High-level Transformations of Data Flow Computations using Canonical TED Representation

M. Ciesielski, J. Guillot*, D. Gomez-Prado, E. Boutillon*
ECE Dept., University of Massachusetts, Amherst, MA, USA
*Lab-STICC, Université de Bretagne Sud, Lorient, France

Abstract

This article describes a systematic method and an experimental software system to perform high-level transformation of the functional design specifications prior to high level synthesis. The initial specification is first transformed into a canonical form and then converted into a data flow graph (DFG) optimized for a particular application. The optimizing transformations are based on a canonical Taylor Expansion Diagram (TED) representation. The system is intended for data flow and computation-intensive designs used in computer graphics and digital signal processing applications. This methodology has been implemented in an experimental software tool, TDS. Results of several experiments are provided.

Keywords: *High-level synthesis, data flow graphs, symbolic algebra, Taylor Expansion Diagrams.*

1 Introduction

A considerable progress has been made during the last two decades in behavioral and High-Level Synthesis (HLS), making it possible to synthesize designs specified using standard programming languages, such C, C++ or system C [1]. Those tools automatically generate a Register Transfer Level (RTL) specification of the circuit from a bit-accurate algorithm description for a given target technology and the application constraints (latency, throughput, resource utilization, etc.). The algorithmic description used as input to high-level synthesis does not require explicit timing information and thus provides a higher level of abstraction than the RTL model. Thanks to high-level synthesis, the designer can faster and more easily explore different architectural solutions. An important productivity leap is thus achieved.

However, the optimization algorithms used by high-level synthesis, such as scheduling, resource binding and allocation, operate on a fixed Data Flow Graph (DFG), extracted directly and without any modification from the initial design specification. In this approach the scope of the ensuing architectural optimization and the quality of the resulting hardware implementation strongly depends on such a specification. In order to explore other solutions the user needs to rewrite the original specification, from which another DFG is derived and synthesized.

A novel method is proposed here whereby the initial functional specification is first transformed into a canonical form and then converted into a DFG optimized for a particular application. Such a transformation and DFG optimization is carried in a systematic way using a canonical representation.

Related Work: Automatic transformation of design specification is an old concept. In fact, software compilers commonly use such optimization techniques as dead code elimination, constant propagation, common subexpression elimination, and others, to simplify and optimize the target code implementation. Some of those compilation techniques are also used by academic and commercial HLS tools, such as Spark [2], Cyber [3] and Catapult-C [1]. In general, these methods rely on manipulations of algebraic expressions based on term rewriting and basic algebraic properties (associativity, commutativity, and distributivity) that do not guarantee optimality. Algebraic factorization decomposition methods, successfully used in logic synthesis, have also been used to optimize polynomial expressions of linear DSP transforms and non-linear filters [4]. However, the polynomial representation employed by these methods is not canonical, which seriously reduces the scope of optimization.

A number of systems have been developed for domain-specific applications such as digital filters and discrete signal transforms [5]. Optimizations employed by these systems rely on the knowledge of mathematical properties of the transforms. In general, they cannot handle computations described by nonlinear polynomials, such as those found in computer graphics and nonlinear filter applications.

This article describes a systematic method for transforming an initial design specification of a *generic* nonlinear computation into an optimized DFG. The generated DFGs are better suited for high-level synthesis than those extracted directly from the initial specification, or obtained using structural DFG transformations and algebraic decomposition methods. The optimizing transformations are based on a *canonical*, graph-based representation, called Taylor Expansion Diagram (TED) [6]. The goal is to generate a DFG, which, when given as input to a standard high-level synthesis tool, will produce hardware implementation optimized for latency and/or hardware cost.

Figure 1 illustrates the idea of design space exploration based on functional TED representation. In traditional synthesis flow a single DFG is extracted from the initial (functional) specification and used by high-level synthesis tools to generate the final implementation (hardware architecture). Optimization of the resulting architectural design obtained from a fixed DFG is limited to local modifications on the RTL level. The set of such solutions is shown in the figure as a cone associated with a given DFG. In order to improve

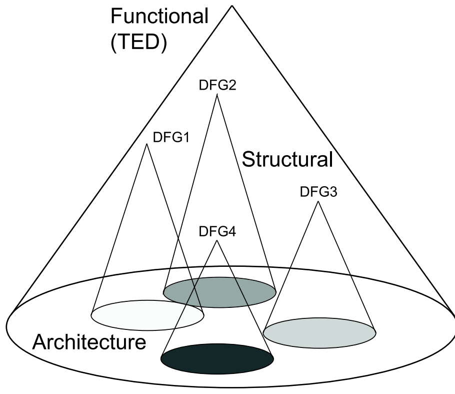

Figure 1. Design space exploration from functional canonical specification.

