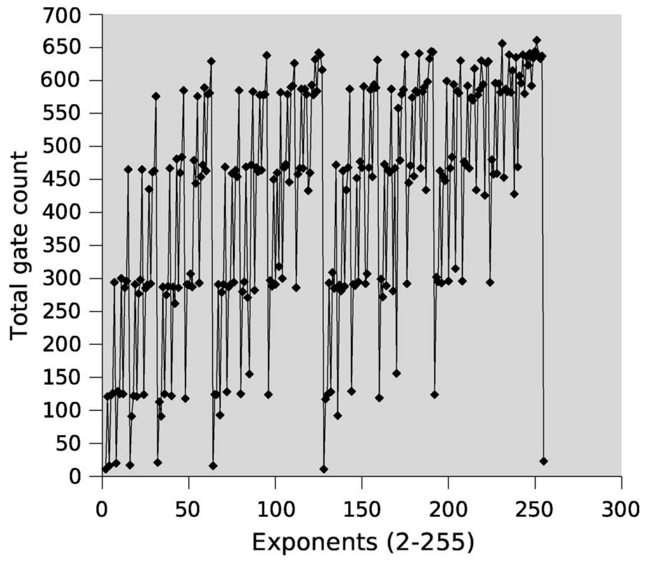

Fig. 14. Variation of two-input gates with exponent over GF(2^8).

661 gates in total. Exponent 255 requires 14 two-input EXOR gates and 9 two-input AND gates, which can be realized with a single eight-input OR gate.

We have also applied the proposed technique on ab^y for $2 \leq y \leq 255$ in GF(2^8) at the gate level. The minimum area is 148.7×10^{-6} mm^2 for ab^{255}, and the maximum area is $60\,902 \times 10^{-6}$ mm^2 for ab^{251}. We could not compare all the cases produced by the Synopsys compilers only since, for most of the cases, it was taking far too long without the aid of the proposed technique, e.g., for ab^{79}, we had to terminate the compiler after 51 h. We could apply the Synopsys compilers for certain cases, e.g., for ab^{68}, it reported $790\,785 \times 10^{-6}$ mm^2 and 24.1-ns delay without the proposed technique, whereas with the proposed technique, it reported $11\,531 \times 10^{-6}$-mm^2 area and 3.58-ns delay, i.e., about 68 times area and nearly one order of magnitude delay improvement, respectively. If the proposed technique is applied at the word level (8-bit word size), then for ab^{251} with b^{251} implemented as LUT, the gate count reduces from 2280 EXOR and 1275 AND gates down to a maximum of 497 EXOR and 315 AND gates. Here, the multiplier and the LUT are minimized at the gate level.

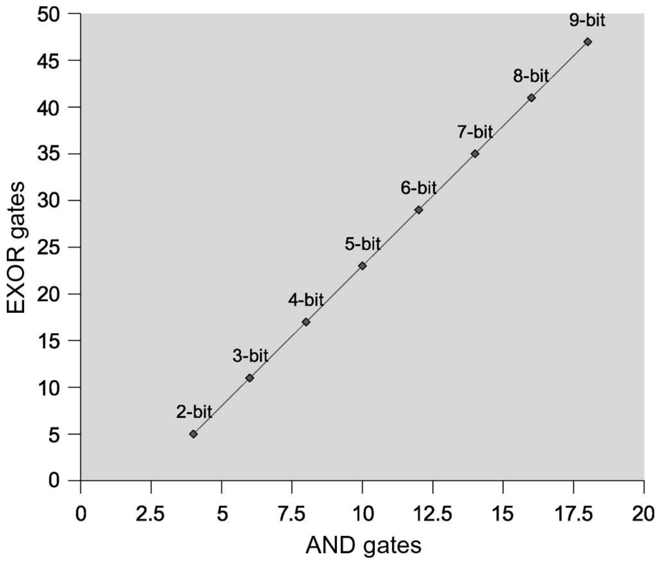

Fig. 15. Gate count for 2–9-bit integer adders.

