High-level Transformations using Canonical Dataflow Representation

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

1 Abstract

This paper describes a systematic method and an experimental software system for high-level transformations of designs specified at behavioral level. The goal is to transform the initial design specifications into an optimized data flow graph (DFG) better suited for high-level synthesis. The optimizing transformations are based on a canonical Taylor Expansion Diagram (TED) representation, followed by structural transformations of the resulting DFG network. The system is intended for data-flow and computation-intensive designs used in computer graphics and digital signal processing applications.

2 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 Hardware Description Languages (HDL) and C or C++ language. 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, precision, etc.). The algorithmic description used as input to high-level synthesis does not require explicit timing information for all operations of the algorithm 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 algorithmic solutions. An important productivity leap is thus achieved.

However, the optimizations offered by the high-level synthesis tools are limited to algorithms for scheduling and resource allocation performed on a fixed Data Flow Graph (DFG), derived directly from the initial HDL specification [1]. Modification of the DFG, if any, is provided by rewriting the initial specification. In this sense the high-level synthesis flow remains "classical": the algorithm is first defined and validated without any hardware constraints; a bit-accurate model is then derived to obtain an initial hardware specification of the design, which becomes input to the HLS flow. With this approach the quality of the final hardware implementation strongly depends on the quality of the hand-written hardware specification. In order to explore other solutions, the user needs to rewrite the original specification, from which another DFG is derived and synthesized.

Why not then relax the process and start the flow at the Algorithm level, where the design is given as an abstract specification, sufficient to generate the required architecture but without the detailed timing and hardware information. While this may not be possible for all the designs (in particular control applications), data-intensive applications can benefit from this approach. For example, in signal processing applications that deal with noisy signals there may be several ways to perform the computation described by the algorithm. Some of them may lead to an acceptable hardware solution even if it introduces a moderate level of internal computation noise (SNR). In general, such a noise will not affect the performance of the system in a significant way, while the resulting architecture may give a better hardware implementation in terms of circuit area, latency, or power.

To give a simple example, in fixed precision computation, the expression $A \cdot B + A \cdot C$ is not strictly equal to $A(B + C)$ in terms of signal-to-noise ratio. Nonlinear operations of rounding, truncation and saturation, required to keep the internal precision fixed, are not applied in the two expressions in the same order; as a result, the two computations may differ slightly. Nevertheless, in a common signal processing application, the two expressions can be considered identical from the computational view point. The one with a better hardware cost can be selected for final hardware implementation. In this example, the expression $A(B + C)$ may be chosen as it needs to schedule fewer operators, thus resulting in smaller latency and/or circuit area.

In this context, the road to automatic transformation of the design specification that preserves its intended "requirement" is open. Such a *specification transformation* tool should allow the designer to express the specification rapidly and to rewrite it into a form that will optimize the final hardware implementation. Such a modification must take into consideration the specific design flow and the constraints of the application.

Automatic specification transformation is an old concept. In fact, software compilers commonly use such optimization techniques as dead code elimination, constant propagation, common subexpression elimination, and others [2]. Some of those compilation techniques are also used by HLS tools. Several high-level synthesis systems, such as Cyber [3] and Spark [4], use different methods for code optimization (kernel-based algebraic factorization, branch balancing, speculative code motion methods, dead code elimination, etc.) but without guaranteeing optimality of the high-level transformations. For example, very few of them, if any, are