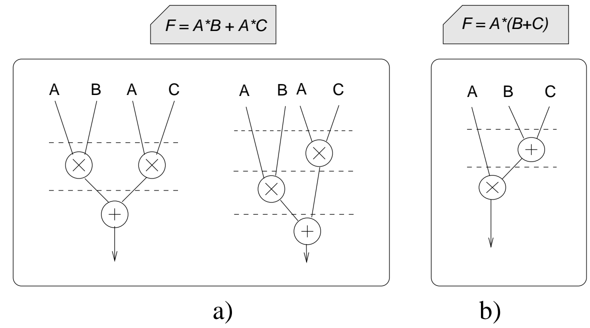

Figure 2. Effect of high-level transformations on DFG structure: a) Initial design specification $F = AB + AC$ and the corresponding scheduled DFGs; b) Transformed specification $F = A(B + C)$ and the resulting DFG.

the solution, an attempt can be made to transform the DFG into another DFG. However, such a transformation, if at all possible, is limited to structural modifications of the graph, such height tree reduction and balancing, which are limited in scope and can explore only a small fraction of the entire solution space.

In contrast, the transformations of the design specification performed on the functional level can produce a set of functionally equivalent DFGs. One such DFG can be selected, based on a given objective and design constraints, to generate the final architecture. This approach provides a much larger scope of architectural optimization and has the biggest impact on the final hardware implementation.

To illustrate the concept of functional-level transformations supported by TED consider a simple computation $F = AB + AC$, where variables A, B, C are word-level signals. Figure 2(a) shows two possible schedules of a DFG derived directly from this expression. The two DFGs have the same structure and differ only in the scheduling of arithmetic operations. The DFG on the left has minimum latency and requires two multipliers and one adder. The one on the right needs only one multiplier and one adder, at a cost of the increased latency.

Figure 2(b) shows a solution that can be obtained by transforming the original specification $F = AB + AC$ into $F = A(B+C)$, which results in a *different DFG*. This DFG requires only one adder and one multiplier and can be scheduled in two control steps. Such an implementation cannot be obtained from the initial DFG by simple structural transformation and requires *functional* transformation (in this case factorization) of the original expression which preserves its original behavior. The solution with best hardware cost can then be chosen for the final hardware implementation.

The remainder of the article describes how such a transformation and the optimization of the corresponding DFG can be obtained using a canonical TED representation. The optimizing transformations are implemented in an experimental software system, called TDS, intended for data-flow and computation-intensive designs used in computer graphics and digital signal processing applications. The system is available online [7].

2 Taylor Expansion Diagrams (TED)

Taylor Expansion Diagram is a compact, graph-based data structure that provides an efficient way to represent word-level computation in a canonical, factored form [6]. It is particularly suitable for computations modeled as polynomial expressions. Here we briefly review the basic formalism of TED, described in detail in [6].

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

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

where $f'(0, y, \ldots)$, $f''(0, y, \ldots)$, etc, are the successive derivatives of f w.r.t. x, evaluated at $x = 0$. The individual terms of the expression are then decomposed iteratively with respect to the remaining variables on which they depend ($y, \ldots$, etc.), one variable at a time.

The resulting decomposition is stored as a directed acyclic graph, called Taylor Expansion Diagram (TED). Each node of the TED is labeled with the name of the variable at the current decomposition level and represents the expression rooted at this node. The top node of the TED represents the main function $f(x, y, \ldots)$ and is associated with the first variable, x. Each term of the expansion at a given decomposition level is represented as a directed edge from the current decomposition node to its respective derivative term.

Each edge is labeled with a pair $(\wedge p, w)$, where $\wedge p$ represents the power of the corresponding variable and w represents the edge weight (multiplicative constant) associated with this term. Here we concentrate on a class of *linear* TEDs, which represent linear multi-variate polynomials. Nonlinear expressions can be easily converted into linear ones, by transforming each occurrence of a nonlinear term x^k into a product $x_1 \cdots x_k$, where $x_i = x_j$. A linear TED contains only two types of edges: the *additive* edges (labeled with power $\wedge 0$), represented in the graph as dotted edges; and the *linear* edges (labeled with $\wedge 1$), represented as solid lines. Linear edges are also referred to as *multiplicative* edges, since they represent multiplication of terms. Explicit labels on the edges can be dropped whenever the graph con-

tains only additive and linear edges with weight 1. Edges with weight 0 are not shown as they correspond to empty terms.

The expression encoded in the TED graph is computed as a sum of the expressions of all the paths, from the TED root to terminal 1. An expression for each path is obtained as a product of the edge expressions, each being a product of the variable in its respective power and the corresponding edge weight. Only non-trivial terms, corresponding to edges with non-zero weights, are stored in the graph.

