Optimization of Data-Flow Computations Using Canonical TED Representation

Maciej Ciesielski, *Senior Member, IEEE*, Daniel Gomez-Prado, *Student Member, IEEE*, Qian Ren, *Member, IEEE*, Jérémie Guillot, and Emmanuel Boutillon, *Member, IEEE*

***Abstract*—An efficient graph-based method to optimize polynomial expressions in data-flow computations is presented. The method is based on the factorization, common-subexpression elimination, and decomposition of algebraic expressions performed on a canonical Taylor expansion diagram representation. It targets the minimization of the latency and hardware cost of arithmetic operators in the scheduled implementation. The generated data-flow graphs are better suited for high-level synthesis than those extracted directly from the initial specification or obtained with traditional algebraic decomposition methods. Experimental results show that the resulting implementations are characterized by better performance and smaller datapath area than those obtained using traditional algebraic decomposition techniques. The described method is generic, applicable to arbitrary algebraic expressions, and does not require any knowledge of the application domain.**

***Index Terms*—Algebraic optimizations, common-subexpression elimination (CSE), data-flow graphs (DFGs), high-level synthesis, Taylor expansion diagrams (TEDs).**

I. Introduction

MANY computations encountered in high-level design specifications are represented as polynomial expressions. They are used in computer graphics designs and digital signal processing (DSP) applications, such as digital filters and DSP transforms, where designs are specified as algorithms written in C/C++. To deal with such abstract descriptions, designers need efficient optimization tools to optimize the initial specification code, prior to architectural (high-level) synthesis. Unfortunately, conventional compilers do not provide sufficient support for this task. Code optimization, such as factorization, common-subexpression elimination (CSE), dead-code elimination, etc., performed by compilers, is done only on the syntactic and lexical levels and is not intended for arithmetic minimization. On the other hand, synthesis techniques, such as scheduling, resource allocation, and binding, employed by high-level synthesis tools, do not address front-end algorithmic optimization [1]. These tools rely on a representation that is derived by a direct translation of the original design specification, leaving a possible modification of that specification to the designer. As a result, the scope of the ensuing architectural optimization is seriously reduced.

This paper introduces a systematic method to perform an optimization of the initial design specification using a canonical graph-based representation called Taylor expansion diagram (TED) [2]. TEDs have already been applied to functional verification and algebraic optimization, such as factorization and CSE, used in front-end synthesis and compilation. However, their scope in this area has been limited to the simplification of linear expressions, such as linear DSP transforms, without considering final scheduled implementations [3], [4].

This paper describes how a canonical TED representation can be extended to handle the optimization of arbitrary *nonlinear* polynomial expressions, using novel factorization and decomposition algorithms. The goal is to generate an optimized data-flow graph (DFG), better suited for high-level synthesis, which will produce the best hardware implementation in terms of its latency and hardware cost. The optimization involves the minimization of the latency and of the hardware cost of arithmetic operations in the final scheduled implementations and not just the minimization of the number of arithmetic operations, as done in all previous works. Expressions with constant multiplications are replaced by shifters and adders to further minimize the hardware cost. The proposed method have been implemented in a software tool, called TDS (for TED-based decomposition system), which is available online [5].

Experimental results show that the DFGs generated from the optimized expressions have smaller latency than those obtained using traditional algebraic techniques; they also require, on average, less area than those provided by currently available methods and tools.

The remainder of this paper is organized as follows. Section II analyzes the state of the art in this field. Section III reviews the TED fundamentals, including a method to represent nonlinear polynomials as linear TEDs. Section IV describes TED decomposition algorithms and introduces the concept of normal factored form (NFF). The generation and optimization of DFG is presented in Section V, along with the analysis of optimization metrics. Section VI describes the generation of DFGs with constant multiplications replaced by shifters and

Manuscript received October 8, 2008; revised February 18, 2009. This paper was recommended by Associate Editor R. Camposano.

M. Ciesielski is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003 USA (e-mail: ciesiel@ecs.umass.edu).

D. Gomez-Prado is with the VLSI CAD Laboratory, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003 USA.

Q. Ren is with Synopsys, Inc., Mountain View, CA 94043 USA.

J. Guillot is with the Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier, 34392 Montpellier, France.

E. Boutillon is with the Laboratoire des Sciences et Techniques de l'Information de la Communication et de la Connaissance, Université de Bretagne Sud, BP 92116 Lorient, France.

Digital Object Identifier 10.1109/TCAD.2009.2024708

adders. Section VII describes the overall TDS system. Finally, Section VIII presents the major results, and Section IX offers the conclusions and perspectives.

II. Previous Work

Research in the optimization of the initial design specifications for hardware designs falls in several categories.

HDL Compilers: Several attempts have been made to provide optimizing transformations in high-level synthesis, hardware description language (HDL) compilers [6]–[9], and logic synthesis [10]. Behavioral transformations have been also used in optimizing compilers [11]. These methods rely on the application of basic algebraic properties, such as associativity, commutativity, and distributivity, to manipulate the algebraic expressions. In general, they do not offer a systematic way to optimize the initial design specification or to derive optimum DFGs for high-level synthesis. While several high-level synthesis systems, such as Cyber [12], [13], Spark [14], or Catapult C [15], [16], apply a host of code optimization methods (kernel-based algebraic factorization, branch balancing, speculative code motion, dead-code elimination, etc.), they do not rely on any canonical representation that would guarantee even the local optimality of the transformations.

Symbolic Algebra Methods: Polynomial models of high-level design specifications have been used in the context of behavioral synthesis for the purpose of component mapping. The approach presented in [17] (*SymSyn* software) uses symbolic polynomial manipulation techniques to automate the mapping of datapaths onto complex arithmetic blocks. The basic blocks are converted to their polynomial representations and matched, using symbolic algebra tools (Maple [18] or Mathematica [19]), against predefined library elements, while minimizing the cost of the components used. However, the decomposition is guided by a selection of explicitly specified *side relations*, which are polynomial expressions that describe the functionality of the available library components. While this works well for the mapping of a fixed data-flow computation onto a given hardware library, it does not address the problem of modifying the initial data-flow specification to obtain the best hardware implementation.

Commercial symbolic algebra tools, such as Maple [18] and Mathematica [19], use advanced symbolic algebra methods to perform an efficient manipulation of mathematical expressions, including fast multiplication, factorization, etc. However, despite the unquestionable effectiveness of these methods for classical mathematical applications, they are less effective in the modeling of large-scale digital circuits and systems.

Domain-Specific Systems: Several systems have been developed for domain-specific applications, such as discrete signal transforms. One such system, FFTW [20], targets code generation for DFT-based computation. The most advanced system for DSP code generation, SPIRAL [21], generates an optimized implementation of linear signal processing transforms, such as discrete Fourier transform (DFT), discrete cosine transform (DCT), discrete Hartley transform (DHT), etc. These signal transforms are characterized by a highly structured form of the transform, with known efficient factorizations such as radix-2 decomposition. SPIRAL uses these properties to obtain solutions in a concise form and applies dynamic programming to find the best implementation. However, these systems are domain specific, and their optimizations rely on the specific knowledge of the transforms.

Kernel Based Decomposition: Algebraic methods have been used in logic optimization to reduce the number of literals in Boolean logic expressions [22]. These methods perform factorization and CSE by applying techniques of kernel extraction [23]. Kernel-based decomposition (KBD), originally employed by logic synthesis, has been recently adopted to optimize polynomial expressions of linear DSP transforms and nonlinear filters [24]. While this method provides a systematic approach to polynomial optimization, the polynomial representation used in this paper is not canonical, which seriously reduces the scope of optimization.

Cut-Based Decomposition: Askar *et al.* [3] proposed an algebraic decomposition method using TED as an underlying data structure. The method is based on applying a series of additive and multiplicative *cuts* to the graph edges, such that their removal separates the graph into disjoint subgraphs. Each additive (multiplicative) cut introduces an addition (multiplication) in the resulting DFG. An admissible cut sequence that produces a DFG with a desired characteristic (minimum number of resources or minimum latency) is sought. The disadvantage of the cut-based method is that it is applicable only to TED graphs with a disjoint decomposition property. Many TEDs, however, such as those shown in Figs. 3 and 4, do not have a disjoint decomposition property and, thus, cannot be decomposed using this method.

In this paper, we show how TEDs can be extended to optimize nonlinear polynomials and how to efficiently generate DFGs that are better suited for high-level synthesis.

III. Polynomial Representation Using TED

A. TED Formalism

TED is a compact word-level graph-based data structure that provides an efficient way to represent computation in a canonical factored form [2]. It is particularly suitable for computation-intensive applications, such as signal and image processing, computer graphics, etc., with computations modeled as polynomial expressions.