TABLE II
2–9-bit PARALLEL INTEGER ADDERS

Integer Adder	Synopsys® only (area,delay,power)	Proposed technique (a,m)	Proposed technique (area,delay,power)
2-bit	(119.3,0.3,115.7)	(4,5)	(106.4,0.4,113.1)
3-bit	(197.8,0.7,214.3)	(11,6)	(216.1,0.81,241)
4-bit	(274.2,1.21,301.8)	(17,8)	(316.1,1.08,378.2)
5-bit	(383.8,1.22,468.5)	(23,10)	(416.1,1.4,515.8)
6-bit	(493.5,1.58,590)	(29,12)	(509.6,1.72,627.5)
7-bit	(638.7,1.61,791.1)	(35,14)	(609.3,2.04,752.8)
8-bit	(687.0,2.0,809.4)	(41,16)	(709.6,2.38,879)
9-bit	*(11877,3.7,13800)*	*(47,18)*	*(809.6,2.69,1000)*

3) Integer Arithmetic: Although this technique was primarily meant for the polynomials over finite fields, we also tested its performance for other types of circuits, e.g., gate-level arithmetic circuits by modeling each output with a multivariate polynomial over GF(2). Table II shows the results for the gate-level optimization of 2–9-bit parallel integer adders. Column 2 shows the results using only the Synopsys design compiler. Columns 3–4 show the results obtained with the proposed technique first in conjunction with the Synopsys compilers. Here, area, delay, and power are in 10^{-6} mm^2, nanoseconds, and microwatts, respectively. Column 2 represents the two-input EXOR and AND gate counts as "a" and "m", respectively. Up to 8 bit, the proposed technique does not bring any noticeable advantage. However, for the 9-bit adder, the proposed technique has improved the result by more than one order of magnitude. Fig. 15 shows that the adder scales linearly in terms of two-input gate count with the proposed technique.

For the integer multipliers, we noticed mixed results. For the smaller multipliers, it reported similar or worst results compared to the Synopsys compilers. For the larger multipliers, e.g., 8-bit ones, it reported up to 25%–35% area improvement. For example, for the 8-bit integer multiplier, we observed

TABLE III
WORD-LEVEL SYNTHESIS AND OPTIMIZATION

Field (*k*,pp)	Polynomial	Exp4 (a,m)	LUT (a,m,LUT)	Gate Level (a,m)
(6,67)	$a^{13} + b^{11}$	(1,10) [(356,360)]	(1,0,2) [(134,83)]	(136,83)
(6,67)	$a^6b^7 + a^7b^{10}$	(1,11) [(391,396)]	(1,2,4) [(289,207)]	(1295,631)
(6,67)	$a^{59}b^{35} + a^{29}b^{63}$	(1,27) [(951,972)]	(1,2,4) [(320,247)]	(1300,745)
(5,37)	$a^{30}b^{29}c + b^{29}c^{17}$	(1,22) [(555,550)]	(1,2,3) [(189,154)]	(1238,748)

area, delay, and power of 447 073.56 μm^2, 23.14 ns, and 976.2 mW, respectively, with the proposed technique, whereas without our technique, we observed area, delay, and power of 698 171.6 μm^2, 20.55 ns, and 403.3 mW, respectively. Note that in this instance, the power has gone up with our technique, whereas the area has gone down.

4) Generic Polynomials—Bit Versus Word Level: Table III shows the results for some sample multivariate polynomials. The polynomials have been synthesized at the word as well as at the gate levels. At the word level, Exp4 and the LUT-based techniques have been applied for the generation of exponents (Section III-D). The symbols "k", "pp", "a," and "m" represent the word size, PP, total number of two-input adders [EXOR gates in GF(2)], and multipliers [AND gates in GF(2)], respectively. The figures within "[]" represent the maximum number of two-input EXOR and AND gates required to implement the adders, multipliers, and LUTs. This figure is obtained by synthesizing the LUTs at the gate level with the proposed technique and then by adding the total number of two-input EXOR and AND gates required for the LUTs, adders, and multipliers (Table I). Clearly, the LUT-based algorithm is winning out, whereas Exp4 is in the second place. However, LUTs suffer from exploding in size quite rapidly, unless they are saved in a DAG-based form, e.g., the BDD, as they are created.

