Abstract
This paper describes an efficient graph-based method to optimize algebraic expressions and, specifically, to minimize the number of arithmetic operations, such as additions and multiplication. The method is based on factorization, common subexpression elimination and decomposition of algebraic expressions performed on a canonical, graph-based representation, called Taylor Expansion Diagram (TED). The method is generic, applicable to arbitrary algebraic expressions, and does not require any knowledge of the application domain. TED-based decomposition is illustrated with examples from signal processing domain, such as DSP transforms, and different linear and non-linear filters. Experimental results show that such an optimization compares favorably with dedicated systems for DSP code generation and filter optimization. As such, it is an attractive alternative for algebraic optimization.

1 Introduction
In order to address the problem of ever-increasing size and complexity of HW/SW applications and to improve computational speed of the design, designers have been striving to raise the level of design abstraction. System level tools and HDL languages have been introduced to help designer in this effort, increasing the design efficiency and reducing the design development time. Nevertheless, it is often necessary to hand-tune the specification code at a lower level due to a lack of efficient compilers and inefficiency of high-level synthesis tools. Such a hand-tuning is extremely
time consuming and there is no guarantee of success due to the high complexity of advanced designs. Examples of such designs include digital signal processing, telecommunications, embedded and multi-media applications. The computationally intensive tasks in these applications are performed by discrete signal transforms, such as the discrete Fourier transform (DFT), discrete cosine transform (DCT), discrete wavelet transform (DWT), Walsh-Hadamard transform (WHT), etc. In these application the designer must possess very good skills in electronics, logic design, computer systems, as well as in signal processing. In order to eliminate redundancy in the code or on a chip, hand tuning requires expertise in all of these domains and demands serious time investment.

The main difficulty in effectively solving this problem is the lack of reliable program optimization tools that could optimize the initial specification code for a particular application, prior to architectural (high-level) synthesis. The quality of results obtained by compilers and synthesis tools is strongly affected by the structure of the code, whether it is written in HDL, or C/C++, etc. Code optimization, such as common subexpression elimination (CSE), dead code elimination, etc., performed by compilers, is done on the syntactic and lexical levels, which is not very efficient as it depends on factors such as the loop unrolling degree, etc. Classical optimization techniques, such as scheduling and resource allocation, employed by high-level synthesis tools, tend to improve results somewhat [1] but do not address all the problems. These tools rely on a representation that is derived by a direct translation of the original design specifications, leaving the optimization of that specification to the designer. As a result, the scope of the ensuing architectural optimization is seriously limited.

Several techniques have been proposed to optimize the initial specification prior to high level synthesis [2, 3, 4, 5, 6, 7, 8]. These techniques rely on the application of basic algebraic properties, such as associativity, commutativity, and distributivity, but they are applied largely in an ad-hoc manner. Similar methods have also been used by optimizing compilers [9] and in logic synthesis [10]. With the exception of pipelining and retiming, which have been researched thoroughly, these techniques do not offer a systematic way to optimize the initial design specification or to derive optimum data flow graphs.

**Contribution:** This paper introduces a systematic method to perform optimization of the initial specification based on a canonical, graph-based representation, called Taylor Expansion Diagram (TED) [11]. TED serves as an efficient data structure to perform factorization, common subexpression elimination (CSE) and decomposition of algebraic expressions derived from the design specification. The goal of such an optimization is to minimize the cost of arithmetic operators (adders and multipliers) in that specification, prior to the ensuing architectural synthesis which maps it into hardware. The better the minimized expressions in terms of the cost of their arithmetic operators, the lower the hardware cost of the final implementation. The proposed method is generic, applicable to arbitrary algebraic expressions and does not require any knowledge of the application domain. The method is illustrated with examples taken from signal processing domain: DSP transforms represented as linear matrices derived from Matlab or other similar tools; and different filters represented as linear and non-linear polynomials.

## 2 Previous Work

Research in optimization of the initial design specifications for hardware designs is relatively scarce. The works fall in several categories: optimizing HDL compilers, symbolic methods,
generic kernel-based factorization, and domain specific optimization systems.