An algebraic multivariate expression $f(x, y, \ldots)$ can be represented using Taylor series expansion with respect to a variable x around the origin $x = 0$ as follows:

$$f(x, y, \ldots) = f(0, y, z, \ldots) + xf'(0, y, z, \ldots) + \frac{1}{2}x^2 f''(0, y, z, \ldots) + \ldots \quad (1)$$

where $f'(x = 0)$, $f''(x = 0)$, etc., are the successive derivatives of f with respect to x evaluated at $x = 0$. For a large class of expressions typically encountered in designs specified at a high level, the expansion is finite. The individual terms of the expression are then decomposed iteratively with respect to the remaining variables $(y, z, \ldots)$ one variable at a time.

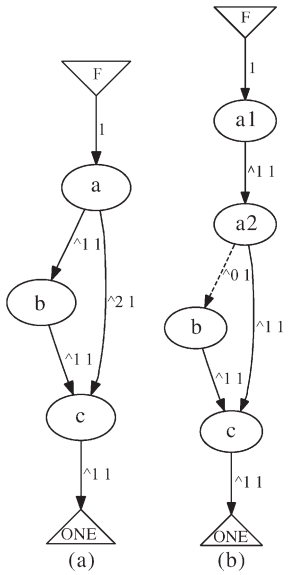

Fig. 1. TED representation for $F = a^2c + abc$: (a) Original nonlinear TED. (b) Linearized TED representing factored form $F = a(a + b)c$.

The resulting decomposition is stored as a directed acyclic graph called TED. The TED nodes represent the individual terms of the expansion, and each TED node is labeled with the name of the decomposing variable. Each edge is labeled with a pair $(^\wedge p, w)$, where $^\wedge p$ represents the power of the variable and w represents the edge weight. The resulting graph can be reduced and normalized in much the same way as binary decision diagrams (BDDs) [25] or binary moment diagrams (BMDs) [26]. The reduced normalized TED is canonical for a fixed order of variables. The expression encoded in the graph is computed as a sum of expressions of all paths in the graph, from the root to terminal 1. A detailed description of this representation, including its construction, reduction, and normalization, can be found in [2].

An example of a TED is shown in Fig. 1(a) for an expression $F = a^2c + abc$. The two terms of the expression, namely $a^2 \cdot c$ and $a \cdot b \cdot c$, can be traced as paths from the root to terminal 1 (ONE). The label ($^\wedge$2, 1) on the edge from nodes a to c denotes quadratic term a^2 with $weight = 1$. The remaining edges are linear, each labeled with ($^\wedge$1, 1).

B. TED Linearization

It has been shown that the TED structure allows for efficient factorization and decomposition of expressions modeled as linear multivariate polynomials [3], [4]. For example, a TED for expression $F = ab + ac$ for variable order (a, b, c) naturally represents the polynomial in its factored form $a(b + c)$. Unfortunately, this efficiency is missing when considering optimizations involving nonlinear expressions. For example, in the TED for function $F = a^2c + abc$ in Fig. 1(a), node a should be factored out, resulting in a more compact form $F = a(a + b)c$, but in its original form, the TED in Fig. 1(a) does not allow for such a factorization.

Fortunately, TED can be readily transformed into a linear form, which supports factorization. Conceptually, a linearized

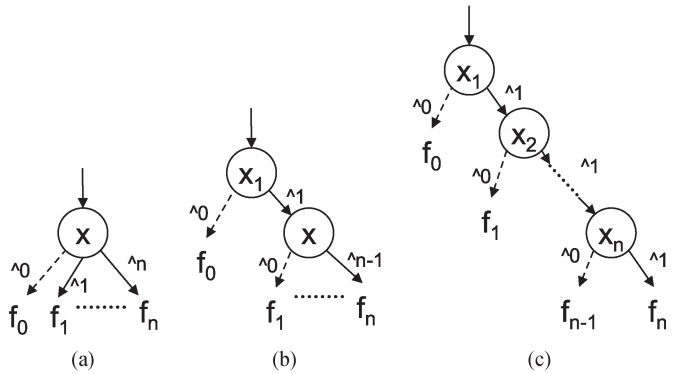

Fig. 2. TED linearization: (a) Original TED containing node x^n. (b) TED after first linearization step. (c) Linearized TED.

TED represents an expression in which each variable x^k, for $k > 1$, is transformed into a product $x^k = x_1 \cdot x_2, \ldots, x_k$, where $x_i = x_j, \forall i, j$.

Consider a nonlinear expression in (2). By replacing each occurrence of x^k by $x_1 \cdot x_2, \ldots, x_k$, this expression can be transformed into a linear form, shown in (3), known as Horner form. A characteristic feature of this form is that it contains a minimum number of multiplications [27] and, hence, is suitable for implementations with a minimum amount of hardware resources

$$F(x) = f_0 + x \cdot f_1 + x^2 \cdot f_2 \cdots + x^n f_n \tag{2}$$

$$= f_0 + x_1 \left(f_1 + x_2(\cdot f_2 \cdots + x_n \cdot f_n)\right). \tag{3}$$

By applying this rule, function $F = a^2c + abc$ can be viewed as $F = a_1a_2c + a_1bc$, which reduces to $F = a_1(a_2 + b)c$ or, equivalently, to $F = a(a + b)c$, as shown in Fig. 1(b).

TED linearization can be performed systematically by iteratively splitting the high-order TED nodes until each node represents a variable in degree 1 and has two children: one associated with a multiplicative (solid) edge and the other with an additive (dotted) edge. This process is shown in Fig. 2. It can be shown that the resulting linear TED is also canonical. In the remainder of this paper, we only consider linear TEDs and linearize the original expressions whenever necessary.

Although TED linearization has been known since the early stages of TED development, it has been used for purposes other than functional optimization. For example, a BTD [28] was proposed as a means to improve the efficiency of the internal TED data structure. Other noncanonical TED-like forms have been used for the purpose of functional test generation for register transfer level (RTL) designs [29].

It should be pointed out that, despite the apparent similarity between linear TEDs and BDDs (or BMDs), the three representations are different. TED represents integer functions of integer inputs, BDDs represent Boolean functions of Boolean inputs, and BMDs are integer functions of binary inputs. In particular, a TED for the expression $ab + a$ reduces to $a(b + 1)$, while a BDD for $ab + a$ reduces to a. A BMD for the same function will be the same as TED only if the inputs a and b are single-bit variables; otherwise, each input must be represented in terms of its component bits.

IV. TED Decomposition

This section reviews the basic concepts and algorithms of TED-based factorization, CSE, and decomposition of polynomial expressions, jointly referred to as *TED decomposition*.

A principal goal of algebraic factorization and decomposition is to minimize the number of arithmetic operations (additions and multiplications) in the expression. An example of *factorization* is the transformation of the expression $F = ac + bc$ into a factored form $F = (a + b)c$, which reduces the number of multiplications from two to one. If a subexpression appears more than once in the expression, it can be extracted and replaced by a new variable. This process is known as *CSE* and results in a decomposed factored form of an expression. The simplification of an expression (or multiple expressions) by means of factorization and CSE is commonly referred to as *algebraic decomposition*.

In this paper, we show how to perform algebraic decomposition directly on a TED graph, taking advantage of its compact canonical representation. In fact, TED already encodes a given expression in factored form. The goal of TED decomposition is to find a factored form encoded in the TED that will produce a DFG with the minimum hardware cost of the final scheduled implementation. This objective is different than a straightforward minimization of the number of operations in the unscheduled DFG, which has been the subject of all the known previous works [3], [4], [24].

The TED decomposition method described here extends the original work of Askar *et al.* [3] in that it is also based on the TED data structure. However, it differs from the cut-based decomposition in the way the algebraic operations are identified in the TED and extracted from the graph to generate the DFG. Instead of the top–down cut-based decomposition of [3], the TED decomposition scheme presented here is applied in a bottom–up fashion. Furthermore, it applies to arbitrary TEDs (linearized if necessary), including those that do not have a disjoint decomposition property. The method is based on a series of functional transformations that decompose the TED into a set of irreducible hierarchical TEDs from which a final DFG representation is constructed. The decomposition is guided by the quality of the resulting DFG and not just by the number of operators.

TED decomposition is composed of the following basic steps:

1) TED construction and linearization;
2) variable ordering;
3) sum- and product-term extraction;
4) recursive TED decomposition;
5) final decomposition of the top-level TED;
6) DFG construction and optimization.

The first step, which is the construction and linearization of the TED, has already been described in Section III. The remaining steps are described next. In the following, all the procedures are applicable to linear TEDs, where each node has at most two edges of different types, *multiplicative* and *additive*.