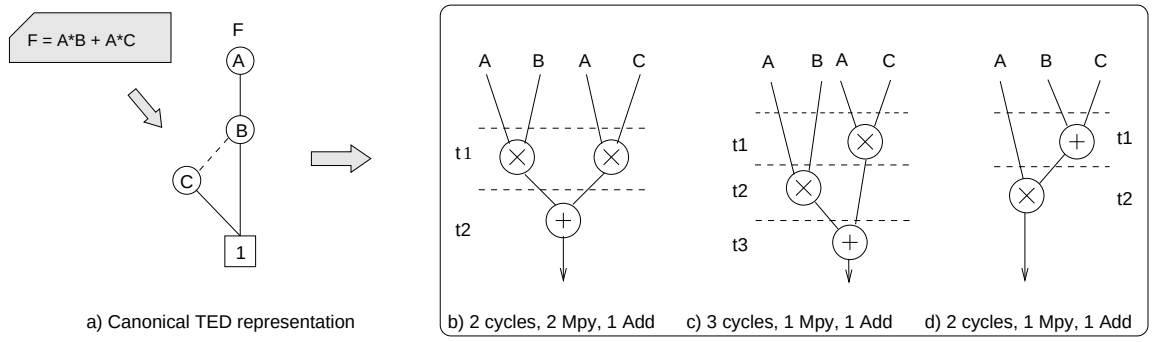


Figure 1. High-level transformations: a) Canonical TED representation; b),c,) DFGs corresponding to the original expression $F = A \cdot B + A \cdot C$; d) DFG for the transformed expression, $F = A \cdot (B + C)$.

able to recognize that the expression $X = (a+b)(c+d)-a \cdot c-a \cdot d-b \cdot c-b \cdot d$ trivially reduces to $X = 0$. With the exception for a few specialized systems for DSP code generation, such as SPIRAL [5], these methods rely on simple manipulations of algebraic expressions based on term rewriting and basic algebraic properties (associativity, commutativity, and distributivity) that do not guarantee optimality.

This paper describes a systematic method for transforming an initial design specification into an optimized DFG, better suited for high-level synthesis. 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 the best hardware implementation in terms of latency and/or hardware cost.

To motivate the concept of high-level transformations supported by the canonical TED representation, consider a simple computation, $F = A \cdot B + A \cdot C$, where variables A, B, C are word-level signals. Figure 1(a) shows the canonical TED representation encoding this expression (discussed in the next section). Figures 1(b) and (c) show two possible scheduled DFGs that can be obtained for this expression using any of the standard HLS tools. We should emphasize that both solutions are obtained from a *fixed DFG*, derived directly form the original expression. They have the same structure and differ only in the scheduling of the DFG operations. The DFG in Figure 1(b) minimizes the design latency and requires one adder and two multipliers, while the one in Figure 1(c) reduces the number of assigned multipliers to one, at a cost of the increased latency.

Figure 1(d) shows a solution that can be obtained by transforming the original specification $F = A \cdot B + A \cdot C$ into $F = A \cdot (B + C)$, which corresponds to a *different DFG*. This DFG requires only one adder and one multiplier and can be scheduled in two control steps, as shown in the figure. This 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 remainder of the paper explains how such a transformation and the optimization of the corresponding DFG can be obtained using the canonical TED representation. These optimizing transformations are implemented in the software system, TDS, intended for data-flow and computation-intensive designs used in computer graphics and digital signal processing applications. TDS system is available on line [7].

3 Taylor Expansion Diagrams (TED)

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 [6]. It is particularly suitable for algorithm-oriented applications, such as signal and image processing, with computations modeled as polynomial expressions.

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:

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

where $f'(x = 0),\ f''(x = 0)$, etc, are the successive derivatives of f w.r.t. x, evaluated at $x = 0$. The individual terms of the expression, $f(0), f'(0), f"(0)$, etc., are then decomposed iteratively with respect to the remaining variables on which they depend (y, .., 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 decomposing 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, $f(0), f'(0), f"(0)$, etc. Each edge is labeled with the weight, representing the coefficient of the respective term in the expression.

Most of the TEDs presented in this work are *linear TEDs*, representing linear multi-variate polynomials and containing only two types of edges: *multiplicative* (or *linear*) edges, represented in the TED as solid lines; and *additive* edges, represented as dotted edges. Nonlinear expressions can be triv-

ially converted into linear ones, by transforming each occurrence of a nonlinear term x^k into a product $x_1 \cdots x_k$, where $x_i = x_j$. Such a transformed expression is then represented as a linear TED.

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 computed as a product of the edge expressions, each such an expression being a product of the variable in its respective power and the edge weight. Only non-trivial terms, corresponding to edges with non-zero weights, are stored in the graph.

