Optimization of Data Flow Computations using Canonical TED Representation

M. Ciesielski, *Senior Member, IEEE,* D. Gomez-Prado, *Student Member, IEEE,*
Q. Ren, *Student Member, IEEE,* J. Guillot, *Student Member, IEEE,* E. Boutillon, *Member, IEEE*

***Abstract*—This paper describes an efficient graph-based method to optimize polynomial expressions in data-flow computations with a goal to produce final hardware implementations with minimum latency and/or hardware cost. The method is based on factorization and decomposition of algebraic expressions performed on a canonical representation, called Taylor Expansion Diagram (TED). It targets the minimization of the hardware cost of arithmetic operations, not just the minimization of the number of such operations in the simplified expression, and minimizes the latency of the scheduled implementations. As a result, the generated data flow graphs (DFG) 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 DFGs generated from such optimized expressions produce final implementations with, on average, 15.5% lower latency and 7.6% better area than those obtained using traditional algebraic decomposition techniques. The described method is generic, applicable to arbitrary algebraic expressions and does not require specific knowledge of the application domain.**

***Index Terms*—High Level Synthesis, Algebraic Optimizations, Data Flow Graphs, Taylor Expansion Diagrams, Common Subexpression Elimination.**

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 specifications, 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 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 best hardware implementation in terms of its latency and hardware cost. The optimization involves 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 work. 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, TDS, available online [5].

Experimental results show that the data flow graphs (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 the paper is organized as follows. Section 2 analyzes the state of the art in this field. Section 3 reviews the TED fundamentals, including a method to represent non-linear polynomials as linear TEDs. Section 4 describes the TED decomposition algorithms and introduces a concept of normal factored form. In Section 5 a DFG optimization along with analysis of optimization metrics is presented. Section 6 extends the TED decomposition method to support the replacement of constant multiplications by shifters. Section 7 describes the TDS system. Finally, section 8 presents major results and offers 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, HDL compilers [6], [7], [8], [6], [9], [7], 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, as term rewriting rules to manipulate the algebraic expressions. In general, they do not offer systematic way to optimize the initial design specification or to derive optimum data flow graphs for high-level synthesis. While several high-level synthesis systems, such as Cyber [12] and Spark [13], apply a host of code optimization methods (kernel-based algebraic factorization, branch balancing, speculative code motion methods, dead code elimination, etc.) they do not rely on any canonical representation that would guarantee even local optimality of the transformations. For example, neither the above mentioned tools nor the commercial tools (such as NEC's CyberWorkBench [14] or Mentor's Catapult C [15], [16]) are able to detect that the expression $F = (a+b)c-ac-bc$ reduces to 0, which is intrinsically supported by TED.

Domain Specific Systems. Several systems have been developed for domain-specific applications, such as discrete signal transforms. One such system, FFTW [17], targets code generation for DFT based computation. The most advanced system for DSP code generation, SPIRAL [18], generates optimized implementation of linear signal processing transforms, such as DFT, DCT, DWT, etc. These signal transforms are characterized by 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 [19]. These methods perform factorization and CSE by applying techniques of kernel extraction [20]. Kernel-based decomposition [20] employed by logic synthesis, has been recently adopted to optimize polynomial expressions of linear DSP transforms and non-linear filters [21]. While this method provides a systematic approach to polynomial optimization, the polynomial representation is not canonical, which seriously reduces the scope of optimization.

Cut-based Decomposition. Askar [4] proposed an algebraic decomposition method using Taylor Expansion Diagram (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 is sought that produces a DFG with a desired characteristic (minimum number of operators or minimum latency). The disadvantage of the cut-based method is that it is applicable only to TED graphs with disjoint decomposition property. Many TEDs, however, such as the ones shown in Figures 3 and 4, do not have disjoint decomposition property and thus cannot be decomposed using his method.

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

III. Polynomial Representation using TED

A. TED Formalism

Taylor Expansion Diagram 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 algorithm-oriented applications, such as signal and image processing, with computations modeled as polynomial expressions.

An algebraic, multi-variate expression, $f(x,y,...)$, can be represented using Taylor series expansion w.r.t. variable x around the origin $x=0$ as follows:

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

where $f'(x=0)$, $f''(x=0)$, etc, are the successive derivatives of f w.r.t. x evaluated at $x=0$. For a large class of expressions typically encountered in designs specified at high level, the expansion is finite. The individual terms of the expression are then decomposed iteratively with respect to the remaining variables ($y,..$, etc.), one variable at a time.