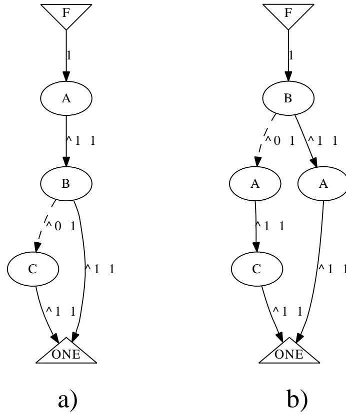

Figure 3. TED for expression $F = AB + AC$ for different ordering of variables: a) TED for variable order A, B, C; b) TED for variable order B, A, C.

The construction of a TED is illustrated in Figure 3 for an expression $F(A, B, C) = AB+AC$ for two variable orders. First assume the variable order $\{A, B, C\}$, see Figure 3(a).

- First level of decomposition, associated with variable A: The term $F(A = 0, B, C) = 0$ corresponds to an additive edge leading to an empty term (not shown). The next term, $F'(A = 0, B, C) = B + C$, is represented by a linear edge, labeled $(^\wedge 1, 1)$, leading to node B. Let us denote the term $F' = B + C$ by $G(B, C)$.
- Second decomposition level, associated with variable B: The term $G(B = 0, C) = C$ corresponds to an additive edge, labeled $(^\wedge 0, 1)$, incident to node C. The term $G'(B = 0, C) = 1$ is a linear edge, labeled $(^\wedge 1, 1)$, connected to the terminal node 1.
- Third level, variable C: $C(0) = 0$ is an empty additive term (not shown). The last term, $C' = 1$, is a linear edge leading to the terminal node 1.

The resulting TED is shown in Figure 3(a). The expression encoded in the graph is computed as a sum of two paths from TED root to terminal node 1: $A \cdot B$ and $A \cdot B^0 \cdot C = A \cdot C$, i.e., $AB + AC$. In fact, TED encodes such an expression in *factored form*, $F = A(B + C)$, since variable A is common to both paths. This is manifested in the graph by the presence of the subexpression $(B+C)$, rooted at node B, which can be factored out. This feature of the TED structure is particularly useful for effecting factorization and common subexpression extraction of algebraic expressions. TED construction for variable order $\{B, A, C\}$ results in a graph shown in Figure 3(b). Note that this ordering does not expose any particular factored form, and hence is not useful for factorization.

In summary, TED is a graphical representation of finite multi-variate polynomials which maps word-level (integer) inputs into word-level outputs. TED is reduced and normalized in a similar way as BDDs and BMDs [8]. Finally, the reduced, normalized and ordered TED is canonical for a given variable order. It should be pointed out that despite 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, b are single bit variables; otherwise each input must be represented in terms of its component bits. A detailed description of the TED representation with its application to verification and high-level synthesis can be found in [6] and [10].

3 TED-based Decomposition

The principal goal of algebraic factorization and decomposition is to minimize the number of arithmetic operations (additions and multiplications) in the algebraic expression. A simple example of *factorization* is the transformation of the expression $F = AB + AC$ into $F = A(B + C)$, 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, which reduces the overall complexity of an expression and its hardware implementation. This process is known as *common subexpression elimination* (CSE). Simplification of an expression (or of a set of expressions) by means of factorization and CSE is commonly referred to as *decomposition*.

Decomposition of algebraic expressions can be performed directly on the TED graph, as it already encodes the expression in a compact, factored form. The goal of TED decomposition is to find a factored form of the expression and the corresponding DFG that will minimize the latency and/or hardware cost of the scheduled and synthesized design.

This section describes two methods for TED decomposition, each applicable to a different class of designs. One is based on factorization and common subexpression extraction performed on a *static* TED, with a fixed variable order. This method is applicable to generic algebraic expressions that do not exhibit any particular structure. The other method is based on *dynamic* CSE, where common subexpressions are derived by dynamically modifying the TED variable order in a systematic way. This method is more suitable for structured linear DSP transforms, with common computing patterns.

3.1 Static TED Decomposition

Static decomposition consist of hierarchical cut-based decomposition and subgraph extraction, followed by transformation of the decomposed TED into a data flow graph (DFG). The details of the complete TED decomposition procedure are given in [9, 10].

Cut-based decomposition is based on identifying a sequence of cuts in the TED structure that decomposes the TED into disjoint subgraphs [9]. An *additive cut*, denoted by A_i, applied to an additive edge, partitions the graph into two sub-graphs, resulting in a disjunctive decomposition: $F = F_1 + F_2$. A *multiplicative cut*, denoted by M_i, is applied to special nodes called *dominators*. A dominator is a node with a property that all the paths from the TED root to terminal node 1 must pass through this node. Cutting the TED at such a node decomposes the expression conjunctively: $F = F_1 \cdot F_2$, where F_1, F_2 are the subgraphs above and below the cut.

Figure 4(a) shows the hierarchical TED decomposition for an expression $P = G + H + F(I + J)$. Each time an additive or multiplicative cut is applied to a TED, a hardware operator (ADD or MULT) is introduced in the the resulting DFG to perform the required operation on the two subexpressions. This way, the *functional* TED representation is eventually transformed into a *structural* DFG representation.