As an example, consider an expression $F = A \cdot B + A \cdot C$ represented by the TED in Figure 1(a). This expression is computed in the graph as a sum of two paths from TED root to terminal node 1: $A \cdot B$ and $A \cdot B^0 \cdot C = A \cdot C$. In fact, TED encodes such an expression in *factored form*, $F = A \cdot (B + C)$, 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 is an important feature of the TED representation, employed by TED-based factorization and common subexpression extraction described in the remainder of the paper.

In summary, TED represents finite multi-variate polynomials and maps word-level inputs into word-level outputs. TED is reduced and normalized in a similar way as BDDs [8] and BMDs [9]. Finally, the reduced, normalized and ordered TED is canonical for a given variable order. Detailed description of the TED representation and its application to verification can be found in [6].

4 TED-based Decomposition

The principal goal of algebraic factorization and decomposition is to minimize the number of arithmetic operations (additions and multiplications) in the expression. A simple example of *factorization* is the transformation of the expression $F = AB + AC$ into $F = A(B + C)$, referred to in Figure 1, 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 mentioned earlier, TED already encodes the expression in a compact, factored form. The goal of TED decomposition is to find a factored form that will produce a DFG with minimum hardware cost of the final, scheduled implementation. This is in contrast to a straightforward minimization of the number of operations in an unscheduled DFG, that has been the subject of all the known previous approaches [10, 11].

This section describes two methods for TED decomposition. One is based on the factorization and common subexpression extraction performed on a TED with a given variable order, without modifying that order. This method is applicable to generic expressions, without any particular structure. The other method is a dynamic CSE, where common subexpressions are derived by dynamically modifying TED variable order in a systematic way. This method is particularly suitable for well-structured DSP transforms, such as DCT, DFT, WHT, etc, where they can discover common computing patterns, such as butterfly.

4.1 Static TED Decomposition

The static TED decomposition approach extends the original cut-based decomposition method of Askar [10]. Basic idea of the cut-based decomposition is to identify in the TED a set of *cuts*, i.e., additive edges and multiplicative nodes (called dominators) whose removal separates the graph into two disjoint subgraphs, Each time an additive or multiplicative cut is applied to a TED, a hardware operator (ADD or MULT) is introduced in the DFG to perform the required operation on the two subexpressions. This way, a *functional* TED representation, is eventually transformed into a *structural* data flow graph (DFG) representation. It has been shown that different cut sequences generate different DFGs, from which the DFG with best property (typically latency) can be chosen.

By construction, the cut-based decomposition method is limited to a disjoint decomposition. Many TEDs, however, such as the one shown in Figure 2(a), do not have a disjoint decomposition property and must be handled differently.

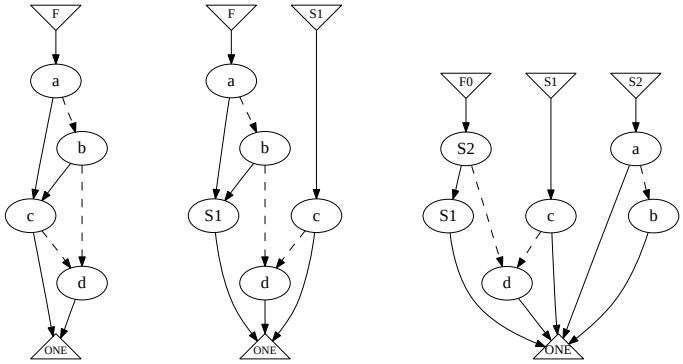
The decomposition described here applies to an arbitrary TED graph (linearized, if necessary), with both disjoint and non-disjoint decomposition. The TED decomposition is applied in a bottom-up fashion by iteratively extracting common terms (sums and products of variables) and replacing them with new variables. The method is based on a series of functional transformations that decompose the TED graph into a set of irreducible TEDs, from which a final DFG representation is constructed. The decomposition is guided by the quality of the resulting scheduled DFG (measured in terms of its latency or resource utilization) and not by the number of operators in an unscheduled DFG.