The resulting decomposition is stored as a directed acyclic graph, called Taylor Expansion Diagram (TED). The TED nodes represent the individual terms of the expansion, and each TED node is labeled with the name of the decomposing variable. The directed edges represent the relationship between the parent node (an expression to be decomposed) and the children nodes (the subexpressions resulting from the decomposition with respect to a given 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 BDDs [22] or BMDs [23]. 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 root to terminal 1. 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, $a^2 \cdot c$ 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 node a to node 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 multi-variate polynomials [3], [4]. For example, a TED for expression $F = ab+ac$, for variable ordering (a,b,c) naturally represents the polynomial in its factored form, $a(b+c)$. Unfortunately, this efficiency is missing when considering optimization involving non-linear expressions. For example, in the TED for function $F = a^2c + abc$ in Figure 1(a), node a should be factored out, resulting in a more compact form $F = a(a+b)c$, but in its current form, the TED in Fig. 1(a) does not allow for such a factorization.

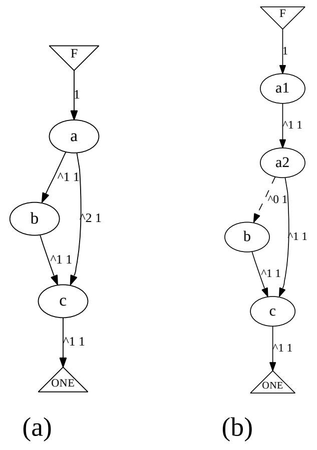

Fig. 1. TED representation for $F = a^2c+abc$: (a) Original, non-linear TED; (b) Linearized TED representing factored form $F = a(a+b)c$.

Fortunately, TED can be readily transformed into a linear form, which supports factorization. Conceptually, a linearized TED represents an expression in which each variable x^k, for $k > 1$, is transformed into a product $x^k = x_1 \cdot x_2 \cdots x_k$, where $x_i = x_j$, $\forall i, j$.

Consider a non-linear expression in Eq.(2). By replacing each occurrence of x^k by $x_1 \cdot x_2 \cdots x_k$, this expression can be transformed into a linear form, shown in Eq.(3), known as Horner form. A characteristic feature of this form is that it contains minimum number of multiplications [24], and hence is suitable for implementations with minimum amount of hardware resources.

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

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

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 $F = a(a + b)c$, see Figure 1(b).

TED linearization can be performed systematically by iteratively splitting the high-order TED nodes until each node represents 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 illustrated in Figure 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 [25] was proposed as a means to improve the efficiency of the internal TED data structure. Other, non-canonical TED-like forms have been used for the purpose of functional test generation for RTL designs [26].

IV. TED Decomposition

This section reviews basic concepts and algorithms of TED-based factorization, common subexpression elimination (CSE), and decomposition of polynomial expressions, jointly referred to as *TED decomposition*.

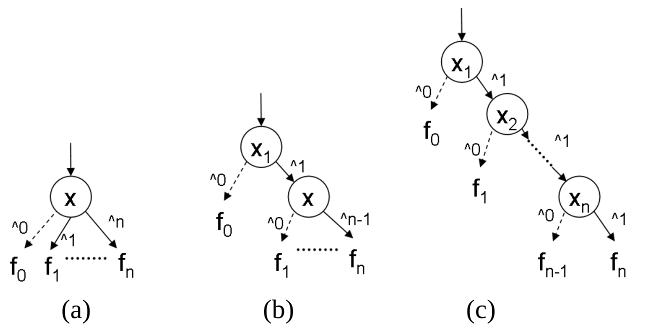

Fig. 2. TED linearization: (a) Original TED containing node x^n; (b) TED after first linearization step; (c) The linearized TED.

A. Basics

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 sub-expression appears more than once in the expression, it can be extracted and replaced by a new variable. This process is known as *common subexpression elimination* (CSE) and results in a decomposed factored form of an expression. Simplification of an expression (or multiple expressions) by means of factorization and common subexpression elimination 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 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 work [3], [4], [21].

The TED decomposition method described here extends the original work of Askar [4] 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 top-down, cut-based decomposition of [4], 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 disjoint decomposition property. The method is based on a series of functional transformations that decompose the graph 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-term and product term extraction;
4) Recursive Taylor decomposition of the top-level graph;
5) DFG construction and optimization.

The first step has been already described in Section III. The issue of variable ordering is intentionally presented last, when the effect of variable ordering on the main decomposition steps becomes well understood. Before describing those steps, we first introduce the *extraction and substitution* operation, sub, which forms the basis of most of the decomposition operations. In the following, all the procedures are applicable to linear TEDs, where each node has at most two edges of different type, multiplicative and/or additive.

B. Subgraph Extraction and Substitution

The extraction & substitution operation 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 [3]. 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 top-most variable in *expr* be v (that is, if this expression were represented by a separate TED, node v would be its top-most 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 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 an arbitrary subexpressions from the graph, regardless of the position of its support variables in the TED and whether they are adjacent or not. 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 Figure 3 for 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.