A. Variable Ordering

The structure of the TED, and, hence, the resulting decomposition, strongly depends on the variable order. In fact, as shown in Section IV-E, the decomposition is uniquely determined by the TED structure and the order of its variables. Hence, TED variable ordering plays a central role in deriving decompositions that will lead to efficient hardware implementations.

Several variable ordering algorithms have been developed, including static ordering and dynamic reordering schemes [30], [31], similar to those for BDDs. However, the significant difference between variable ordering for BDDs and for TEDs is that ordering for linearized TEDs is driven by the complexity of the resulting normal factored form (NFF) and the structure of the resulting DFGs, rather than by the number of nodes in the diagram.

B. Subgraph Extraction and Substitution

Before describing the actual decomposition algorithm, we introduce the *extraction* and *substitution* operation, sub, which forms the basis of most of the decomposition operations. It has been implemented in the TDS system as the sub command. Given an arbitrary subexpression $expr$ of the expression encoded in the TED, the command $sub\ var = expr$ extracts the subexpression $expr$ from the TED and substitutes it with a new variable var.

The sub operation is implemented as follows [4]. First, the variables in the expression $expr$ are pushed to the bottom of the TED, respecting the relative order of variables in $expr$. Let the topmost variable in $expr$ be v (i.e., if this expression were represented by a separate TED, node v would be its topmost node). Assuming that $expr$ is contained in the original TED, this expression will appear in the reordered TED as a subgraph rooted at node v. (There may be other nodes with variable v, but because of the canonicity of the TED, there will be only one such subgraph, pointed to by at least one reference edge.) At this point, the extraction of $expr$ is accomplished by removing the subgraph rooted at v and connecting the reference edge(s) to terminal node 1. Depending on the overall decomposition strategy (static, dynamic, etc.), the graph can be reordered again to its original order, with the newly created variable var placed directly above the original position of v (recall that variable v may still be present in other parts of the graph).

The sub procedure can extract arbitrary subexpressions from the graph, regardless of the position of its support variables in the TED. Furthermore, if an internal portion of subexpression $expr$ is used by other portions of the TED, i.e., if any of the internal subgraph nodes is referenced by the TED at nodes different than its top node v, that portion of $expr$ is automatically duplicated before extraction and variable substitution. Those operations are part of the standard TED manipulation package. This case is shown in Fig. 3 for a TED of function $F = (a + b) \cdot (c + d) + d$. The sum term $(c + d)$ is extracted from the graph and replaced by a new variable S_1, leaving the subgraph rooted at node d (referenced by node b) in the TED.

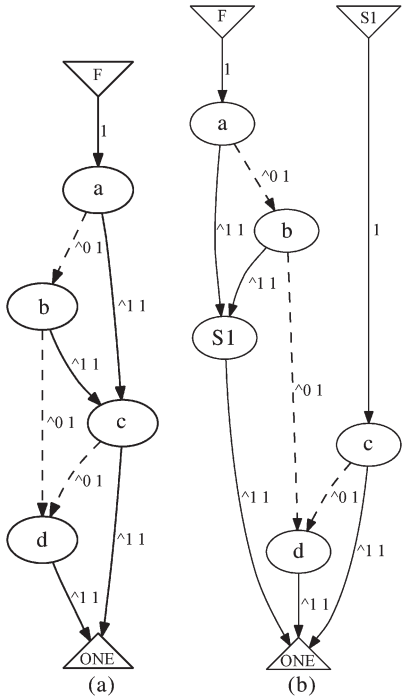

Fig. 3. Extraction of sum term with term duplication: (a) Original function for $F = (a + b) \cdot (c + d) + d$. (b) Extracting $S_1 = c + d$ with duplicated node d.

C. Recursive TED Decomposition

TED decomposition is performed in a bottom–up manner, by extracting simpler terms and replacing them with new variables (nodes), followed by a similar decomposition of the resulting higher level graph(s). The final result of the decomposition is a series of irreducible TEDs related hierarchically.

The decomposition algorithms will be illustrated using the following expression:

$$F = x \cdot z \cdot u + p \cdot w \cdot r + x \cdot q \cdot r + y \cdot r. \tag{4}$$

In its current form, this expression contains seven multiplications and three additions. We will show how to simplify this expression to obtain a DFG with fewer multiplication operations and smallest latency by performing a decomposition of its TED. Fig. 4(a) shows the TED for this expression for a particular variable order.

Note that the TED in Fig. 4(a) cannot be decomposed with the cut-based approach: It does not have additive cuts that would separate the graph disjunctively into disjoint subgraphs; neither does it have a multiplicative cut (dominator nodes) that would decompose it conjunctively into disjoint subgraphs.

1) Product-Term Extraction: The *extractable product term* is defined as a product of variables Πv_i which appear in the expression only once. Such an expression can be extracted from the TED without duplicating any of its variables.

Extractable product terms can be identified in the TED as a set of nodes connected by a series of multiplicative edges only. By definition, the intermediate nodes in the set cannot have any incident incoming or outgoing edges other than the multiplicative edges by which they are connected (i.e., they are referred to only once). Only the starting and ending nodes in the set can have incident additive edges or more than one incoming or outgoing multiplicative edge; the ending node can also be the terminal node 1.

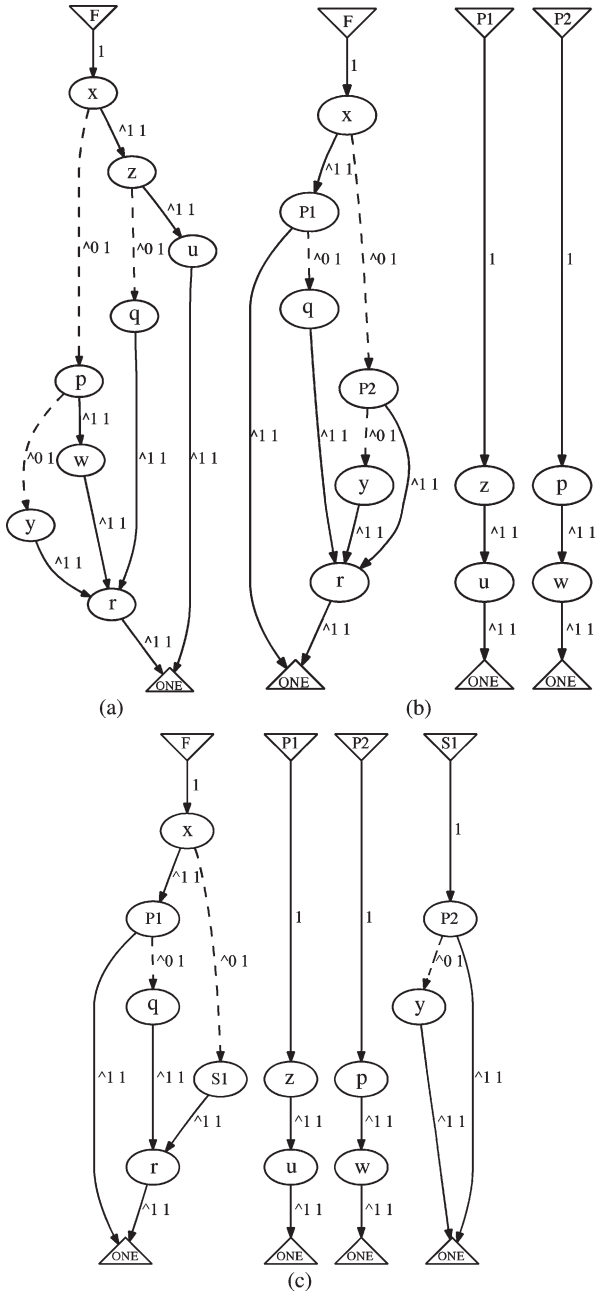

Fig. 4. TED decomposition for expression $x \cdot z \cdot u + p \cdot w \cdot r + x \cdot q \cdot r + y \cdot r$: (a) Original TED. (b) Simplified hierarchical TED after product-term extraction, $P_1 = z \cdot u$ and $P_2 = p \cdot w$. (c) Final hierarchical TED after sum-term extraction, $S_1 = P_2 + y$, and its normal factored form $F = x \cdot (z \cdot u + q \cdot r) + (p \cdot w + y) \cdot r$.