We noticed a significant speedup by incorporating algorithm GfCoeffHs1 (Fig. 5), as compared to algorithm CompByPath [5]. For example, in Table III, algorithm GfCoeffHs1 takes 0.63 and 129.23 s to compute the coefficients for the polynomials in rows 3 and 4, respectively. This figure is 22.51 and 1507.06 s, respectively, with algorithm CompByPath, i.e., more than one order of magnitude speedup with algorithm GfCoeffHs1. The speed improvement is also better with the larger fields.

5) Standard Benchmarks: We also tested it on standard benchmarks (gate level), e.g., the IWLS'93 set. It produced good results for those cases which have arithmetic circuits built-in, e.g., for rd84, t481, and alu4, it reported two-input gate counts as (a, m): (21, 21), (17, 10), and (778 650), respectively.

Note that in all the cases, many two-input EXOR gates were of the form $1 + x$, which is a single NOT gate but still counted as an EXOR gate.

We also tried to compare with SIS executed with script.rugged [19]. For $m > 5$, SIS was taking a very long time to complete for the multipliers, exponentiation, and integer adder circuits, and its execution had to be terminated. For the other cases, the proposed technique reported better results. At the worst case, we noticed about 20 min for the entire synthesis process including the Synopsys compilers with the proposed technique. On an average, it was a few minutes.

Note that the technique in [10] does not report any results on the polynomials over GF(2^m). It also does not report any synthesis results, i.e., chip area, delay, or power.

From a traditional logic synthesis point of view, it is generally believed that algebraic methods could be suitable for control circuits at the gate level, although our approach, despite being entirely algebraic, shows very good results for arithmetic circuits as well.

V. CONCLUSION

In this paper, we presented a novel synthesis and optimization technique for the polynomials over GF(2^m) for gate as well as word levels. The experimental results suggest that this technique can significantly reduce area, delay, and power (up to 68 times area, and significantly more than one order of magnitude delay and power). In addition, this technique can closely match techniques tailored for hand-synthesizing circuits in GF(2^m). Therefore, we can conclude that, if this technique is used in two steps, as outlined in Section III-E, on its own or in conjunction with the existing techniques specifically tailored for synthesizing multipliers, exponentiators, etc., over GF(2^m), then near optimal circuits can be designed for the polynomials over GF(2^m). This technique can also produce 100% testable circuits for stuck-at faults, which require small number of vectors (approximately 2 m) for testing. Also, this technique is capable of producing structurally identical netlists for circuits with the same function, which are independent of the initial specification. This property of the proposed technique may help with fast verification of circuits synthesized with it.

REFERENCES

[1] A. Dur and J. Grabmeier, "Applying coding theory to sparse interpolation," *SIAM J. Comput.*, vol. 22, no. 4, pp. 695–703, Aug. 1993.

[2] A. Halbutogullari and Ç. K. Koç, "Mastrovito multiplier for general irreducible polynomials," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 503–518, May 2000.

[3] A. Hosangadi, F. Fallah, and R. Kastner, "Optimizing polynomial expressions by algebraic factorization and common subexpression elimination," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 10, pp. 2012–2022, Oct. 2006.

[4] A. Jabir and D. Pradhan, "MODD: A new decision diagram and representation for multiple output binary functions," in *Proc. DATE*, Paris, France, Feb. 2004, pp. 1388–1389.

[5] A. Jabir and D. Pradhan, "An efficient graph based representation of circuits and calculation of their coefficients in finite field," in *Proc. IWLS*, Lake Arrowhead, CA, Jun. 2005, pp. 218–225.

[6] A. Reyhani-Masoleh and M. A. Hasan, "Low complexity bit parallel architectures for polynomial basis multiplication over GF(2^m)," *IEEE Trans. Comput.*, vol. 53, no. 8, pp. 945–959, Aug. 2004.

[7] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL: CRC Press, 1997.

[8] R. E. Blahut, *Fast Algorithms for Digital Signal Processing*. Reading, MA: Addison-Wesley, 1984.

[9] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[10] C. Files and M. Perkowski, "New multivalued functional decomposition algorithms based on MDDs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 19, no. 9, pp. 1081–1086, Sep. 2000.

[11] C. Paar, P. Fleischmann, and P. Soria-Rodriquez, "Fast arithmetic for public-key algorithms in Galois fields with composite exponents," *IEEE Trans. Comput.*, vol. 48, no. 10, pp. 1025–1034, Oct. 1999.