The basic procedure of the TED decomposition is the *sub* operation, which extracts a subexpression *expr* from the TED and substitutes it with a new variable *var*. First, the variables in the expression *expr* are pushed to the bottom of the TED, respecting the relative order of variables in the expression. Let the top-most variable in *expr* be v. Assuming that *expr* is contained in the original TED, this expression will appear in the reordered TED as a subgraph rooted at node v. The extraction of *expr* is accomplished by removing the subgraph rooted at v and connecting the reference edge(s) to terminal node 1.

The extraction operation is shown in Figure 2(a,b), where subexpression $expr = (c + d)$ is extracted from $F = (a + b)(c + d) + d$. If an internal portion of the extracted subexpression 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. This is also visible in Figure 2(b), with node d being duplicated in the process.

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 top-level level graph. The final result of the decomposition is a series of *irreducible* TEDs, related hierarchically. Specifically, the decomposing algorithm identifies and extracts sum-terms and product terms in the TED and substitutes them by new variables, using the sub operation described above. Computational complexity of the extraction algorithms is polynomial in the number of TED nodes. Each new term constitutes an *irreducible TED* graph, which is then translated directly into a DFG composed of the operators of one type (adders for sum-term, and multipliers for product term).

The TED decomposition and DFG construction is illustrated with a simple example in Figure 2. This TED does not have a single additive cut edge that would separate the graph disjunctively into two disjoint subgraphs; neither does it have a dominator node that would decompose it conjunctively into disjoint subgraphs, and hence cannot be decomposed using cut-based method. The decomposition starts with identifying and extracting expression $S_1 = c+d$, followed by extracting $S_2 = a + b$, represented as an irreducible TED. Note that term d is automatically duplicated in this procedure.



(a) (b) (c)

Figure 2. Static TED decomposition; (a) Original TED for expression $F_0 = ac+bc+ad+bd$: (b) TED after extracting $S_1 = c + d$; (c) Final TED after extracting $S_2 = a + b$, resulting in Normal Factored Form $F_0 = (a+b)\cdot(c+d)+d$.

Product terms can be identified in the TED as a set of nodes connected by multiplicative edges, such that the intermediate nodes in the series do not have any additive incoming or outgoing edges. Only the starting and ending nodes can have incident additive edges. In Fig. 2(a) no product terms can be found at this decomposition level. A sum term appears in the TED graph as a set of variables, connected by multiplicative edges to a common node and linked together by additive edges. In Fig. 2(a) two such sum-terms can be identified and extracted as reducible TEDs: $S_1 = c + d$ and $S_2 = a + b$. Each irreducible subgraph is then replaced by a single node in the original TED to produce TED shown in Fig. 2(c). This procedure is repeated iteratively until the TED is reduced to the simplest, irreducible form. The resulting TED is then subjected to the final decomposition using the fundamental Taylor expansion procedure. The graph is traversed in a topological order, starting at the root node. At each visited node v the expression $F(v)$ is computed as $F(v) = F_0 + v \cdot F_1$, where F_0 is the function rooted at the first node reached from v by an additive edge, and F_1 is the function rooted at the first node reached from v by a multiplicative edge. Using this procedure, the TED in Figure 2(c) produces the decomposed expression $F = S_2 \cdot S_1 + d$, where $S_1 = c + d$ and $S_2 = a + b$.

4.2 Dynamic TED Factorization

An alternative approach to TED decomposition is based on the 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

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

and computed by the following algorithm:

```
for (j = 0; j < N ; j++)
 { tmp = 0;
   for (k = 0; k< N ; k++)
     tmp+=x[k]*cos(pi*j*(k+0.5)/N);
 y[j] = tmp; }
```

It can be represented in a 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, eq. (2).

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

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. The matrix M for the DCT example has four distinct coefficients, A, B, C, and D (for simplicity, we neglect the fact that $A = cos(0) = 1$). This representation makes it possible to factorize the expressions and subsequently reduce the number of operations to 6 multiplications and 8 additions, as shown by the equations (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 3(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 common subexpression extraction (CSE) for this class of DSP design is to represent the coefficients of the matrix 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 3(a).