C. Hierarchical 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.

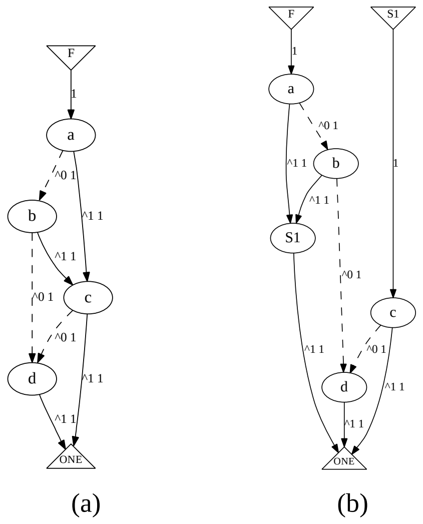

Fig. 3. Extraction of sum-term with term duplication; (a) Original function for $F_0 = (a + b) \cdot (c + d) + d$: (b) Extracting $S_1 = c + d$ with duplicated node d.

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 decomposition of its TED. Figure 4(a) shows the TED for this expression for a particular variable order.

Note that the TED in Figure 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 multiplicative cut (dominator nodes) that would decompose it conjunctively into disjoint subgraphs.

1) Product-Term Extraction: The *product term* is simply defined as a product of variables, Πv_i, of the expression. Product terms can be identified in the TED as a set of nodes connected by a series of multiplicative edges only, such that the intermediate nodes in the series do not have additive incoming or outgoing edges (this pattern is similar to that of an AND function in a BDD). Only the starting and ending nodes can have incident additive edges; the ending node can also be the terminal node 1.

The term $z{\cdot}u$ in the TED Fig. 4(a) is an example of a product term that can be extracted from the graph. It corresponds to a series of multiplicative edges $(z, u), (u, 1)$, where 1 is the terminal node 1 (ONE). Note that the intermediate node u has no additive edge, and hence can be included in the product term. Such terms are extracted from the TED (expression) and 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,

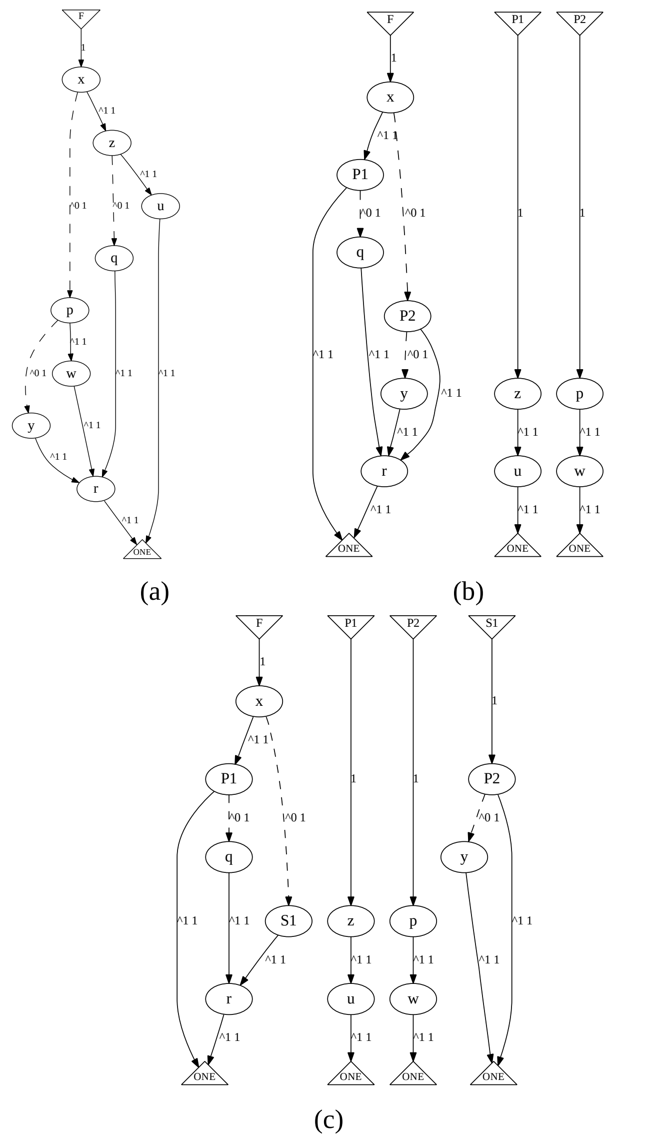

Fig. 4. TED decomposition for expression $x{\cdot}z{\cdot}u{+}p{\cdot}w{\cdot}r{+}x{\cdot}q{\cdot}r{+}y{\cdot}r$: (a) Original TED; (b) Simplified hierarchical TED after product term extraction, $P_1 = z \cdot u$ and $P_2 = p \cdot w$; (c) Final hierarchical TED after sum-term extraction, $S_1 = P_2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

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

2) Sum-Term Extraction: Similarly to the 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 (this pattern is similar to that of an OR function in a BDD).