The extractable product terms can be readily identified in the TED by traversing the graph in a bottom–up fashion and creating a list of nodes connected by a simple series of multiplicative edges. Starting with terminal node 1, the procedure examines each node in the TED in a reverse variable order. For each node, it traverses all its in-incident nodes in a depth-first fashion and includes them in the product term if they satisfy the product

term condition. In the TED in Fig. 4(a), the first node visited from terminal 1 is node r. No product term can be constructed with node r, since it has more than one incoming multiplicative edge. The next in-incident node u has only one incident edge; therefore, it produces an extractable product term $z \cdot u$. The list stops at z since it has two outgoing edges. Next, the procedure examines the next node in order, which is node r, and its in-incident edges. This search results in a product term $p \cdot w$. No other terms can be found in this graph. Note that the terms $y \cdot r$, $w \cdot r$, or $q \cdot r$ are not extractable product terms, since their extraction requires duplication of node r.

Each product term extracted from the TED (expression) is replaced by a single node (new variable). The newly introduced variable representing such a term is represented by a simple TED. We refer to such a TED as *irreducible TED*, as it is composed of a single product term that cannot be further reduced.

The TED in Fig. 4(a) has two extractable product terms $z \cdot u$ and $p \cdot w$ corresponding to new variables P_1 and P_2, respectively. They are represented by irreducible TEDs P_1 and P_2, as shown in Fig. 4(b). The left part of the figure shows a reduced TED with the two nodes P_1 and P_2 referring to the irreducible graphs.

2) Sum-Term Extraction: Similar to the extractable product term, the *sum term* is defined as a sum of variables $\sum v_i$ in the expression. A sum term appears in the TED as a set of nodes incident to multiplicative edges joined at a single common node, such that the nodes in question are connected by a chain of additive edges only.

If the graph does not have any sum terms, an application of product-term extraction, described earlier, may expose additional sum terms. As an example, the original TED in Fig. 4(a) does not have any sum terms, while the one obtained from product-term extraction, shown in Fig. 4(b), contains the sum term $S_1 = P_2 + y$.

The sum terms can be readily identified in the TED by traversing the graph in a bottom–up fashion and creating, for each node v, a list of nodes reachable from v by a multiplicative edge and verifying if they are connected by a chain of additive edges. The procedure starts at terminal node 1 and traverses all the nodes in the graph in a reversed variable order. In the example in Fig. 4(b), the set of nodes reachable from terminal node 1 is $\{P_1, r\}$. Since these nodes are not linked by an additive edge, they do not form a sum term in the expression. Node r is examined next. The list of nodes reachable from node r by multiplicative edges is $\{q, y, P_2\}$, with $\{P_2, y\}$ linked by an additive edge. Hence, they form a sum term $(P_2 + y)$. Such a term is substituted by a new variable S_1 and represented as an irreducible TED. No other sum term can be extracted from the simplified TED. The resulting hierarchical TED is shown in Fig. 4(c).

In the aforementioned example, the sum-term variables were adjacent in the TED and directly connected by an additive edge. The method actually works for an arbitrary variable ordering, i.e., even if the sum-term nodes are separated by other variables in the chain of additive edges. This case is shown in Fig. 5(a) for expression $F = a \cdot m + b \cdot n + c \cdot m + d \cdot n$. First, consider the sum-term variables $\{b, d\}$ common to node

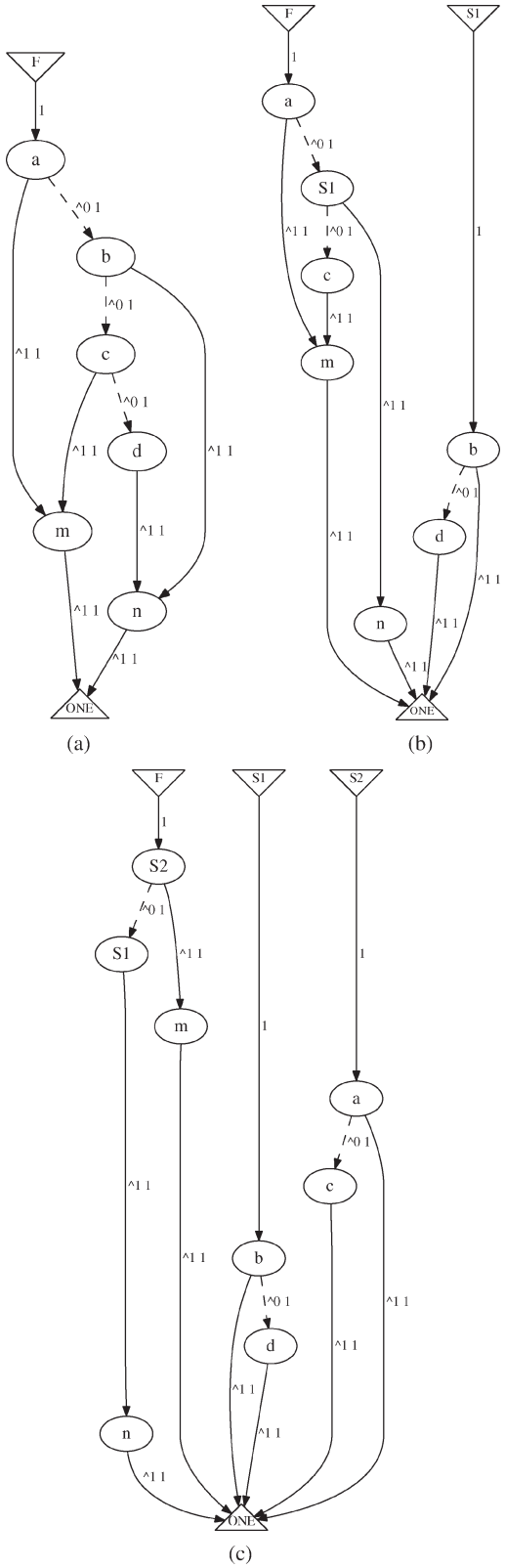

Fig. 5. TED for $F = a \cdot m + b \cdot n + c \cdot m + d \cdot n$: (a) Original TED. (b) TED after extracting sum term $(b + d)$. (c) TED after extracting sum term $(a + c)$.

n. These nodes are linked by a series of additive edges that include variable c, not connected to n. Yet, the sum term $(b + d)$ can be readily extracted from the graph using the sub operation

described in Section IV-B. This is because of the associativity property of addition

$$a \cdot m + b \cdot n + c \cdot m + d \cdot n = a \cdot m + c \cdot m + b \cdot n + d \cdot n. \tag{5}$$

The resulting TED is shown in Fig. 5(b), with $S_1 = (b + d)$ extracted from the expression. Note the effect of this extraction on the TED, which has been reordered.

The same procedure applies to the sum term $(a + c)$ associated with node m, resulting in the final factorization

$$a \cdot m + b \cdot n + c \cdot m + d \cdot n = (a + c)m + (b + d)n \tag{6}$$

as shown in Fig. 5(c).

This example illustrates the power of the extraction and substitute procedure, which is capable of identifying such terms without explicitly reordering the variables and implicitly applying the associativity property using simple graph transformations.

3) Final Decomposition: The product-term and sum-term extraction procedures are repeated iteratively until the top-level TED is reduced to an irreducible form. At this point, the top-level graph is decomposed using the fundamental Taylor decomposition principle, described in Section III.

The graph is traversed in a topological order, starting at the root node. At each visited node v, associated with variable x, the expression $F(v)$ is computed as

$$F(v) = x \cdot F_1(v) + F_0(v) \tag{7}$$

where $F_0(v)$ is the function rooted at the first node reachable from v by an additive edge and $F_1(v)$ is the function rooted at the first node reachable from v by a multiplicative edge.

To illustrate this process, consider the top-level TED shown in the left part of Fig. 4(c). The following expression is derived for this graph

$$F = x \cdot (P_1 + q \cdot r) + S_1 \cdot r \tag{8}$$

where $P_1 = z \cdot u$, $P_2 = p \cdot w$, and $S_1 = (P_2 + y)$ are the irreducible component TEDs obtained by the decomposition.

Such an expression is then transformed into a structural representation, which is a DFG, where each algebraic operation (multiplication or addition) is represented as a hardware operator (multiplier or adder). The relationship between the expression derived from the TED decomposition and the resulting DFG is analyzed in Section IV-E.

D. Dynamic Factorization

An alternative approach to TED factorization has been proposed in [4]. This method relies on the observation that a node with multiple incoming (reference) edges represents a common subexpression that can be extracted from several places in the graph and substituted with a new variable (the sub routine, described earlier, can be used for this purpose). In addition, constants are represented as special TED nodes, rather than as labels on the graph edges, and placed in the TED above all variables. This makes it possible to factor out constants as well as algebraic expressions. The TED variables are then rotated down in order to find the best candidate subexpression for extraction. Note that this procedure dynamically changes the variable order and hence modifies the initial TED; this is in contrast to static factorization methods that keep the variable order unchanged (except for the local sub operation). This approach may result in further minimization of operators. Further details of this approach are provided in [4].