An important property of this decomposition is that different cut sequences generate different DFGs, from which the one with the required property (such as minimum latency or resource utilization) can be selected. Figures 4(b),(c) show two DFGs resulting from two different cut sequences, (A_3, A_1, M_1, A_2) and (A_1, A_3, M_1, A_2), applied to the TED in Figure 4(a). The two DFGs differ in their structure (tree vs serial) and hence will produce different hardware architectures.

In summary, the cut-based decomposition is a hierarchical, top-down process in which a TED is successively decomposed into smaller, disjoint subgraphs until reaching a trivial structure composed of only a single node connected to the terminal node 1 by a multiplicative edge. It should be noted that the cut-based decomposition is applicable only to those TEDs, which at every decomposition step have additive or multiplicative cuts, resulting in disjoint subgraphs. The decomposition of TEDs that do not exhibit such a property must be handled differently.

A non-disjoint decomposition is illustrated in Figure 5 for the expression $F = ac + bc + ad + bd + ab$, encoded in the TED in Figure 5(a) with a variable order $\{c, d, a, b\}$. There is no top-level additive or multiplicative cut that can decompose this TED into two disjoint subgraphs. Instead, the A-cut applied to this TED decomposes the graph into two non-disjoint subgraphs, $F_1 = ab$ and $F_2 = (c + d)(a + b)$, with subgraph b being common to both partitions, as shown in Figure 5(b). Such a decomposition is accomplished by explicit extraction of common subgraphs. Expression F_2 is then decomposed using M-cut into $(c + d)$ and $(a + b)$. The resulting decomposition is shown in Figure 5(c).

Figure 5(d) shows a non-disjoint decomposition of the same expression encoded in a TED with a different variable order, namely $\{a, b, c, d\}$. In this case a common subgraph that needs to be extracted is $S = (c + d)$. The resulting decomposition is $F = a(b + S) + b \cdot S$, with $S = (c + d)$.

Note that the resulting expression depends on the structure (and the variable order) of the TED. Such an expression is referred to as Normal Factored Form (NFF) for a given TED. We will formally define NFF and prove its properties

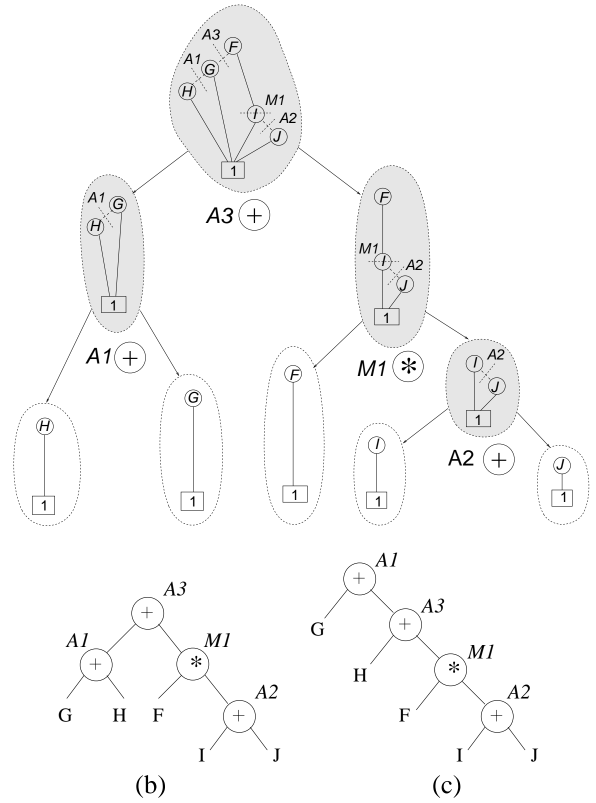

Figure 4. (a) Hierarchical TED decomposition for expression $P = G + H + F(I + J)$ for cut sequence (A_3, A_1, M_1, A_2); (b) DFG obtained for this cut sequence; (c) DFG for cut sequence (A_1, A_3, M_1, A_2).

in Section 4 in the context of DFG generation.