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 Figure 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 above 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 illustrated in Figure 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 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:

$$a \cdot m + b \cdot n + c \cdot m + d \cdot n = a \cdot m + c \cdot m + b \cdot n + d \cdot n \quad (5)$$

The resulting TED is shown in Figure 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

$$a \cdot m + b \cdot n + c \cdot m + d \cdot n = (a + c)m + (b + d)n \quad (6)$$

as shown in Figure 5(c).

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

3) Dynamic Factorization: An alternative approach to TED factorization has been proposed by Guillot et. al. in [3]. 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 can be used for this purpose). The TED variables are then rotated in order to find best candidates 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.

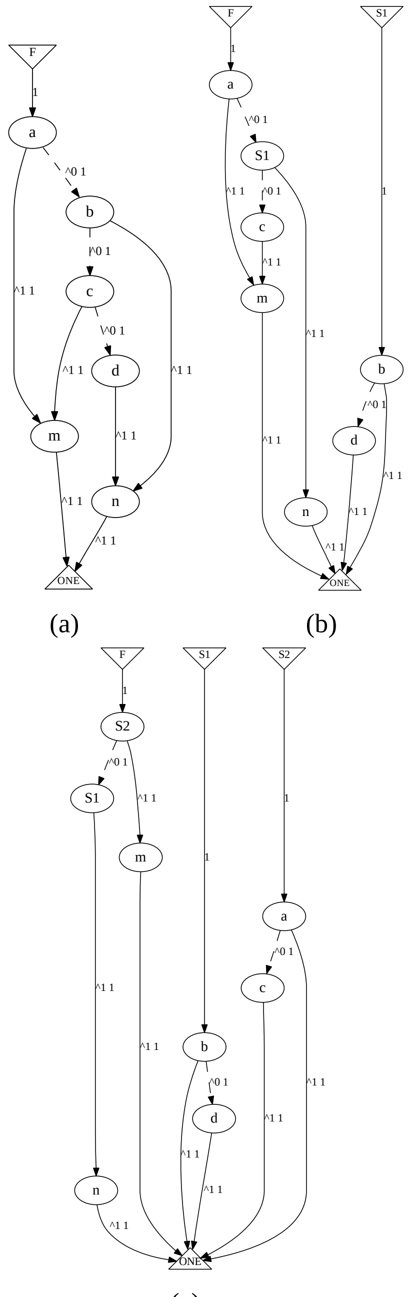

Fig. 5. TED for $F = a \cdot m + b \cdot n + c \cdot m + d \cdot n$: (a) original TED; (b) TED after extracting sum-term $(a + c)$; (c) TED after extracting sum-term $(b + d)$.

Further details of this approach are provided in [3] and [27].

4) 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, 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 the next section.

D. Normal Factored Form

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. We refer to such a form as Normal Factor Form (NFF).

Definition 1: The factored form expression associated with a given TED is called Normal Factored Form (NFF) for that TED if:

- The order of variables in the factored form expression is compatible with the order of variables in the TED;
- There is one-to-one mapping between the arithmetic operations 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 the hierarchical TED obtained from TED decomposition.

To illustrate the concept of normal factored form, consider again the TED in Figure 4(c) and the normal factored form $x \cdot (P_1 + q \cdot r) + S_1 \cdot r$, resulting from TED decomposition, given in equation (8). The variables in this equation do appear in the order compatible with the top-level TED. In particular, the term $x \cdot (P_1 + qr)$ 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, 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 NFF.

We should emphasize that the one-to-one mapping between the arithmetic operations in NFF and the TED edges applies to the irreducible TEDs, obtained by TED decomposition, and not to the initial TED. While the initial TED in our example (prior to its decomposition) has 6 nontrivial multiplicative edges, the hierarchical TED has 5 such edges, corresponding to the 5 multiplications of the NFF expression. (Nontrivial edges are those that are connected to the internal nodes of the graph, not to the terminal 1). 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 $S_1 = P_2 + y$.

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

Lemma 1: Normal Factored Form derived from a linear TED is unique.

The proof comes directly from the construction of the TED decomposition algorithm, described in Section IV.

E. Impact of Variable Ordering:

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. For this reason TED variable ordering plays central role in deriving decompositions that will lead to efficient hardware implementations. Several variable ordering algorithms have been developed, including static ordering and dynamic re-ordering schemes [28], [29], similar to those in 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 NFF (the number of multiplicative edges in the decomposed TED) and the structure of the resulting DFGs, rather than by the number of TED nodes.

V. DFG Optimization

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

Each irreducible TED is first transformed into a simple DFG using the basic property of the normal factored form: 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. (This operatin is similar to the *sweep* operation, applied to a Boolean network in logic synthesis). 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 been already 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 Figure 4(c) is shown in Figure 6.