The candidate expressions for factorization in such a TED are obtained by identifying the nodes with multiple parent (reference) edges. The subexpression rooted at such nodes are extracted from the graph and replaced by new variables.

The TED in Figure 3(a) exposes two subexpressions for possible extraction: 1) the rightmost node associated with variable x_0 (show in red), the root of subexpression $(x_0 - x_3)$; and 2) the rightmost node associated with variable x_1 (pointed to by nodes C, B), which is the root of subexpression $(x_1 - x_2)$. The first expressions is extracted from the graph and substituted with a new variable, $S1 = (x_0 - x_3)$. Variable $S1$ is then pushed to the top of the diagram, below constant nodes, as shown in Figure 3(b). This new structure exposes another expression to be extracted, namely $S2 = (x_0 + x_3)$. Once the subexpression is extracted, variable $S2$ is also pushed up. The next iterations of the algorithm leads to substitutions $S3 = (x_1 - x_2)$ and $S4 = (x_1 + x_2)$, resulting in a final TED shown in Figure 3(c). At this point there are no more original variables that can be pushed to the top, and the algorithm terminates. As a result, the above TED-based common subexpression elimination results in the following expressions:

$$y_0 = A \cdot (S2 + S4), \;\; y1 = B \cdot S1 + C \cdot S3$$

$$y2 = D \cdot (S2 - S4), \;\; y3 = C \cdot S1 - B \cdot S3,$$

where:
$S1 = (x0 - x3), \; S2 = (x0 + x3), \; S3 = (x1 - x2)$ and $S4 = (x1 + x2)$.

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.

5 DFG Generation and Optimization

The TED decomposition procedures described in the previous section produce simplified algebraic expressions in factored form. Each addition operation in the expression corresponds to an additive edge of some irreducible TED, and each multiplication corresponds to a multiplicative edge in an irreducible TED, obtained from the TED decomposition. We refer to such a form as *Normal Factor Form* (NFF) of the TED.

It can be shown that normal factored form is minimal and unique for a given TED with fixed variable ordering. The form is *minimal* in the sense that it requires minimum number of operators of each type (adders and multipliers) to describe the algebraic expression encoded in the TED. No other expression that can be derived from this TED (with the given variable order) can have fewer operations. For example, the NFF for the TED in Fig. 1 is $F = A \cdot (B + C)$, with one ADD and one MULT operator. Other forms, such as $AB + AC$, have two multiply operators, which are not present in this graph (the multiplicative edges leading to the terminal node 1 represent trivial multiplications by 1 and do not count). Such a form is also *unique*, if the order of variables in the normal factored form is compatible with that in the TED. In the above example, the forms $A(C + B)$ or $(B + C)A$ are not NFFs, since the variable order in those expressions is not compatible with that in the TED in Fig. 1(a).

The concept of normal factored form can be further clarified with the TED in Fig. 2. The normal factored form for this TED is $F_0 = (a + b) \cdot (c + d) + d$. It contains three adders, corresponding to the three additive edges in the irreducible TEDs, S_1, S_2, and the top-level TED, F_0, shown in Fig. 2(c); and one multiplier corresponding to the multiplicative edge in the TED for F_0, in Fig. 2(c). This is the minimum number of operators that can be obtained for the TED with the variable order $\{a, b, c, d\}$. The form is unique, since the ordering of variables in each term is compatible with the ordering of variables in the TED.

It should be obvious from the above discussion that the NFF for a given TED depends only on the structure of the initial TED and the ordering of its variables. Hence, a TED variable ordering plays a central role in deriving decompositions that will lead to efficient hardware implementations. Several variable ordering algorithms have been developed for this purpose, including static ordering and dynamic re-ordering schemes, similar to those in BDDs. However, TED ordering is driven by the complexity of the NFF and the structure of the resulting DFGs, rather than by the number of TED nodes.