2.1 HDL Compilers

Several attempts have been made to provide optimizing transformations in high-level synthesis [3, 4, 5, 7, 12, 8, 13]. Behavioral transformations have been also used in other areas, such as optimizing compilers [9] and logic synthesis [10]. These methods rely on simple manipulations of algebraic expressions based on term rewriting and algebraic properties of associativity, commutativity, and distributivity. Several high-level synthesis systems, such as Cyber [14], Spark [15] and others, use a host of methods for code optimization (such as kernel-based algebraic factorization, branch balancing, speculative code motion methods, dead code elimination, etc.) but without relying on any canonical representation that would guarantee local optimality of the transformations.

One of the most advanced work in this field is that of Gupta et. al, who developed a C-to-VHDL synthesis framework, SPARK [16, 13]. It synthesizes behavioral description using parallelizing transformations in order to reduce the number of operations and latency. They include common subexpression elimination, dead code elimination, speculative and loop-invariant code motions, loop pipelining, and scheduling. It applies those techniques by moving operations across conditionals and loops based on time criticality of an operation, and in the process exposes parallelism available in the algorithm. However, the details of how these techniques work are not described in sufficient details. It seems that the application of particular transformation techniques (mainly speculative code motion and common subexpression elimination) used in the system is not systematic. For example, the algorithms is based on simple branch balancing techniques that “insert new scheduling steps dynamically in the shorter of the two branches of a conditional block without increasing the longest path through the conditional” [16, 13]. In addition, SPARK fails to identify functionally equivalent expressions; the adopted common subexpression elimination technique requires equivalent form of expressions, which means that the expressions need to be represented in the exact same way, which is seldom the case. For example, consider a design with three inputs $A$, $B$ and $C$, and two outputs: $F_1 = A \cdot B + B \cdot C$ and $F_2 = A \cdot (B + C)$. SPARK cannot determine that $F_1 = A \cdot B + B \cdot C$ and $F_2 = A \cdot (B + C)$ are functionally equivalent, and implements them separately instead of computing the simpler function $F_2$ and then assigning $F_1 = F_2$.

Finally, the SPARK methodology is particularly targeted to control-intensive microprocessor functional blocks and multimedia applications. In contrast, this work addresses mostly data intensive designs such as DSP transformations and filters, based on a systematic techniques for factorization, common subexpression elimination, and decomposition.

2.2 Symbolic Algebra Methods

Many computations encountered in algorithmic and behavioral design specifications can be represented as polynomial expressions. They include digital signal and image processing designs, digital filter designs, and many other designs that employ complex transformations. Polynomial representation proved to be an effective data structure for representing both high-level specification and bit-level description of a target implementation.

Polynomial models of high-level design specifications have been used in the context of behavioral synthesis, and in particular for component mapping. System SymSyn [17, 18] uses advanced...
symbolic algebra methods based on ideals and Groebner bases to automate mapping of data paths onto arithmetic blocks represented as polynomials. Each component from the library is modeled as a polynomial and is matched against the expression associated with the data path. However, these systems require explicit specification of the “side relations” (i.e., polynomials from the library of elements) to facilitate the optimization. Also, they not address behavioral transformations, such as tree height reduction and balancing in a systematic way. Instead, they rely on commercial symbolic algebra tools, such as Maple [19], to solve the polynomial decomposition problem using simplification modulo polynomial. While this may work well for the mapping onto hardware library, it does not solve a more general problem of finding the best architecture for a given optimization objective. We believe that these methods can benefit from canonical representations, such as those described in the sequel.

2.3 Kernel-based Decomposition

Algebraic methods have been used in logic optimization to reduce the number of literals in Boolean expressions of multi-level logic networks [20]. These methods perform factorization and common subexpression elimination by applying techniques of kernel extraction [21, ?]. They have been perfected over last 25 years, resulting in efficient logic synthesis tools for Boolean circuits. These methods have been recently adopted to optimize polynomial expressions representing linear DSP transforms and non-linear filters in the work of Hosangadi et. al [39]. Two main algorithms: (1) distill, which performs multi-cube decomposition and (2) condense, which performs single-cube decomposition, are adopted to optimize polynomials in their research. First, a Kernel vs Co-kernel Matrix (KCM) is constructed by the kerneling algorithm. Based on KCM, kernel and cube intersections are extracted by distill and condense algorithms, respectively. Distill algorithm, which targets multi-term subexpressions, is a greedy iterative algorithm, in which the best prime rectangle (heuristically leading to best kernel extraction), is extracted in each iteration. Similarly, condense algorithm extracts single term common subexpressions.