[12] C. Wang, V. Singal, and M. Ciesielski, "BDD decomposition for efficient logic synthesis," in *Proc. ICCAD*, 1999, pp. 626–631.
[13] C. H. Liu, N. F. Huang, and C. Y. Lee, "Computation of ab^2 multiplier in GF(2^m) using an efficient low complexity cellular architecture," *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. E83-A, no. 12, pp. 2657–2663, 2000.
[14] 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 GF(2^m)," *IEEE Trans. Comput.*, vol. 53, no. 3, pp. 375–380, Mar. 2004.
[15] D. Jankovic, R. Stanković, and C. Moraga, "Optimization of GF(4) expressions using the extended dual polarity property," in *Proc. ISMVL*, May 2003, pp. 50–55.
[16] D. K. Pradhan and A. M. Patel, "Reed-Müller like canonic forms for multivalued functions," *IEEE Trans. Comput.*, vol. C-24, no. 2, pp. 206–210, Feb. 1975.
[17] D. M. Miller and R. Drechsler, "On the construction of multiple-valued decision diagrams," in *Proc. 32nd ISMVL*, Boston, MA, 2002, pp. 245–253.
[18] D. Y. Grigoriev and M. Karpinski, "Fast parallel algorithms for sparse multivariate polynomials over finite fields," *SIAM J. Comput.*, vol. 19, no. 6, pp. 1059–1063, Dec. 1990.
[19] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Sequential circuit design using synthesis and optimization," in *Proc. Int. Conf. Comput. Des.*, Oct. 1992, pp. 328–333.
[20] G. DeMicheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.
[21] H. Wu and M. A. Hasan, "Efficient exponentiation of a primitive root in GF(2^m)," *IEEE Trans. Comput.*, vol. 46, no. 2, pp. 162–172, Feb. 1997.
[22] J. C. Jeon and K. Y. Yoo, "Low power exponent architecture in finite field," *Proc. Inst. Electr. Eng.*, vol. 152, no. 6, pt. E, pp. 573–578, Dec. 2005.
[23] J. L. Imaña, J. M. Sánches, and F. Tirado, "Bit parallel finite field multipliers for irreducible trinomials," *IEEE Trans. Comput.*, vol. 55, no. 5, pp. 520–533, May 2006.
[24] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. Piscataway, NJ: IEEE Press, 1994.
[25] M. Ben-Or and P. Tiwari, "A deterministic algorithm for sparse multivariate polynomial interpolation," in *Proc. 20th Symp. Theory Comput.*, Apr. 1988, pp. 301–309.
[26] M. Clausen, A. Dress, J. Grebmeier, and M. Karpinski, "On zero-testing and interpolation of k-sparse polynomials over finite fields," *Theor. Comp. Sci.*, vol. 84, no. 2, pp. 151–164, Jan. 1991.
[27] V. Miller, "Uses of elliptic curves in cryptography," in *Proc. Advances Cryptology—CRYPTO*, H. Williams, Ed., 1986, pp. 417–426.
[28] M. J. Ciesielski, P. Kalla, Z. Zeng, and B. Rouzeyere, "Taylor expansion diagrams: A compact, canonical representation with applications to symbolic verification," in *Proc. Des. Autom. Test Eur.*, Mar. 2002, pp. 285–289.
[29] N. Iliev, J. E. Stine, and N. Jachimiec, "Parallel programmable finite field GF(2^m) multipliers," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI Emerging Trends (ISVLSI)*, Feb. 2004, pp. 299–302.
[30] D. K. Pradhan, "A theory of galois switching functions," *IEEE Trans. Comput.*, vol. C-27, no. 3, pp. 239–249, Mar. 1978.
[31] R. S. Stankoviç and R. Dreschler, "Circuit design from Kronecker Galois field decision diagrams for multiple-valued functions," in *Proc. ISMVL-27*, May 1997, pp. 275–280.
[32] W. Stallings, *Cryptography and Network Security*. Englwood Cliffs, NJ: Prentice-Hall, 1999.
[33] T. Sasao and F. Izuhara, "Exact minimization of FPRMs using multi-terminal EXOR TDDs," in *Representations of Discrete Functions*, T. Sasao and M. Fujita, Eds. Norwell, MA: Kluwer, 1996, pp. 191–210.
[34] T. Sasao and M. Fujita, *Representations of Discrete Functions*. Norwell, MA: Kluwer, 1996.
[35] U. Kebschull and W. Rosenstiel, "Efficient graph-based computation and manipulation of functional decision diagrams," in *Proc. Eur. Des. Autom. Conf.*, Feb. 1993, pp. 278–283.
[36] S. B. Wicker, *Error Control Systems for Digital Communication and Storage*. Englewood Cliffs, NJ: Prentice-Hall, 1995.
[37] Z. Zilic and Z. Vranesic, "A multiple-valued reed-Müller transform for incompletely specified functions," *IEEE Trans. Comput.*, vol. 44, no. 8, pp. 1012–1020, Aug. 1994.
[38] Z. Zilic and Z. G. Vranesic, "A deterministic multivariate interpolation algorithm for small finite fields," *IEEE Trans. Comput.*, vol. 51, no. 9, pp. 1100–1105, Sep. 2002.