5.1 Data Flow Graph Generation

Once the algebraic expression represented by TED has been decomposed, a structural DFG representation of the optimized expression is obtained by replacing the algebraic operations in the normal factored form into hardware operators of the DFG. However, unlike Normal Factored Form, the DFG representation is not unique. While the number of operators remains fixed (dictated by 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, taking into consideration signal arrival times can be used for this purpose [1].

An important feature of the TED decomposition, concluded by the generation of an optimized DFG, is that it has insight into the final DFG structure. Different DFG solutions can be generated by modifying the TED variable order, performing static and dynamic factorization, followed by a fast generation of the minimum-latency DFG. This approach makes it possible to minimize the hardware resources or latency in the final, *scheduled* implementation, not just

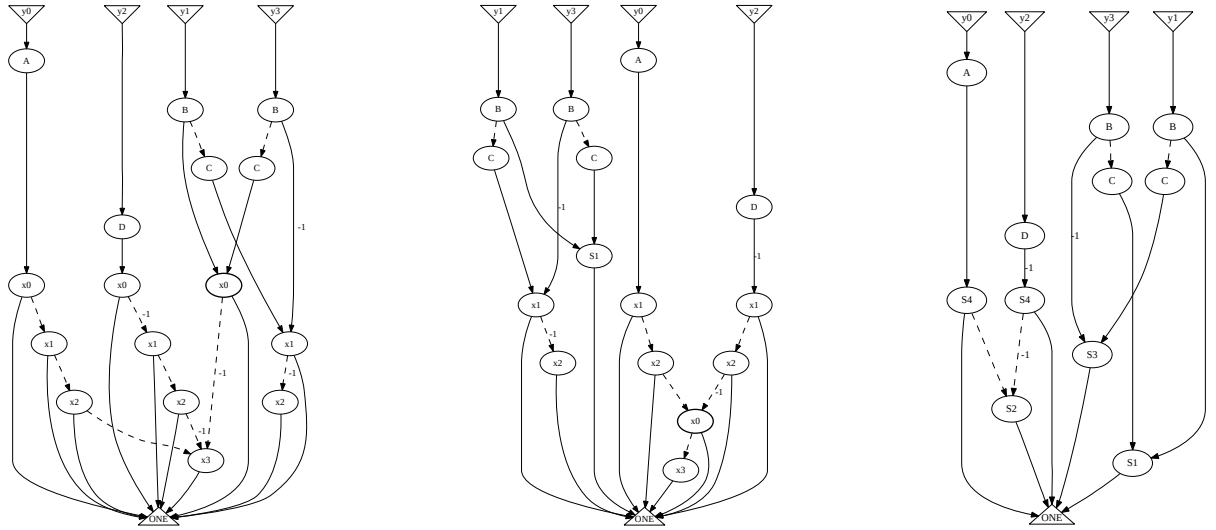


Figure 3. a) Initial TED of DCT2-4, b) TED after extracting $S_1 = (x_0 - x_3)$; c) final TED after extracting $S_2 = (x_0 + x_3)$, $S_3 = (x_1 - x_2)$ and $S_4 = (x_1 + x_2)$, resulting in the final factored form of the transform.

the number of operations in the DFG graph. The solution that meets the required objective is selected.

In summary, TED variable ordering, static and dynamic factorization/CSE, and DFG restructuring, are at the core of the optimization techniques employed by TED decomposition.

5.2 Replacing 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 or subtractions. Standard techniques are available to perform such a transformation based on Canonical Signed Digit (CSD) representation. However, these methods do not address common subexpression elimination or factorization involving shifters. In this section we present a systematic way to transform integer multiplications into shifters using the TED structure. This is done by introducing a special '*left shift*' variable L into the TED, while maintaining its canonicity. The modified TED can then be decomposed using methods described earlier.

First, each integer constant C is represented in the CSD format as $C = \sum_i (k_i \cdot 2^i)$, where $k_i \in (-1, 0, 1)$. By introducing a new variable L to replace constant 2, C can be represented as $\sum_i (k_i \cdot L^i)$. The term L^i in this expression represents left shift by i bits. The TED with the shift variables is then subjected to a regular TED decomposition. Finally, in the DFG generated by the TED decomposition the terms involving shift variables, L^k, are replaced by k-bit shifters.