While kernel-based factorization provides a systematic way to optimize non-linear polynomials, the work of Hosangadi et al [?] has several drawbacks. First of all, polynomials need to be preprocessed to remove redundant terms. According to the [?], polynomials such as $a^2+ab+ab+b^2$ cannot be optimized by their algorithm because there are two instances of the cube $ab$. If such a duplication of cubes were allowed, the kernels would have also contained the duplicated terms and there is no way to represent them in the KCM. The task of removing redundant terms is a known difficult problem for larger designs. When the expression is composed of large number of terms, redundancy removal is difficult without the help of a canonical representation, such as TED. (In Boolean logic this has been resolved with a help of Binary Decision Diagrams, BDDs). Furthermore, the sum of product (SOP) and factored forms used in their work for manipulation of logic expressions are not canonical, which reduces the scope of optimization. It can be shown that similar or better results can be obtained using the canonical form proposed in this paper.

2.4 Domain Specific Systems

Several systems have been developed for domain-specific applications, such as discrete signal transforms. One such system, FFTW [22, 23], developed by researchers at MIT, targets specifically code generation for DFT computation. The system is based on a flexible composition strategy for
DFT transform of arbitrary size, based on code modules (called codelets) developed and hand-optimized for small transform sizes.

The most advanced system for DSP code generation is SPIRAL, developed at CMU and used in commercial applications [24]. SPIRAL is a code generator system that automatically generates optimized implementation of linear signal processing transforms, such as DFT, DCT, DWT, and WHT. It translates the task of finding platform-adapted implementations into an intelligent search in the space of possible fast algorithms and their implementations. It captures linear discrete signal (DSP) transforms and their fast algorithms in a concise mathematical framework. The transforms are expressed as a matrix-vector product, $y = M \cdot x$, where $x$ is a vector of $n$ data points, $M$ is an $n \times n$ matrix, and $y$ is the transformed vector. Fast algorithms for signal transforms arise from factorizations of the transform matrix $M$ into a product of sparse matrices, $M = M_1 \cdot M_2 \cdots M_n$. Such signal transforms are characterized by a property that efficient factorizations of their mathematical expressions do exist and the transform matrices $M_i$ are highly structured [24]. SPIRAL uses this structure to write the factorizations in a concise form. It automatically searches in the space of different algorithms and the space of their implementations for the best match for a given computing platform. Many optimizations are performed at a high level of program representation, thus overcoming compiler limitations. Dynamic programming is used as a main optimization engine for the best implementation.

In this paper we show how the optimization of expressions describing linear DSP transforms, filters and other designs expressed by linear and nonlinear polynomials can be performed using the generic TED data structure. In this work we concentrate on the factorization, common subexpression elimination (CSE) and decomposition, leaving the front-end part (derivation of matrices, calculation and storage of constants, etc.) and additional optimization to the dedicated systems such as SPIRAL. The goal is not to compete with these dedicated systems, but to demonstrate that the generic optimization can give comparable results both in terms of quality and the CPU time. The TED-based factorization described in this paper does not require any domain knowledge of the transform or of the computation. As such, the proposed optimization offers an interesting alternative to such specialized systems.

3 Canonical TED Representation

Taylor Expansion Diagram [11] is a canonical, word-level data structure that offers an efficient way to represent computation in a compact, factored form. 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, $f(x, y, \ldots)$, can be represented using Taylor series expansion, w.r.t. variable $x$ as follows:

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

where $f'(x)$, $f''(x)$, etc, are the successive derivatives of $f$ w.r.t. $x$. The terms of the decomposition are then decomposed with respect to the remaining variables ($y$, $\ldots$, etc.), one variable at a time. The resulting decomposition is stored as a directed acyclic graph whose nodes represent the terms of the expansion. The number of children at each node depends on the order of the polynomial expression.
(w.r.t its decomposing variable) rooted at that node. The resulting graph is reduced, normalized and canonical for a fixed ordering of variables. The expression represented by the graph is evaluated as a sum of the expressions of all the paths of the graph, from root to terminal 1.

Figure 1(a) shows one-level decomposition of function $f(x, y, \ldots)$ at variable $x$. The nodes $f(x = 0, y, \ldots), f'(x = 0, y, \ldots)$, etc, represent subsequent derivative functions that depend only on the remaining variables: $y$, etc.