3.2 Dynamic TED Factorization

An alternative approach to TED decomposition is based on a dynamic factorization and common subexpression elimination (CSE). This approach is illustrated with an example of the Discrete Cosine Transform (DCT), used frequently in multimedia applications. The DCT of type 2 is defined as

$$Y(j) = \sum_{k=0}^{N-1} x_k cos[\frac{\pi}{N} j(k + \frac{1}{2})], k = 0, 1, 2, ..., N-1$$

This computation can be represented in matrix form as $y = M \cdot x$, where x and y are the input and output vectors, and M is the transform matrix composed of the cosine terms. Matrix M for $N = 4$ is shown in eq. (2).

$$\mathbf{M} = \begin{bmatrix} cos(0) & cos(0) & cos(0) & cos(0) \\ cos(\frac{\pi}{8}) & cos(\frac{3\pi}{8}) & cos(\frac{5\pi}{8}) & cos(\frac{7\pi}{8}) \\ cos(\frac{\pi}{4}) & cos(\frac{3\pi}{4}) & cos(5\frac{\pi}{4}) & cos(7\frac{\pi}{4}) \\ cos(\frac{3\pi}{8}) & cos(\frac{7\pi}{8}) & cos(\frac{\pi}{8}) & cos(\frac{5\pi}{8}) \end{bmatrix} = \begin{bmatrix} A & A & A & A \\ B & C & -C & -B \\ D & -D & -D & D \\ C & -B & B & -C \end{bmatrix} \quad (2)$$

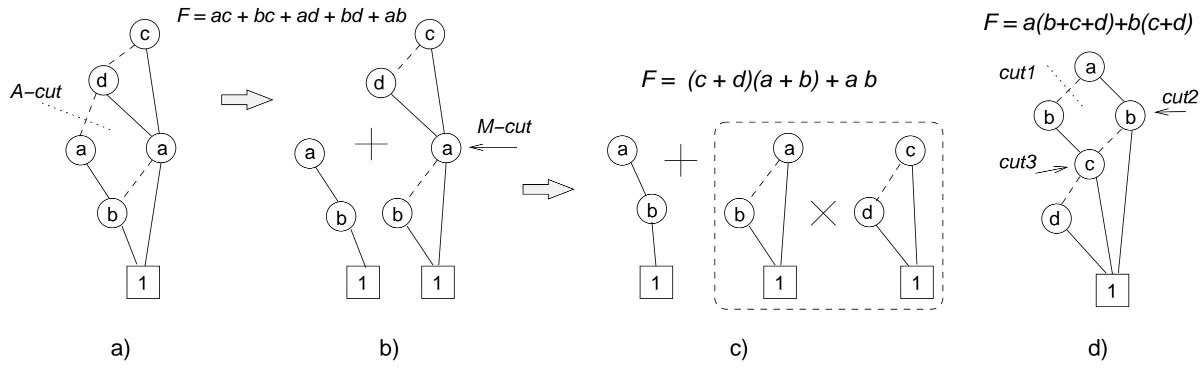

Figure 5. Example of cut-based TED decomposition for expression $F = ac + bc + ad + bd + ab$: a) Initial TED for variable order $\{c, d, a, b\}$; b) Component TEDs after applying additive cut A; c) TEDs after applying multiplicative cut M, and the resulting normal factored form $F = (c+d)(a+b)+ab$; d) TED for variable order $\{a, b, c, d\}$ resulting in normal factored form $F = a(b+S)+b\cdot S$, with $S = (c+d)$.

In its direct form the computation involves 16 multiplications and 12 additions. However, by recognizing the dependence between the cosine terms it is possible to express the matrix using symbolic coefficients, as shown in the above equation. The coefficients with the same numeric value are represented by the same symbolic variable. Matrix M for the DCT example has four distinct coefficients, A, B, C, D. This representation makes it possible to factorize the expressions and reduce the number of operations to 6 multiplications and 8 additions, as shown in eq.(3). This simplification can be achieved by extracting subexpressions $(x_0 + x_3)$, $(x_0 - x_3)$, $(x_1 + x_2)$, and $(x_1 - x_2)$, shared between the respective outputs, and substituting them with new variables.

$$\begin{aligned} y_0 &= A \cdot ((x_0 + x_3) + (x_1 + x_2)) \\ y_1 &= B \cdot (x_0 - x_3) + C \cdot (x_1 - x_2) \\ y_2 &= D \cdot ((x_0 + x_3) - (x_1 + x_2)) \\ y_3 &= C \cdot (x_0 - x_3) - B \cdot (x_1 - x_2) \end{aligned} \quad (3)$$

The initial TED representation for the DCT matrix in eq. (2) is shown in Figure 6(a). The subsequent parts of the figure show the transformation of the TED that produces the above factorization.

The key to obtaining efficient TED-based factorization and CSE for this class of designs is to represent the coefficients of the expressions as variables and to place them on top of the TED graph. This is in contrast to a traditional TED representation, where constants are represented as labels on the graph edges. In the case of the DCT transform, the coefficients A, B, C, D are treated as symbolic variables and placed on top of the TED, as shown in Figure 6(a).

The candidate expressions for factorization in such a TED are obtained by identifying the nodes with multiple incoming (parent) edges. The subexpression rooted at such nodes are extracted from the graph and replaced by new variables. As a rule, a node pointed to by the largest number of parent edges is chosen first; in case of a tie the one located closer to the bottom is chosen.