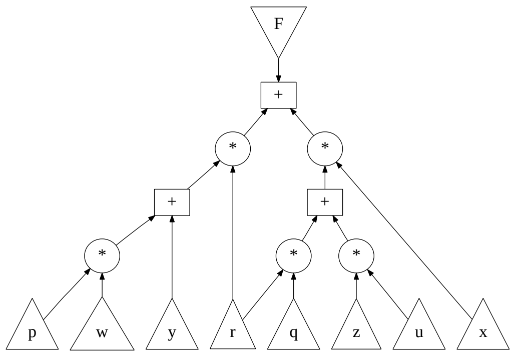

Fig. 6. DFG generated from the decomposed TED and Normal Factored Form: $F = x \cdot (z \cdot u + q \cdot r) + (p \cdot w + y) \cdot r$ show in Figure 4(c).

It should be obvious that, unlike Normal Factored Form, 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 (employed in the TDS system), a chain of product terms or sum terms is replaced by logarithmic trees in each DFGs, and all the component DFGs are composed together to form the final DFG. The actual delays of the operations (taken from the tech library) are considered during the DFG balancing in order to minimize the expected latency.

Since the DFG construction is computationally inexpensive ($O(n)$, for an expression with n literals), its cost in the number of operations and latency can be evaluated quickly from its normal factored form. It is precisely this metric, involving the number of operations (specifically multiplications, which have higher implementations cost) and the estimated DFG latency that is used to guide the TED decomposition.

In summary, two basic mechanisms are used during the decomposition to obtain DFGs with minimum expected latency and/or minimum number of operations (1) Variable ordering, including implicit reordering during the term extraction and substitution, as described in Section IV-C2; and (2) DFG balancing to minimize latency.

VI. Replacing Constant Multipliers by Shifters

Multiplications by constants are common in designs involving linear systems, especially in computation intensive applications such as DSP. It is well known that multiplications by integers can be implemented more 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 [30]. However, these methods do not address common subexpression elimination or factorization involving shifters.

We now present a systematic way to transform integer multiplications into shifters using the TED structure. 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 known TED simplification methods.

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 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 shift variables, L^k, are replaced by actual shifters (by k bits). Such a replacement minimized hardware cost of the operators.

Example in Figure 7 illustrates this procedure for the expression $F = 7a + 6b$. The original TED, shown in Figure 7(a), is transformed into an expression with the shift variable L:

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

shown in Figure 7(c). The term L^3 is then linearized and the TED ordered, as shown in Figure 7(d). The TED is then decomposed into a DFG, shown in Figure 7(e). After replacing variables L_i by L, the DFG in Figure 7(f) is obtained. Finally, all constant multiplications with inputs L^k are replaced by k-bit shifters, as shown in Figure 7(g). The optimized expression corresponding to this DFG is

$$F = ((a + b) << 2 - b) << 1 - a \tag{11}$$

where the symbol"$<< k$" refers to left shift by k bits This implementation requires only three adders/subtracters and two shifters, a considerable gain compared to the two multiplications and one addition of the original expression $F = 6a + 7b$.