E. Normal Factored Form

The recursive TED decomposition procedure described in the previous section produces a simplified algebraic expression in factored form. By imposing additional rules regarding the ordering of variables in the expression, such a form can be made unique and minimal. We refer to such a form as normal factor form (NFF).

Definition 1: The factored form expression associated with a given TED is called an NFF for that TED if the order of variables in the factored form expression is compatible with the order of variables in the TED.

To illustrate the concept of NFF, consider again the TED in Fig. 4(c) and the associated NFF, $x \cdot (P_1 + q \cdot r) + S_1 \cdot r$, resulting from the TED decomposition, given in (8). The variables in this equation do appear in the order compatible with the top-level TED. In particular, the term $x \cdot (P_1 + q \cdot r)$ appears first, since x is the top-level variable in the TED, followed by $S_1 \cdot r$, which is placed lower in the graph. Similarly, the order of variables in the term $(P_1 + q \cdot r)$ is compatible with that in the TED and so is the order of variables in each subexpression P_1, P_2, and S_1 associated with the corresponding irreducible TED (which, in turn, is compatible with the original TED). In general, the term with the highest variable in the TED is printed first in the NFF.

One should note that there is a one-to-one mapping between the arithmetic operations $(+, \cdot)$ in NFF and the corresponding edges in the decomposed TED. Specifically, each addition operation corresponds to an additive edge, and each multiplication corresponds to a multiplicative edge in an irreducible TED obtained from TED decomposition. To illustrate this point, refer to the set of irreducible TEDs in Fig. 4(c) and compare it to the DFG generated from this decomposition, shown in Fig. 6. The five multiplication operations in the DFG correspond to the three nontrivial multiplicative edges in the top TED graph ($x \cdot P_1$, $q \cdot r$, and $S_1 \cdot r$) and two nontrivial multiplicative edges in the subgraphs $P_1 = z \cdot u$ and $P_2 = p \cdot w$. Similarly, there are three additions corresponding to the three additive edges in these graphs: two in the top-level TED $F = x \cdot (P_1 + qr) + S_1 r$ and one in the TED for $S_1 = P_2 + y$.

An important feature of the NFF is that it is unique for a TED with a fixed variable order, as expressed by the following theorem.

Theorem 1: The NFF derived from a linear TED is unique.

Proof: By construction, NFF is a recursively defined sum of products or a product of sums of algebraic expressions. At the lowest decomposition level, a product term (sum term) is a product of variables Πv_i (sum of variables $\sum v_i$), where each variable v_i appears only once and the variables are ordered

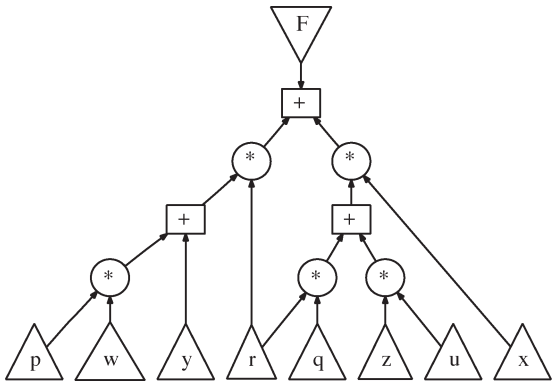

Fig. 6. DFG generated from the decomposed TED and the associated NFF: $F = x \cdot (z \cdot u + q \cdot r) + (p \cdot w + y) \cdot r$ shown in Fig. 4(c).

according to their order in the original TED. Hence, the expression describing such a term is unique. A subsequent extraction of sum or product terms on the new TED creates new variables corresponding to new subexpressions. By canonicity, each such variable is unique, with a unique position in the current TED; therefore, the corresponding expression is also unique. Hence, at any point of the decomposition, each subexpression is unique, and the final NFF expression is unique. ■

Note that the resulting NFF of the decomposed TED depends only on the structure of the initial TED, which in turn depends on the ordering of its variables.

V. DFG Generation and Optimization

Once a TED has been decomposed and the corresponding NFF produced, a structural DFG representation of the expression is constructed in a straightforward fashion.

Each irreducible TED is first transformed into a simple DFG using the basic property of the NFF: Each additive edge in the TED maps into an addition operation, and each multiplicative edge maps into a multiplication operation in the resulting DFG. Each node of the DFG has exactly two children, representing two operands associated with the operation. Operations with multiple operands are broken into a chain of two-operand operations or into a logarithmic tree to minimize latency. During the construction, each node of the DFG is stored in the hash table, keyed by the corresponding TED function. If, at any point of the DFG construction, the expression corresponding to a DFG node is present in the table, it is reused. For example, when constructing a node for $f = a \cdot b \cdot c$, a node for subexpression $z = a \cdot b$ is constructed first, followed by the construction of $f = z \cdot c$ (the processing of nodes follows the order of variables in the TED). If a DFG node associated with subexpression $z = a \cdot b$ has already been constructed, the node for $f = z \cdot c$ is reused in this operation. This removes potential redundancy from the DFG that has not been captured by factorization (caused by potentially poor variable order). Therefore, if a good variable order has not been found, further DFG optimization can be achieved directly on the DFG.

All the DFGs are then composed together to form the final DFG. The DFG obtained from the decomposition of the TED in Fig. 4(c) is shown in Fig. 6.

It should be noted that, unlike NFF, the DFG representation is not unique. While the number of operations remains fixed (dictated by the structure of the TED and its variable order), a DFG can be further restructured and balanced to minimize latency. Several methods employed by logic and high-level synthesis can be used for this purpose [10]. In the simplest case, a chain of product or sum terms is replaced by logarithmic trees in each DFGs, and all the component DFGs are composed together to form the final DFG. Since the construction of DFG is computationally inexpensive ($O(n)$ for an expression with n variables), its cost can be evaluated quickly from its NFF. In addition to the latency and the number of operations of the unscheduled DFG, such a cost can include the number of resources, obtained by performing a fast heuristic scheduling. Alternatively, a number of resources can be provided by the user as a constraint.

In summary, two basic mechanisms are used to guide the decomposition to obtain a DFG with the desired property: 1) variable ordering, including implicit reordering during the term extraction and substitution, and 2) DFG restructuring to minimize the expected latency and/or the number of resources.

VI. Replacing Constant Multipliers by Shifters

Multiplications by constants are common in designs involving linear systems, particularly in computation-intensive applications such as DSP. It is well known that multiplications by integers can be implemented efficiently in hardware by converting them into a sequence of shifts and additions/subtractions. Standard techniques are available to perform such a transformation based on canonical signed digit (CSD) representation [32]. TEDs provide a systematic way to transform constant multiplications into shifters, while considering factorization involving shifters. This is done by introducing a special *left shift* variable into a TED representation, while maintaining its canonicity. The modified TED can then be optimized using the decomposition techniques described earlier.

First, each integer constant C is represented in CSD format as $C = \sum_i (k_i \cdot 2^i)$, where $k_i \in (\bar{1}, 0, 1)$. By introducing a new variable L to replace constant 2, constant C can be represented as:

$$C = \sum_i (k_i \cdot 2^i) = \sum_i (k_i \cdot L^i). \tag{9}$$

The term L^i in this expression can be interpreted as a left shift by i bits.

The next step is to generate the TED with the shift variables, linearize it, and perform decomposition. Finally, in the generated DFG, the terms involving the shift variables L^k are replaced by k-bit shifters. Such a replacement minimizes hardware cost of the datapath operators.