The TED in Figure 6(a) exposes several subexpressions for possible extraction: $(x_0 - x_3)$ and $(x_0 + x_3)$, rooted at variable x_0; and $(x_1 - x_2)$, rooted at the rightmost node of variable x_1. The first two expressions are extracted and substituted by new variables $S1 = (x_0 - x_3)$ and $S2 = (x_0 + x_3)$ as they correspond to the nodes located lowest in the TED. Variables $S1, S2$ are then pushed up, below constant nodes, and the TED is reordered to expose new candidates for extraction. The result is shown in Figure 6(b). New candidate expressions, $(x_1 - x_2)$ and $(x_1 + x_2)$, are then extracted and substituted by variables, $S3, S4$, resulting in the TED shown in Figure 6(c). At this point there are no more nontrivial expressions to be extracted, and the algorithm terminates. The details of the dynamic CSE algorithm are described in [11]. As a result, the above TED-based common subexpression elimination results in the following simplified expressions:

$$\begin{aligned} y_0 &= A \cdot (S2 + S4) \\ y1 &= B \cdot S1 + C \cdot S3 \\ y2 &= D \cdot (S2 - S4) \\ y3 &= C \cdot S1 - B \cdot S3 \end{aligned} \quad (4)$$

Considering that A=1, the computation of such optimized expressions requires only 5 multiplications and 8 additions, a significant reduction from the 16 multiplications and 12 additions of the initial expressions. This result is identical to the one that can be obtained using SPIRAL, a specialized system for DSP transform optimization [5].

4 DFG Generation and Optimization

The recursive TED decomposition procedures described in the previous sections produces simplified algebraic expression(s) 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 an expression (or a set of expressions) as *Normal Factor Form* (NFF).

Definition 1 *The factored form expression associated with a given TED is called Normal Factored Form (NFF) if the order of variables in the expression(s) is compatible with the order of variables in the TED.*

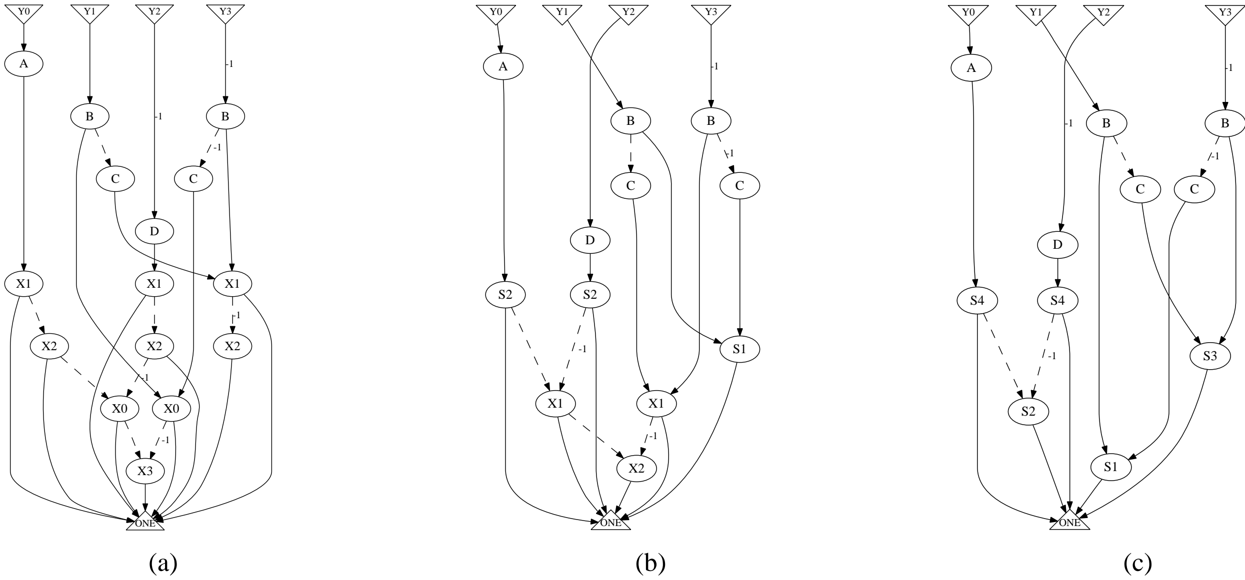

Figure 6. a) Initial TED of DCT2-4; b) TED after extracting $S_1 = (x_0 - x_3)$ and $S_2 = (x_0 + x_3)$; c) Final TED after extracting $S_3 = (x_1 - x_2)$ and $S_4 = (x_1 + x_2)$.