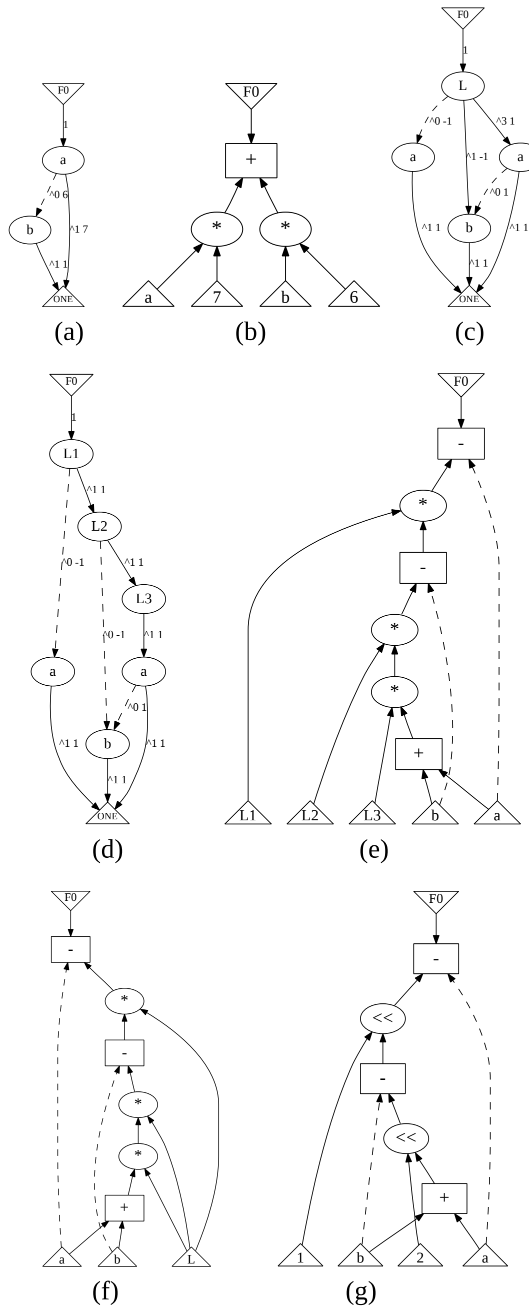

Fig. 7. Replacing constant multiplications by shift operations for expression $F_0 = 7a + 6b$: (a) Original TED; (b) Initial DFG; (c) TED after introducing a *shift* variable L; (d) Linearized TED; (e) DFG derived from the linearized TED; (f) DFG obtained after combining variables L_i into L; (g) Final DFG after replacing multipliers by shifters.

VII. TDS System

The TED decomposition and DFG optimization methods presented in this paper were implemented as part of an experimental software system, TDS. The system performs behavioral transformation of a design, specified on an algorithmic or behavioral level, into a data flow graph (DFG), optimized for latency and/or resource utilization, prior to high-level synthesis. The system is intended for data-flow and computation-intensive designs used in digital signal processing 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 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.

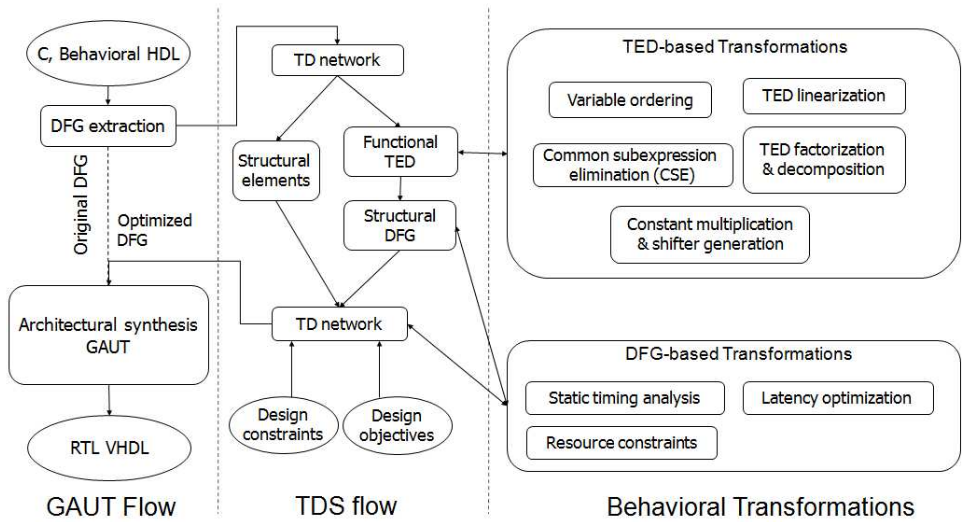

Fig. 8. TDS system flow

The overall TDS system flow is shown in Figure VII. The left part of the figure shows traditional high-level synthesis flow. It is based on high-level synthesis tool GAUT [31], which extracts the data flow graph from the initial specification. The flow on the right is the TDS system that transforms the extracted data flow graph 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 pre-coded 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.

Command	Description
vars	Define the order of variables
poly	Input the polynomial expression
read	Read optimization script or cdfg file
tr	Generate polynomial expressions for transform
flatten	Create global TED by flattening the description
shifter	Replace constant multipliers by variables L
show	Show the TED (-t) or the DFG (-d)
linearize	Linearize the TED
reorder	Reorder to minimize multipliers (-m) or nodes (-n)
top	Move variable to the top
bottom	Move variable to the bottom
reloc	Relocate variable to a given level
candidate	List candidate expressions for extraction
sub	Substitute expression by a new variable
extract	Extract TED associated with a particular output
dfactor	Perform dynamic factorization
print	Print out the NFF (-f) or statistics (-s)
dfgbalance	Balance the resulting DFG
write	Write output file in cdfg format

TABLE I
TED AND DFG OPTIMIZATION COMMANDS.

VIII. Experimental Results

The TED decomposition described in this paper was implemented as part of a prototype system 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.The optimization uses the optimization scripts, which include one or more steps of TED ordering, factorization, replacement of constant multipliers by shifters, 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 (based on a Xilinx FPGA family): multiplier=18ns, adder/sub=8ns, shifter=9ns; and clock period=10ns. Notice that multiplication requires two clock cycles. Table II compares the implementation of a SG filter using: 1) the original design; 2) the design produced by CSE decomposition system of [21]; and 3) and the 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, respectively. Each of the remaining rows shows 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 area for datapath 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 '–' (over-constrained).