The example in Fig. 7 shows this procedure for the expression $F_0 = 7a + 6b$. The original TED for this expression is shown in Fig. 7(a), with the corresponding DFG shown

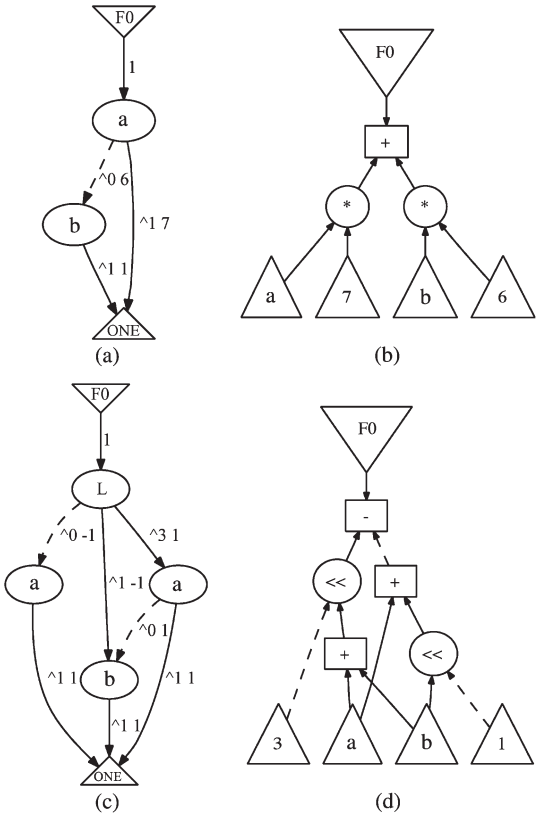

Fig. 7. Replacing constant multiplications by shift and add operations: (a) Original TED for $F_0 = 7a + 6b$. (b) Initial DFG with constant multipliers. (c) TED after introducing *left shift* variable L. (d) Final DFG with shifters, corresponding to the expression $F_0 = ((a + b) \ll 3) - (a + (b \ll 1))$.

in Fig. 7(b). The original polynomial is transformed into an expression with shift variable L

$$F = (L^3 - 1) \cdot a + (L^3 - L^1) \cdot b = L^3 \cdot (a + b) - L \cdot b - a \tag{10}$$

represented by a TED in Fig. 7(c). The TED is then decomposed using techniques described earlier. Finally, all constant multiplications by L^k are replaced by k-bit shifters, resulting in a DFG in Fig. 7(d). The optimized expression corresponding to this DFG is

$$F_0 = ((a + b) \ll 3) - (a + (b \ll 1)) \tag{11}$$

where the symbol "$\ll k$" refers to a left shift by k bits. This implementation requires only three adders/subtractors and two shifters, a considerable gain compared to the two multiplications and one addition of the original expression.

VII. TDS System

The TED decomposition and DFG optimization methods presented in this paper were implemented as part of an experimental software system, called TDS. The system transforms the initial functional design specification into a canonical form (TED) and converts it into a DFG. The generated DFG is optimized for latency and/or resource utilization and used as input to high-level synthesis. The system is intended for data-flow and computation-intensive designs used in DSP applications. It is available online at [5].

The initial design specifications, written in C or behavioral HDL, is translated into a hybrid network composed of functional blocks (TEDs) and structural elements. TEDs are obtained from polynomial expressions describing the functionality of the arithmetic components of the design. The TEDs are then transformed into a structural DFG representation through a series of decomposition steps described in this paper, including TED linearization, factorization, extraction, substitution, and the replacement of constant multipliers by shifters. The "structural" elements include comparators and other operators that cannot be expressed analytically for the purpose of TED construction and, hence, are treated as "black boxes." The entire network (global DFG) is further restructured to minimize the design latency.

The overall TDS system flow is shown in Fig. 8. The left part of the figure shows traditional high-level synthesis flow. It is based on the high-level synthesis tool GAUT [33], which extracts a fixed DFG from the initial specification. The flow on the right is the TDS system that transforms the extracted internal data-flow representation into an optimized DFG, which is then passed back to GAUT for high-level synthesis.

TDS uses a set of interactive commands and optimization scripts. There are two basic groups of commands: 1) those that are used for the construction and manipulation of TEDs and DFGs and 2) those that operate on a hybrid TDS network, with multiple TEDs and structural elements.

The input to the system can be provided in several ways: 1) by reading a $cdfg$ file (using the *read* command), produced by GAUT; 2) by directly typing in the polynomial expression (the *poly* command); or 3) by specifying the type and size of the DSP transform precoded in the system (the *tr* command). The optimized DFG is produced by writing a file (the *write* command) in a $cdfg$ format compatible with GAUT.

Table I lists some of the TDS commands used in the optimization scripts. Most commands have several options.

VIII. Experimental Results

The TED decomposition described in this paper was implemented as part of a prototype system, which is the TDS. The design, written in C, is first compiled by GAUT to produce an initial data-flow netlist in $cdfg$ format, which serves as input to TDS. TDS transforms this netlist into a set of TEDs and performs all the TED- and DFG-related optimizations using the optimization scripts. These scripts include one or more steps of TED ordering, factorization, CSE, replacement of constant multipliers by shifters and adders, and DFG restructuring. The result is written back in the $cdfg$ format and entered into GAUT, which generates the synthesizable VHDL code for final logic synthesis.

The results shown in the tables are reported for the following delay parameters, taken form the *notech* library of GAUT $multiplier\ delay = 18$ ns, $adder/sub\ delay = 8$ ns, $shifter\ delay = 9$ ns, and $clock\ period = 10$ ns. Note that multiplication requires two clock cycles.

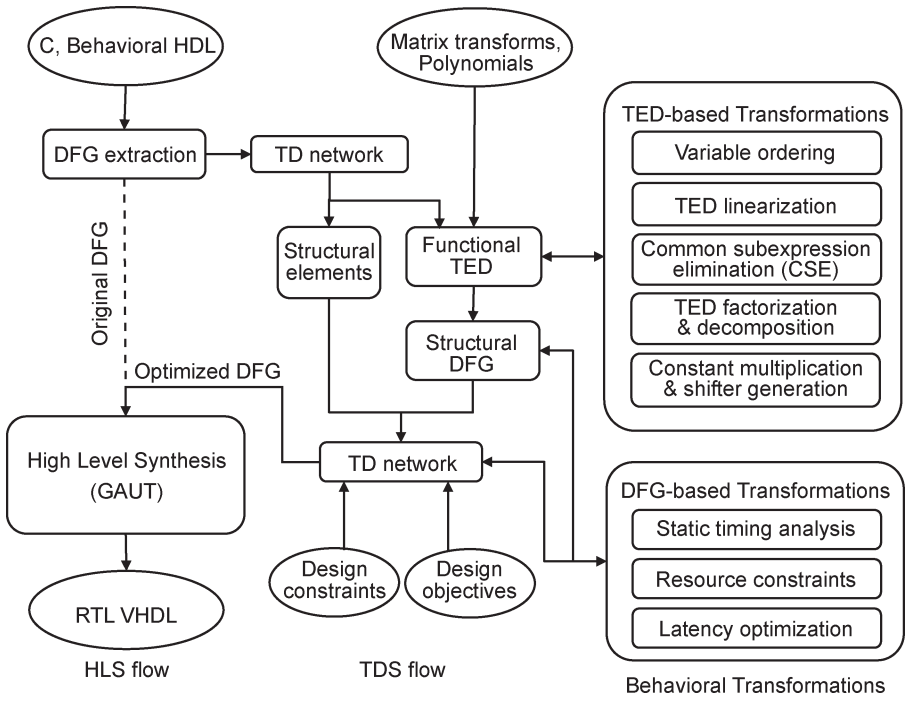

Fig. 8. TDS system flow.

TABLE I
TDS OPTIMIZATION COMMANDS

Command	Description
balance	Balance the DFG (-d) or the Netlist (-n)
bottom	Move variable to the bottom
candidate	List candidate expressions for extraction
dcse	Dynamic common subexpression elimination
decompose	Decompose the TED into NFF
dfg2ted	Transform DFG into TED
dfg2ntl	Construct netlist from DFG
dfactor	Perform dynamic factorization
dfgschedule	Schedule DFG with (-m) MPY (-a) ADD (-s) SUB
extract	Extract an output from TED (-t) or from Netlist (-n)
flatten	Create global TED by flattening the description
linearize	Linearize the TED
ntl2ted	Transform Netlist into set of TEDs
poly	Input the polynomial expression
print	Print out the NFF (-f) or statistics (-s)
read	Read optimization script, cdfg or matrix file
reloc	Relocate variable to a given level
reorder	Reorder TED to minimize operators or nodes
remapshift	Replace constant multipliers by shifters/adders
scse	Static common subexpression elimination
shifter	Replace constants by variables L
show	Show the TED (-t) or the DFG (-d)
sub	Substitute expression by a new variable
ted2dfg	Generate DFG from the TED
top	Move variable to the top
tr	Generate polynomial expressions for DSP transform
vars	Define the order of variables
write	Write output file in $cdfg$ format

Table II compares the implementation of a Savitzky–Golay (SG) filter using the following: 1) the DFG extracted from the original design by GAUT; 2) the DFG produced by the KBD system of [24]; and 3) the DFG produced by TDS. The table reports the following data: The top row, marked *DFG*, shows the number of arithmetic operations in an *unscheduled DFG* generated by each solution. The operations include adders, multipliers, shifters, and subtractors. The remaining rows show the actual number of resources used for a given latency in a *scheduled* DFG, synthesized by GAUT. It also shows the actual implementation area using two synthesis tools: GAUT (which reports datapath area only) and Synopsys DC Compiler (which gives area for datapath, steering, and control logic) in their respective area units. The minimum achievable latency of each method (measured by the scheduled DFG solution) is shown in bold font. The results for circuits that cannot be synthesized for a given latency are marked with "−" (overconstrained).