The example in Figure 4 illustrates this procedure for the expression $F = 7a + 6b$. The original TED is shown in Figure 4(a) and the corresponding DFG with constant multipliers in Figure 4(b). The original expression is transformed into an expression with the shift variable L: $F = (L^3 - 1) \cdot a + (L^3 - L^1) \cdot b = L^3 \cdot (a + b) - (a + L \cdot b)$, and represented by a non-linear TED shown in Figure 4(c). Each edge of the TED is labeled with a pair $(\hat{}p, w)$, where $\hat{}p$ represents the power of variable (stored as the node label), and w represents the edge weight (multiplicative constant) associated with this term. For example, the edge labeled $(\hat{}3, 1)$ coming out of variable L represents a non-linear term $L^3 \cdot 1$. The TED subgraph rooted at the right node, labeled a, represents expression $a + b$. The dotted between nodes a and b, labeled $(\hat{}0, 1)$, simply represents an addition.

The modified TED is then transformed into a DFG, where multiplications with inputs L^k are replaced by k-bit shifters, as shown in Figure 4(d). The optimized expression corresponding to this DFG is $F = ((a + b) << 2 - b) << 1 - a$, 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 needed to implement the original expression.

6 TDS System

The TED decomposition described here was implemented as part of a prototype system, TDS, shown in Fig. 5. The input to the system is the design specified in C/C++, behavioral HDL, or given in form or a DSP matrix. The system can also internally generate matrices for some of the standard linear DSP transforms, such as DCT, DFT, WHT, etc., to be used as input. The left part of the figure shows the traditional high-level synthesis flow, which extracts the control data flow (CDFG) from the initial specification and performs standard high-level synthesis operations: scheduling, alloca-

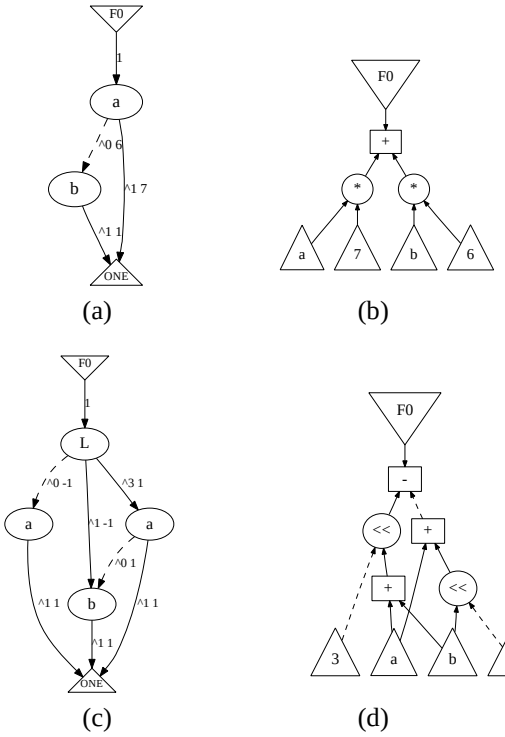


Figure 4. Replacing constant multiplications by shift operations: (a) Original TED for $F_0 = 7a + 6b$; (b) Initial DFG with constant multipliers; (c) TED after introducing *shift* variable L; (d) Final DFG with shifters, corresponding to the expression $F = ((a + b) << 2 - b) << 1 - a$.

tion 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 a high-level synthesis tool. 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 format interfaces are available).

The DFG extracted from the initial specification is translated into a hybrid network composed of islands of functional blocks, represented using TEDs, and other operators ("structural" elements). Specifically, TEDs are constructed from polynomial expressions that have finite Taylor expansion, describing arithmetic components of the design (adders, multipliers, multiplexers) as well as "if-then-else" statements. Those operators that cannot be represented as functional TEDs (such as comparators, saturators, etc.), are considered as black boxes in the hybrid network.