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 is available from literature for the CSE system [21]. Also, we do not report the circuit delay produced by Synopsys, as it is almost identical for all solutions (this is not surprising, since it is limited to the delay of the multiplier).

Design		Original design		CSE solution		TDS solution	
	Latency (ns)	$+,\times,\ll,-$	Area GAUT / SynDC	$+,\times,\ll,-$	Area GAUT / SynDC	$+,\times,\ll,-$	Area GAUT / SynDC
SG Filter	DFG $\longrightarrow$	2,16,6,0		4,14,3,0		6,11,3,0	
	L=120	–	–	1,5,2,0	439	1,4,1,0	348
					22,057		20,849
	L=130	–	–	1,5,1,0	431	2,3,1,0	273
					22,057		18,021
	L=140	–	–	1,4,1,0	348	1,3,1,0	265
					19,952		18,160
	L=150	–	–	1,4,1,0	348	1,3,1,0	265
					19,648		17,862
	L=160	1,4,2,0	356	1,3,1,0	265	1,2,1,0	182
			20,442		17,428		14,795

TABLE II
SG FILTER IMPLEMENTATIONS SYNTHESIZED WITH THE GAUT AND SYNOPSYS DC.

As seen in table II, the minimum latency for the DFG extracted from the original design, without any modification, is 160ns. The DFG solution produced by both CSE and TDS has minimum latency of 120ns, a 25% improvement w.r.t. the original design. However, the TDS implementation requires smaller area than both the original and CSE synthesized 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, computer graphic algorithms, etc.), synthesized with GAUT. As an example, examine the Quintic Spline design, for which the CSE solution had smallest number of operations in the unscheduled DFG, and the latency 120 ns. The DFG obtained by TDS produced the implementation with 110 ns, i.e., 21.42% faster, even though it had more DFG operations. Furthermore, for the minimum latency of 120 ns, obtained by CSE, TDS produces implementation with area 22.02% smaller than that of CSE. Similar behavior can be seen in all the remaining designs. In all cases the latency of the scheduled DFGs produced by TDS is smaller; and, with the exception for Quartic Spline design, all of them have also smaller hardware area for the minimum latency produced by CSE.

Table IV summarizes the implementation results for these benchmarks. We can see that the implementations obtained from TDS have latencies smaller on average by 15.51% and 27.22% w.r.t. the CSE and original DFGs, respectively. And for the reference latency (defined as the minimum latency obtained by the other two methods), the TDS implementations have, on average, 7.64% (w.r.t CSE) and 36.33% (w.r.t. original) smaller area.

Design		Original design		CSE solution		TDS solution	
	Latency (ns)	$+,\times,\ll,-$	Area	$+,\times,\ll,-$	Area	$+,\times,\ll,-$	Area
Cosine wavelet	DFG $\longrightarrow$	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 $\longrightarrow$	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 $\longrightarrow$	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 $\longrightarrow$	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 $\longrightarrow$	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 $\longrightarrow$	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 III
COMPARISON OF MINIMUM ACHIVABLE LATENCY AND AREA FOR DIFFERENT DESIGNS. THE AREA REPORTED IS FOR GAUT.

Design	TDS vs			
	Original		CSE	
	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

TABLE IV
PERCENTAGE IMPROVEMENT OF TDS VS ORIGINAL AND CSE ON ACHIEVABLE LATENCY; AND AREA AT THE MINIMUM ACHIEVABLE LATENCY.

IX. Conclusions and Future Work

As demonstrated by the presented results, a simple-minded minimization of the *number* of arithmetic operations in an algebraic expression, advocated by other researchers in this field, does not necessarily translate into a minimum hardware cost or a minimum latency in a scheduled DFG; as such it does guarantee a good hardware implementation. Such a blind minimization can even hinder other implementations, that are either faster or that require fewer resources for a given latency constraint. We have implemented a new approach to the optimization of polynomial expressions that yields DFGs better suited for high-level synthesis and produces better hardware implementations. The presented approach discovers solutions that have lower latency than those obtained by simply minimizing the number of arithmetic operations; and for the minimum latency obtained by those methods, requires on average smaller hardware area.

While currently the TDS system is integrated with an academic high-level synthesis tool, GAUT, we believe that it could be beneficial as a pre-compilation step in a commercial synthesis software, 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. Finally, to turn TDS into a commercial-quality system, a finite precision of the operators need to be addressed. This can be done by extending TED representation, which is inherently based on infinite precision arithmetic, to handle finite precision computations. This is a subject of the ongoing work.

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 *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, New York, NY, USA, 2002, pp. 261–266, ACM Press.

[2] M. Ciesielski, P. Kalla, and S. Askar, "Taylor Expansion Diagrams: A Canonical Representation for Verification of Data Flow Designs", *IEEE Trans. on Computers*, vol. 55, no. 9, pp. 1188–1201, Sept. 2006.