No CPU time for TED decomposition is given for these experiments as they take only a fraction of a second to complete, and no such data are available from literature for the KBD system [24]. Furthermore, we do not report the circuit delay produced by Synopsys as it is almost identical for all solutions (equal to the delay of a multiplier).

As shown in Table II, the minimum latency for the DFG extracted from the original design, without any modification, is 160 ns. The DFG solutions produced by both KBD and TDS have minimum latency of 120 ns each, a 25% improvement with respect to the original design. However, the TDS implementation requires a smaller area than both the original and the KBD solution, as measured by both synthesis tools. In fact, all entries in the table show a tight correlation between the synthesis results of Synopsys DC and GAUT, which allows us to limit the results of other experiments to those produced by GAUT only.

Table III presents a similar comparison for designs from different domains (filters, digital transforms, and computer graphic applications) synthesized with GAUT. As an example, examine the Quintic Spline design, for which the KBD solution had the smallest number of operations in the unscheduled DFG and the latency of 140 ns. The DFG obtained by TDS

TABLE II
SG FILTER IMPLEMENTATIONS SYNTHESIZED WITH GAUT AND SYNOPSYS DC

Design		Original design		KBD [24] solution		TDS solution	
	Latency (ns)	+,×,≪,−	Area GAUT / SynDC	+,×,≪,−	Area GAUT / SynDC	+,×,≪,−	Area GAUT / SynDC
SG Filter	DFG →	2,16,6,0		4,14,3,0		6,11,3,0	
	L=120	–	–	1,5,2,0	439 / 22,057	1,4,1,0	348 / 20,849
	L=130	–	–	1,5,1,0	431 / 22,057	2,3,1,0	273 / 18,021
	L=140	–	–	1,4,1,0	348 / 19,952	1,3,1,0	265 / 18,160
	L=150	–	–	1,4,1,0	348 / 19,648	1,3,1,0	265 / 17,862
	L=160	1,4,2,0	356 / 20,442	1,3,1,0	265 / 17,428	1,2,1,0	182 / 14,795

TABLE III
COMPARISON OF MINIMUM ACHIEVABLE LATENCY AND AREA FOR DIFFERENT DESIGNS

Design		Original design		KBD [24] solution		TDS solution	
	Latency (ns)	+,×,≪,−	Area	+,×,≪,−	Area	+,×,≪,−	Area
Cosine wavelet	DFG →	9,12,9,14		10,10,4,5		9,10,12,7	
	L=110	–	–	–	–	3,2,4,1	447
	L=120	–	–	2,4,1,1	402	3,3,2,1	364
	L=130	–	–	2,4,1,1	402	2,3,2,1	356
	L=140	–	–	2,3,1,1	319	2,3,2,1	273
	L=150	–	–	1,3,1,1	311	2,2,2,1	273
	L=160	–	–	2,2,1,1	236	1,2,2,1	265
	L=170	–	–	1,2,1,1	228	2,2,1,1	236
	L=180	2,5,1,1	476	1,2,1,1	228	2,2,1,1	236
Chroma	DFG →	8,12,0,2		10,6,7,8		7,13,0,9	
	L=100	–	–	–	–	2,5,0,3	455
	L=110	2,4,0,2	364	2,3,3,2	413	2,4,0,2	364
Chebyshev polys	DFG →	3,15,5,0		7,7,4,1		6,7,10,6	
	L=100	–	–	–	–	2,3,2,1	347
	L=110	–	–	–	–	2,2,2,1	264
	L=120	–	–	1,3,1,1	302	1,2,2,1	256
	L=130	–	–	1,2,1,1	219	1,2,1,2	227
	L=140	–	–	1,2,1,1	219	1,2,1,1	219
	L=150	–	–	1,2,1,1	219	1,2,1,1	219
	L=160	–	–	1,2,1,1	219	1,2,1,1	219
	L=170	1,3,1,0	265	1,1,1,1	136	1,1,1,1	136
Quintic Spline	DFG →	5,28,2,0		5,13,3,0		6,14,4,0	
	L=110	–	–	–	–	1,5,1,0	460
	L=120	–	–	–	–	2,4,2,0	422
	L=130	–	–	–	–	1,4,1,0	377
	L=140	–	–	1,4,1,0	377	1,3,1,0	294
	L=150	–	–	1,3,1,0	294	1,3,1,0	294
	L=160	–	–	1,3,1,0	211	1,3,1,0	294
	L=170	–	–	1,2,1,0	211	1,3,1,0	294
	L=180	1,5,1,0	460	1,2,1,0	211	1,2,1,0	211
Quartic Spline	DFG →	4,21,2,0		5,11,4,0		5,13,4,0	
	L=100	–	–	–	–	2,5,1,0	468
	L=110	–	–	–	–	1,5,1,0	460
	L=120	–	–	–	–	2,4,1,0	385
	L=130	–	–	1,3,1,0	294	1,4,1,0	377
	L=140	–	–	1,3,1,0	294	1,3,1,0	294
	L=150	–	–	1,2,1,0	211	1,3,1,0	294
	L=160	1,5,0,0	423	1,2,1,0	211	1,3,1,0	294
VCI 4x4	DFG →	11,12,0,0		11,0,8,9		9,2,4,6	
	L=70	4,7,0,0	613	–	–	4,2,4,4	406
	L=80	4,6,0,0	530	–	–	4,2,2,2	302
	L=90	3,4,0,0	356	–	–	2,2,2,2	286
	L=100	3,4,0,0	356	2,0,4,2	208	2,1,2,2	203

TABLE IV
RESULTS OF DFG PRODUCED BY TDS VERSUS ORIGINAL AND KBD

Design	TDS vs			
	Original		KBD [24]	
	Latency (%)	Area (%)	Latency (%)	Area (%)
SG Filter	25.00	27.62	0.00	20.73
Cosine	38.88	50.42	8.33	9.45
Chrome	9.09	0.00	9.09	11.86
Chebyshev	41.17	48.68	16.66	15.23
Quintic	38.88	54.13	21.42	22.02
Quartic	37.50	30.50	23.07	-28.23
VCI 4x4	0.00	42.98	30.00	2.40
Average	27.22	36.33	15.51	7.64

produced the implementation with 110 ns, i.e., 21.4% faster, even though it had more operations in the unscheduled DFG. Furthermore, for the minimum latency of 120 ns, obtained by KBD, TDS produces an implementation with an area that is 22% smaller than that of KBD. Similar behavior can be observed in the remaining designs. In all the cases, the latency of the scheduled DFGs produced by TDS is smaller, and, with the exception for the Quartic Spline design, all of them have also smaller hardware areas for the minimum latency achieved by KBD. The reason that Quartic design requires larger area for the reference latency can be attributed to latency-oriented minimization. Note, however, that the reference latency in this case is 130 ns, as achieved by KDB, compared with 100 ns obtained with TDS.

Table IV summarizes the implementation results for these benchmarks. We can see that the implementations obtained from TDS have latencies smaller on average by 15.5% and 27.2% with respect to the KBD and original DFGs, respectively. Moreover, for the reference latency (defined as the minimum latency obtained by the other two methods), the TDS implementations have, on average, 7.6% (with respect to KBD) and 36.3% (with respect to the original) smaller area.

IX. CONCLUSION AND FUTURE WORK

The results demonstrate that the TED-based optimization of polynomial expressions yields DFGs that are better suited for high-level synthesis than those obtained by direct data-flow extraction from the initial specification. Specifically, the generated DFGs have lower latency and/or produce implementations that require smaller hardware area. This is because each step of the TED-based decomposition is evaluated in terms of the DFG, constructed in the background. Such an evaluation is fast; it can be obtained from an NFF in constant time. It should be noted that this metric is different than a simple-minded minimization of the *number* of arithmetic operators in an unscheduled DFG (i.e., in the factored algebraic expression), advocated by other methods.

TDS is intended as a tool for the simplification and factorization of generic data-flow computations, which does not require any knowledge of the design structure. The fundamental limitation of the system is that the TED decomposition relies on variable reordering, which is an expensive operation. This can be alleviated by performing variable ordering incrementally, at each step of the decomposition process (on smaller TEDs) rather than on the original TED.

While, currently, the TDS system works as a front end to an academic high-level synthesis tool, i.e., GAUT, we believe that it could also be useful as a precompilation step in a commercial synthesis environment, such as Catapult C. In this case, the interface can be provided by translating the optimized DFGs into C, to be used as input to those tools.