Abusaleh M. Jabir (M'06) received the B.Sc. degree (with honors) in applied physics and electronics and the M.Sc. degree (with distinction) in computer science from the University of Dhaka, Dhaka, Bangladesh, and the D.Phil. degree in computing from the Computing Laboratory, University of Oxford, Oxford, U.K., in 2001.

He was with the Hardware Compilation Group, a subdivision of the renowned Programming Research Group, at the University of Oxford. He is currently a Senior Lecturer with the School of Technology, Oxford Brookes University, Oxford. Prior to that, he served as a Lecturer with the Department of Computer Science, University of Dhaka. While completing his D.Phil. degree, he worked with Celoxica Ltd., Oxford, as a Senior Member of the research staff. His research interests include computer architectures, digital systems design, particularly for low-power applications, tests, and verification, efficient hardware design for error control and reliability, and cryptosystems.

Dr. Jabir was 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."

Dhiraj K. Pradhan (S'70–M'72–SM'80–F'88) received the Ph.D. degree in electrical engineering from the University of Iowa, in 1972.

He is currently a Professor of computer science with the Department of Computer Science, University of Bristol, Bristol, U.K. Prior to that, he was a Professor of electrical and computer engineering at Oregon State University, Corvallis. He held the COE Endowed Chair Professorship in Computer Science at Texas A&M University, College Station, where he was also the Founder of the Laboratory of Computer Systems. He also held a Professorship at the University of Massachusetts, Amherst, where he also served as Coordinator of Computer Engineering. He was also with the University of California, Berkeley, Oakland University, Rochester, MI, and the University of Regina, Regina, SK, Canada. He was a Visiting Professor with Stanford University, Stanford, CA. In the past, he was a Staff Engineer with IBM; more recently, he served as the founding CEO of Reliable Computer Technology, Inc. He is the holder 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 computer-aided design and test, as well as to fault-tolerant computing, computer architecture, and parallel processing research, with major publications in journals and conference proceedings, spanning more than 30 years. During this long career, he has been well funded by various agencies in Canada, the U.S., and the U.K. He is the coauthor and editor of various books, including *Fault-Tolerant Computing: Theory and Techniques, Vols. I and II* (Prentice-Hall, 1986), *Fault-Tolerant Computer Systems Design* (Prentice-Hall, 1996, second printing 2003), and *IC Manufacturability: The Art of Process and Design Integration* (IEEE Press, 2000).

Prof. Pradhan is a Fellow of the Association for Computing Machinery and the Japan Society for the Promotion of Science. He continues to serve as an Editor for prestigious journals, including IEEE TRANSACTIONS. Furthermore, he has served as the General Chair and Program Chair for various major conferences. He was also the recipient of a Humboldt Prize, Germany. In 1997, he was also awarded the Fulbright-Flad Chair in Computer Science. He has received Best Paper Award honors, including the 1996 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS Best Paper Award, with W. Kunz.

Jimson Mathew (S'01) received the B.Tech. degree from Mahatma Gandhi University, Kerala, India, and the M.S. degree from Nanyang Technological University, Singapore. He is currently working toward the Ph.D. degree in the Department of Computer Science, University of Bristol, Bristol, U.K.

His research interests primarily focus on the design and testing of Galois-field-based arithmetic circuits, fault-tolerant computing, and low-power design.