Theorem 1 *Normal Factored Form derived from a linear TED is unique.*

Proof: By construction, each time a TED is decomposed disjunctively (or conjunctively) an arithmetic operation ADD (or MULT) is introduced in the expression, so that $F = F_1 \text{OP} F_2$, where OP is the respective arithmetic operation. Each subexpression, F_1, F_2, is represented by a TED, with variable order compatible with that of the original TED. Let the order of the two subexpressions ($F_1 \text{OP} F_2$ vs $F_2 \text{OP} F_1$) in the expression for F be determined by the position of the top variable of each TED. That is, the expression whose top TED node is placed higher in the original TED will be listed first. This rule is applied recursively to F_1 and F_2, each time ordering a pair of subexpressions according to the position of their top variables. By imposing this rule, the ordering of subexpressions is unique, hence the resulting NFF is unique. QED

The uniqueness of Normal Factored Form is illustrated with the TED in Fig.5(a)-(c) for the variable order $\{c, d, a, b\}$. On the top decomposition level, shown in Fig. 5(b), the TED is split into two non-disjoint subgraphs, $F = F_1 + F_2$, where $F_1 = (c + d)(a + b)$ and $F_2 = a \cdot b$ (variable c is placed above a). At the next decomposition level, subgraph F_1 is split into $F_3 \cdot F_4$, where $F_3 = (c + d)$ and $F_4 = (a + b)$, each with variable order compatible with that of the TED. Similarly, subgraph F_2 is expressed as $F_2 = a \cdot b$, since variable a is placed above b in the TED. The final NFF for this TED is $F = (c + d)(a + b) + ab$. The form is unique, and the order of variables $\{c, d, a, b\}$ is compatible with that of the original TED. This expression contains three additions, corresponding to the three additive edges of the TED, and two multiplications, for the two local dominators in the TEDs for $F_2 = a \cdot b$ and $F_1 = (c + d)(a + b)$. This is the most compact representation for this expression, resulting in a minimum number of operators of each type. In this sense the NFF is also minimal for a given TED.

In summary, the NFF for a given TED depends on the structure of the initial TED and the ordering of its variables. Several variable ordering algorithms have been developed for this purpose, including static ordering and dynamic reordering schemes, similar to those in BDDs. However, the ordering of the TED is driven by the complexity of the resulting NFF and the structure of the generated DFG, rather than by the number of TED nodes. The structure of the DFG can be evaluated by performing a fast ASAP or list scheduling to derive a lower bound on the DFG latency.

Once the algebraic expression represented by a TED has been decomposed, a structural DFG representation of the optimized expression is obtained by replacing the algebraic operations in the resulting NFF with hardware operators. This process is fast and done in the background during the TED decomposition. However, unlike Normal Factored Form, the DFG representation is not unique. While the number of operators remains fixed (dictated by the structure of the ordered TED), the DFG can be further restructured and balanced to minimize its latency. In addition to replacing operator chains by logarithmic trees, standard logic synthesis methods, such as collapsing and re-decomposition, can be applied. Furthermore, multiplications by constants are replaced by shifters and adders to minimize the number of multipliers. Standard techniques are available to perform such a transformation based on Canonical Signed Digit (CSD) representation.

An important feature of the TED decomposition is that it has insight into the final DFG structure. Different DFG solutions can be generated by modifying the TED variable order during the decomposition, followed by a fast generation of the minimum-latency DFG using heuristic scheduling. This approach makes it possible to minimize the hardware resources or latency in the final, *scheduled* implementation.

5 TDS System

The TED decomposition described here was implemented as part of a prototype system, TDS, shown in Fig. 7. The input to the system is the design specified in C/C++ or given in form of a DSP matrix. The left part of the figure shows the traditional high-level synthesis flow, which extracts DFG from the initial specification and performs standard high-level synthesis operations: scheduling, allocation and resource binding. The right part of the figure shows the actual TDS optimization flow. It transforms the data flow graph extracted from the initial specification into an optimized DFG using a host of TED-based decomposition and DFG optimization techniques, and passes the modified DFG to an HLS tool for synthesis. Currently, an academic synthesis tool, GAUT [12], is used for front-end parsing and for the final high level synthesis, but any of the existing HLS tools can be used for this purpose (provided that compatible interfaces are available).

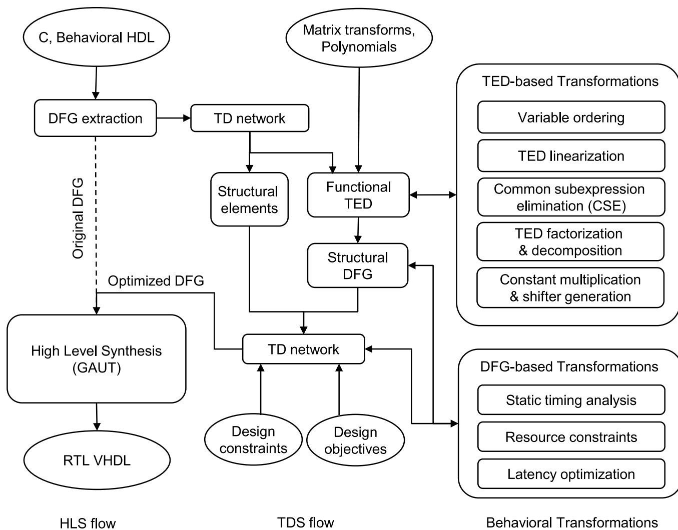

Figure 7. TDS system flow.