Several open problems need to be addressed in order to make the TDS system applicable to industrial designs. One of them is the need to handle finite precision computation. TED representation, which is inherently based on an infinite precision model, may produce decompositions that are not optimal for designs with fixed bit widths, as it may introduce unacceptable computational errors. One of the possible research directions is to find decompositions which, under a given bit width, will minimize the amount of error (e.g., modeled as noise). This problem is currently under investigation.

REFERENCES

[1] S. Gupta, M. Reshadi, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Dynamic common sub-expression elimination during scheduling in high-level synthesis," in *Proc. 15th ISSS*, 2002, pp. 261–266.

[2] M. Ciesielski, P. Kalla, and S. Askar, "Taylor expansion diagrams: A canonical representation for verification of data flow designs," *IEEE Trans. Comput.*, vol. 55, no. 9, pp. 1188–1201, Sep. 2006.

[3] M. Ciesielski, S. Askar, D. Gomez-Prado, J. Guillot, and E. Boutillon, "Data-flow transformations using Taylor expansion diagrams," in *Proc. Des. Autom. Test Eur.*, 2007, pp. 455–460.

[4] J. Guillot, "Optimization techniques for high level synthesis and pre-compilation based on Taylor expansion diagrams," Ph.D. dissertation, Lab-STICC, Université Bretagne Sud, Lorient, France, Oct. 2008.

[5] *TDS—TED-Based Dataflow Decomposition System*, Univ. Massachusetts, Amherst, MA. [Online]. Available: http://www.ecs.umass.edu/ece/labs/vlsicad/tds.html

[6] V. Chaiyakul, D. Gajski, and R. Ramachandran, "High-level transformations for minimizing syntactic variances," in *Proc. Des. Autom. Conf.*, 1993, pp. 413–418.

[7] M. Potkonjak and J. Rabaey, "Optimizing resource utilization using transformations," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 3, pp. 277–292, Mar. 1994.

[8] M. Srivastava and M. Potkonjak, "Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughtput," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 3, no. 1, pp. 2–19, Mar. 1995.

[9] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Using global code motion to improve the quality of results in high level synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 23, no. 2, pp. 302–312, Feb. 2004.

[10] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[11] J. Ullman, *Computational Aspects of VLSI*. Rockville, MD: Comput. Sci., 1983.

[12] K. Wakabayashi, *Cyber: High Level Synthesis System From Software into ASIC*. Norwell, MA: Kluwer, 1991, pp. 127–151.

[13] K. Wakabayashi, "Cyberworkbench: Integrated design environment based on C-based behavior synthesis and verification," in *Proc. DAC*, Apr. 2005, pp. 173–176.

[14] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau, *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Norwell, MA: Kluwer, 2004.

[15] *High Level Synthesis Tool: Catapult-C*, Mentor Graphics, Wilsonville, OR. [Online]. Available: http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/

[16] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, *High Level Synthesis From Algorithm to Digital Circuits*. New York: Springer-Verlag, 2008.

[17] A. Peymandoust and G. DeMicheli, "Application of symbolic computer algebra in high-level data-flow synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 22, no. 9, pp. 1154–1165, Sep. 2003.

[18] *Maple*, Maplesoft, Waterloo, ON, Canada. [Online]. Available: http://www.maplesoft.com

[19] *Mathematica*, Wolfram Res., Champaign, IL. [Online]. Available: http://www.wolfram.com
[20] M. Frigo, "A fast Fourier transform compiler," in *Proc. PLDI*, 1999, pp. 169–180.
[21] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proc. IEEE*, vol. 93, no. 2, pp. 232–275, Feb. 2005.
[22] E. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," ERL, Dept. EECS, Univ. California, Berkeley, CA, Tech. Rep. UCB/ERL M92/41, 1992.
[23] R. Brayton, K. Rudell, R. Vincentelli, and A. Wang, "Multi-level logic optimization and the rectangular covering problem," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 1987, pp. 66–69.
[24] 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. 2005.
[25] A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli, "Partitioned ROBDDs: A compact canonical and efficient representation for Boolean functions," in *Proc. ICCAD*, 1996, pp. 547–554.
[26] R. Bryant and Y. Chen, "Verification of arithmetic functions with binary moment diagrams," in *Proc. Des. Autom. Conf.*, 1995, pp. 535–541.
[27] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1976.
[28] A. Hooshmand, S. Shamshiri, M. Alisafaee, B. Alizadeh, P. Lotfi-Kamran, M. Naderi, and Z. Navabi, "Binary Taylor diagrams: An efficient implementation of Taylor expansion diagrams," in *Proc. ISCAS*, 2005, vol. 1, pp. 424–427.
[29] B. Alizadeh, "Word level functional coverage computation," in *Proc. ASP-DAC*, 2006, pp. 7–12.
[30] D. Gomez-Prado, Q. Ren, S. Askar, M. Ciesielski, and E. Boutillon, "Variable ordering for Taylor expansions diagrams," in *Proc. IEEE Int. High Level Des. Validation Test Workshop*, Nov. 2004, pp. 55–59.
[31] D. Gomez-Prado, "Variable ordering for Taylor expansion diagrams," M.S. thesis, Univ. Massachusetts, Amherst, MA, Jan. 2006.
[32] F. J. Taylor, *Digital Filter Design Handbook*. New York: Marcel Dekker, 1983.
[33] *GAUT, Architectural Synthesis Tool*, Lab-STICC, Université de Bretagne Sud, Lorient, France. [Online]. Available: http://www-labsticc.univ-ubs.fr/www-gaut/

Maciej Ciesielski (SM'95) received the M.S. degree in electrical engineering from Warsaw Polytechnic, Warsaw, Poland, in 1974 and the Ph.D. degree in electrical engineering from the University of Rochester, Rochester, NY, in 1983.

From 1983 to 1986, he was a Senior Research Engineer with GTE Laboratories on the silicon compilation project. In 1987, he joined the University of Massachusetts, Amherst, where he is a Professor and the Associate Department Head of the Department of Electrical and Computer Engineering. He teaches and conducts research in the area of electronic design automation, and specifically, in logic and high-level synthesis, optimization, and verification of digital systems.

Dr. Ciesielski is the recipient of the Doctorate Honoris Causa from the Université de Bretagne Sud, Lorient, France, for his contributions to the field of electronic design automation.

Daniel Gomez-Prado (S'05) received the B.S.E.E. degree from San Marcos University, Lima, Peru, in 2000 and the M.S. degree in electrical and computer engineering (ECE) from the University of Massachusetts, Amherst (UMass), in 2006, where he is currently working toward the Ph.D. degree in the Department of ECE.

He is currently a Research Assistant with the Very Large Scale Integration Computer-Aided Design Laboratory, Department of ECE, UMass. His research interests include high-level synthesis, functional verification, and design space exploration.

Mr. Gomez-Prado has been the recipient of the Fulbright Scholarship, the Isenberg Scholarship, and the Graduate School fellowship at UMass.

Qian Ren (M'09) received the B.S. degree in bioscience and biotechnology from Tsinghua University, Beijing, China, in 1997, the M.S. degree in electrical engineering from Texas A&M University, College Station, in 2000, and the Ph.D. degree from the University of Massachusetts (UMass), Amherst, in 2008.

Between 2003 and 2008, he was a Research Assistant with the VLSI CAD Laboratory, Department of Electrical and Computer Engineering, UMass. His doctoral work focused on algorithmic-level synthesis for data-flow and computation-intensive designs. He is currently a Senior Research and Development Engineer with Synopsys, Inc., Mountain View, CA, developing optical simulation and correction techniques for future silicon manufacturing processes.

Jérémie Guillot received the M.S. degree in microelectronics and the Ph.D. degree from the Université de Bretagne Sud, Lorient, France, in 2004 and 2008, respectively. His Ph.D. focused on the use of Taylor expansion diagrams for high-level synthesis.

He is currently a Postdoctoral Researcher with the Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier, France, where he works on the adaptability of MPSoC architectures.

Emmanuel Boutillon (M'05) received the Engineering Diploma and the Ph.D. degree from the Ecole Nationale Supérieure des Télécommunications (ENST), Paris, France, in 1990 and 1995, respectively.

In 1991, he was an Assistant Professor with the Ecole Multinationale Supéieure des Télécommunications, Dakar, Africa. In 1992, he joined ENST as a Research Engineer, where he conducted research in the field of very large scale integration for digital communications. In 2000, he joined the Laboratoire des Sciences et Techniques de l'Information, de la Communication et de la Connaissance, Université de Bretagne Sud, Lorient, France, as a Professor. His current research interests are on the interactions between algorithms and architectures in the field of wireless communications and signal processing.