TDS optimizes the resulting hybrid TED/DFG network and transforms it into a final DFG using TED- and DFG-related optimizations. The entire DFG network is finally restructured 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

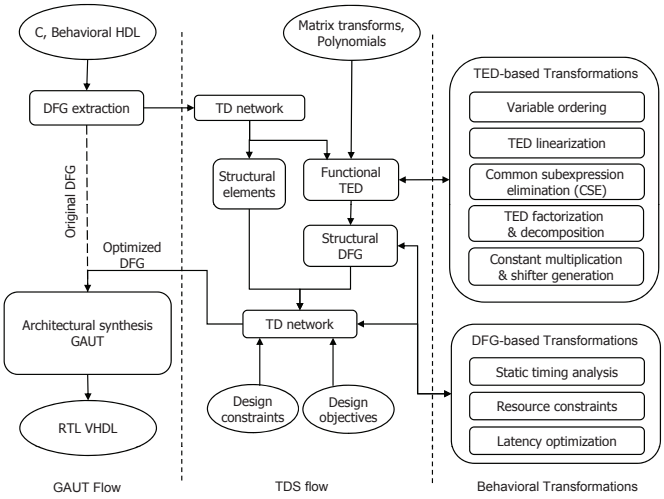


Figure 5. TDS system flow.

multiplications by shifters, DFG construction, balancing, etc.

7 Experimental Results

The system was tested on a number of practical designs from computer graphics applications, filters and DSP designs. Table 1 compares the implementation of a *quintic spline* filter using: 1) the original design written in C; 2) the design produced by CSE decomposition system of [11]; 3) and 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) for unscheduled DFG. The remaining rows list the actual number of resources used for a given latency in the *scheduled DFG*; and the implementation area using GAUT (datapath only). 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		CSE 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. Quintic Spline achivable latency and area for different designs. The area reported is for GAUT.

As seen in the table the CSE solution has the smallest number of operations in the *unscheduled DFG*, and the latency of 140 ns. The latency of DFG obtained by TDS was 110 ns, i.e., 21.4% faster, even though it had more DFG operations. And for the minimum latency of 140 ns, obtained by CSE, TDS produced circuit implementation with area 22% smaller than CSE. Similar behavior has been observed for

all the tested designs. In all cases the latency of DFGs produced by TDS was smaller; and (with an exception for Quartic Spline design), all of them also had smaller hardware area for the minimum latency produced by CSE.

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 2. Percentage improvement of TDS vs Original and CSE on achievable latency; and area at the minimum achievable latency.

Table 2 summarizes the implementation results for these benchmarks. We can see that the implementations obtained by TDS have latency that is on average 15.5% smaller than that of CSE, and 27.2% smaller than the original DFGs. The hardware area of the TDS solutions for the reference latency (defined as the minimum latency obtained by the other two methods), is on average 7.6% smaller than that of CSE, and 36.3% smaller than the original design, without any DFG modification. These are significant improvements.

8 Conclusions

A new approach to the optimization of initial specification of data flow designs have been implemented that yields DFGs better suited for high-level synthesis and produces better hardware implementations than currently available. The TED-based optimization system guides the algebraic decomposition to obtain solutions with minimum latency and/or with minimum cost of operators in a scheduled DFG. The optimized DFGs produce designs with lower latency than those obtained by a straightforward minimization of the number of arithmetic operations in the expression; and for the minimum latency obtained by the other methods, the generated designs require on average smaller hardware area.

The TDS system, developed as part of this research, goes beyond simple algebraic decomposition. It allows to handle arbitrary data-flow designs and algorithms written in C/C++ to optimized their data flow graphs prior to high-level synthesis. 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 DFGs can be translated back into C (while preserving the optimized structure), and used as input to HLS, instead of the original DFGs. Finally, to turn TDS into a commercial-quality system, the issue of finite precision of the operators needs to be addressed. This is a subject of an ongoing work.

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

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

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

[4] 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.

[5] 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.

[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] M. Ciesielski, D. Gomez-Prado, and Q. Ren, "TDS - Taylor Decomposition System.", http://incascout.ecs.umass.edu/tds.

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

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

[10] 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.

[11] 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.

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