The DFG extracted from the initial specification is translated into a hybrid network composed of functional blocks represented with TEDs, and other operators that cannot be expressed as TEDs, treated as black boxes. TDS optimizes the resulting hybrid network and transforms it into a final DFG using TED- and DFG-related optimizations. The entire DFG network is finally balanced to minimize the latency. The system provides a set of interactive commands and optimization scripts that include: variable ordering, TED linearization, static and dynamic factorization, decomposition, replacement of constant multiplications by shifters, DFG construction, balancing, etc. The quality of the decomposition is controlled by choosing initial TED ordering and reordering of component TEDs.

6 Experimental Results

The system was tested on a number of practical designs from computer graphics applications and digital filters, with the main goal to minimize the DFG latency. Table 1 compares the implementation of a *quintic spline* filter using: 1) the original design written in C; 2) the design produced by a *kernel-based decomposition* system (KBD) of [4]; and 3) the design produced by TDS. All DFGs were synthesized using GAUT. The top row of the table reports the number of arithmetic operations (adders, multipliers, shifters, subtractors) in the unscheduled DFG. The remaining rows give the number of resources used for a given latency in an *scheduled DFG*; and the datapath area using GAUT. Minimum latency for each solution are shown in bold. The results for circuits that cannot be synthesized for a given latency are marked with '–' (over-constrained).

Design		Original design		KBD solution		TDS solution	
	Latency (ns)	$+,\times,\ll,-$	Area	$+,\times,\ll,-$	Area	$+,\times,\ll,-$	Area
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

Table 1. Latency and area for Quintic Spline design produced by GAUT for different DFGs.

The latency of DFG obtained by TDS is 110 ns, 21.4% smaller than that of KBD of 140 ns. While the KBD solution has the smallest number of operations in the *unscheduled DFG*, the synthesized design requires more resources for the minimum latency of 140 ns, obtained by KBD. Specifically, the design area of the TDS implementation is 22% smaller than that of the KBD solution. Similar behavior was observed for all the tested designs.

Design	TDS vs Original		TDS vs KBD	
	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 2. Percentage improvement of TDS vs Original and KBD on achievable latency, and area at the minimum achievable latency.

Table 2 summarizes the implementation results for these benchmarks. The implementations obtained by TDS have latency on average 15.5% smaller than that of KBD, and 27.2% smaller than the original DFGs. The area results reported in the table are given for the *reference latency*, defined as the minimum latency obtained by the two methods under comparison. The hardware area of the TDS solutions for the reference latency is on average 7.6% smaller than that of KBD, and 36.3% smaller than the original design, without any DFG modification. These are significant improvements. The reason that the DFG of *Quartic* design produced by TDS requires more area than KBD can be explained by the fact that

the main objective of the decomposition was to minimize the DFG latency. In particular, the minimum latency obtained by TDS for this design was 110 ns, compared to 130 ns obtained by KBD. The TDS solution is characterized by more parallelism, which for the reference latency of 130 ns required more resources than the KBD solution.

7 Conclusions

A new approach for transforming the initial specification of data flow designs, based on TED decomposition, has been implemented. The TED-based optimization system guides the algebraic decomposition to obtain solutions with minimum latency and/or minimum cost of resources in a scheduled DFG. The resulting designs are better than those obtained by a straightforward minimization of the number of arithmetic operations in the algebraic expression.

While currently the TDS system is integrated with an academic high-level synthesis tool, GAUT, we believe that it could be successfully used as a pre-compilation step in a commercial synthesis software, such as Catapult C. In this case, the optimized DFG can be translated back into C (while preserving the optimized structure) and used, instead of the original DFG, as input to HLS. Finally, the TED model implicitly assumes infinite precision arithmetic. The issue of finite precision of the operators should be addressed to make the system applicable to real DSP applications.

Acknowledgments: This work was supported in part by the National Science Foundation, award No. CCF-0702506, and in part by CNRS, contract PICS 3053.

References

[1] P. Coussy and A. Morawiec, *High Level Synthesis from Algorithm to Digital Circuits*, Springer, Aug 2008.

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

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

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

[5] 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", *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[6] 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.

[7] "TDS - TED-based Behavioral Decomposition System.", http://www.ecs.umass.edu/ece/labs/vlsicad/tds.html.

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

[9] 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.

[10] D. Gomez-Prado, M. Ciesielski, Q. Ren, J. Guillot, and E. Boutillon, "Optimization Data Flow Graphs to Minimize Hardware Implementations", in *Design Automation and Test in Europe, Workshop, DATE 09*, pp. 117-122, April 2009.

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

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