[3] J. Guillot, E. Boutillon, D. Gomez-Prado, S. Askar, Q. Ren, and M. Ciesielski, "Efficient Factorization of DSP Transforms using Taylor Expansion Diagrams", in *Design Automation and Test in Europe, DATE-06*, 2006.

[4] M. Ciesielski, S Askar, D. Gomez-Prado, J. Guillot, and E. Boutillon, "Data-Flow Transformations using Taylor Expansion Diagrams", in *Design Automation and Test in Europe*, 2007, pp. 455–460.

[5] M. Ciesielski, D. Gomez-Prado, and Q. Ren, "TDS - taylor decomposition system. software for optimizing data flow graphs through ted optimization", http://incascout.ecs.umass.edu/tds.

[6] M. Potkonjak and J. Rabaey, "Optimizing Resource Utilization Using Transformations", in *IEEE Transactions on Computer Aided Design*, 1994.

[7] S. Gupta, N. Savoiu, N.D. Dutt, R.K. Gupta, and A. Nicolau, "Using Global Code Motion to Improve the Quality of Results in High Level Synthesis", *IEEE Trans. on CAD*, pp. 302–311, 2004.

[8] V. Chaiyakul, D. Gajski, and R. Ramachandran, "High-level transformations for minimizing syntactic variances", *Design Automation Conf.*, pp. 413–418, 1993.

[9] M. Srivastava and M. Potkonjak, "Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughtput", in *IEEE Transactions on VLSI Systems*, 1995.

[10] G. DeMicheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 94.

[11] J.D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, Rockville, Maryland, 1983.

[12] K. Wakabayashi, *Cyber: High Level Synthesis System from Software into ASIC*, pp. 127–151, Kluwer Academic Publishers, 1991.

[13] S. Gupta, R.K. Gupta, N.D. Dutt, and A. Nicolau, *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*, Kluwer Academic Publishers, 2004.

[14] Kazutoshi Wakabayashi, "Cyberworkbench: integrated design environment based on c-based behavior synthesis and verification", in *DAC*, April 2005, pp. 173–176.

[15] Mentor Graphics, "High Level Synthesis Tool: Catapult-C", 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*, Springer, 2008.

[17] M. Frigo, "A Fast Fourier Transform Compiler", in *Proc. PLDI*, 1999, pp. 169–180.

[18] M. Püschel, J.M.F. Moura, J. Johnson, D. Padua, M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms", *Proceedings of the IEEE*, vol. 93, no. 2, 2005.

[19] E. Sentovich *et al.*, "SIS: A System for Sequential Circuit Synthesis", Tech. Rep. UCB/ERL M92/41, ERL, Dept. of EECS, Univ. of California, Berkeley., 1992.

[20] R. Vincentelli A. S. Brayton R, K. Rudell and Wang A., "Multi-level logic optimization and the rectangular covering problem", in *Proc. Int. Conf. Comput.-Aided Des.*, November 1987.

[21] A. Hosangadi, F. Fallah, and R. Kastner, "Optimizing Polynomial Expressions by Algebraic Factorization and Common Subexpression Elimination", in *IEEE Transactions on CAD*, Oct 2005, vol. 25, pp. 2012–2022.

[22] A. Narayan and *et al.*, "Partitioned ROBDDs: A Compact Canonical and Efficient Representation for Boolean Functions", in *Proc. ICCAD*, '96.

[23] R. E. Bryant and Y-A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams", in *Proc. Design Automation Conference*, 1995, pp. 535–541.

[24] A.. Aho, J. Hopcroft, and J. Ullman, *The Design And Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1976.

[25] 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 *ISCAS (1)*. 2005, pp. 424–427, IEEE.

[26] Bijan Alizadeh, "Word level functional coverage computation", in *ASP-DAC*, Fumiyasu Hirose, Ed. 2006, pp. 7–12, IEEE.

[27] Jeremie Guillot, *Optimization Techniques for High Level Synthesis and Pre-Compilation Based on Taylor Expansion Diagrams*, PhD thesis, Lab-STICC, Universié Bretagne Sud, 2008.

[28] D. Gomez-Prado, Q. Ren, S. Askar, M. Ciesielski, and E. Boutillon, "Variable Ordering for Taylor Expansion Diagrams", in *IEEE Intl. High Level Design Validation and Test Workshop, HLDVT-04*, 2004, pp. 55–59.

[29] Daniel Gomez-Prado, "Varaible Ordering for Taylor Expansion Diagrams", Master's thesis, University of Massacusetts Amherst, 2006.

[30] Fred. J. Taylor, *Digital Filter Design Handbook*, Marcel Dekker, 1983.

[31] Université de Bretagne Sud Lab-STICC, "GAUT, Architectural Synthesis Tool", http://http://www-labsticc.univ-ubs.fr/www-gaut/, 2008.