Figure 1(b) shows TED for function $F = A^2 + AB + 2AC + 2BC$. Additive edges, corresponding to $F(0)$ are represented in the graph as dotted lines. Linear (first order) edges, associated with $F'(0)$ are shown as solid lines. The second order edges, associated with $F''(0)$ are shown and double lines. For the ordering of variables $(A, B, C)$, the reduced and normalized TED is constructed as follows. The decomposition is performed first with respect to variable $A$, producing the following terms: $F(A = 0) = 2BC$, $F'(A = 0) = B + 2C$, and $\frac{1}{2} \cdot F''(A = 0) = 1$. Next the expansion is applied to the resulting non-trivial terms $F(0), F'(0)$ with respect to variable $B$, and subsequently with respect to variable $C$. Note the multiplicative weights assigned to the edges (default weight is 1).

TED is a multiplicative diagram which maps word-level (integer) inputs into integer, word-level (integer) outputs. The function encoded in the TED is computed as a sum of the expressions of all the paths from root to terminal node 1. (For this reason, TED can be viewed as a graph without edges leading to terminal 0, and containing only terminal node 1.) The expression of each path is computed as a product of the expressions of all the edges in the path; the expression of each edge is in turn a product of the variable in its respective power, weighted by the numerical label (coefficient) assigned to the edge. For example, the expression for the function encoded in the TED in Figure 1(b) is computed as a sum of four paths from root to terminal node 1: $F = A^2 + AB + 2AC + 2BC$.

In this work we will concentrate on TEDs representing multi-variate linear polynomials, in which each variable appears only in first degree (power). This is the case for all the linear discrete transforms and linear filters discussed in this paper. In this case each node has at most two edges (edges connected to node zero are removed from the diagram). We will refer to such TEDs as linear TEDs. Figure 2 shows an example of a linear TED for a function $F = 3a + 5ab + 2b$. (Note: despite its apparent similarity, TED for a linear form is different than BMD, and represent arithmetic expressions with real variables). We shall later explain how a non-linear polynomials can be also represented as linear TED.

In the TED graph, each outgoing edge of the vertex corresponding to some variable $v$ is labeled with a pair of numbers $(k, l)$, where $k$ indicates the degree of the parent vertex $v$, and $l$ indicates the coefficient of variable $v^k$ in the original expression. (Actual labels may be different due to the normalization of the graph, the topic beyond the scope of this paper). In our case, since the
function is linear in all its support variables, each node of the graph contains up to two edges: additive (dotted) edge, labeled 0; and multiplicative (solid) edge, labeled 1. Edges labeled with coefficients 0 are not shown.

The function of the graph is evaluated by adding the product of all the vertices in their respective powers and their coefficient labels along all the paths from the root to terminal 1. In the case of function $F1(a, b) = 3a + 5ab + 2b$ in Figure 2, we have: $F1 = 1a^1*(5*b^1+3*b^0)+2*a^0*(1*b^1)$. This notation will be used throughout the paper.

In summary, TED will serve as a canonical view of the computation, regardless of the specific data flow or architectural implementation.

3.1 TED-based Factorization and Common Subexpression Extraction

We will now demonstrate how to perform factorization of expressions for linear DSP transforms using the TED data structure. Here we concentrate only on the factorization and common subexpression elimination (CSE), leaving the front-end part (derivation of matrices, calculation and storage of constants, etc.) to other systems, such as MATLAB or SPIRAL [24]. Our goal is not to compete with these advanced dedicated systems, but to show that the generic factorization performed on a TED can give comparable results with a shorter CPU runtime. TED-based factorization described here does not require any knowledge of the transform or the computation, it can be applied to arbitrary data flow computation, and performs efficient factorizations directly on the graph.

We illustrate the factorization of the DSP transform specification by the following example of the Discrete Cosine Transform (DCT), used frequently in multimedia applications. DCT is an important element of all the compression algorithms, and in the JPEG algorithm. The DCT of type 2 is defined by: The DCT of type 2 is defined as

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

and computed by the following algorithm:

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

The first task is to create an array to store all the cosine term needed for the computation. DCT2 can be computed as

\[ y = M \cdot x \]

where \( y \) and \( x \) are the output and input vectors and \( M \) is the transform matrix composed of the cosine terms in eq(2). In its direct form the computation involves 16 multiplication and 12 addition for the DCT of size 4.

\[
M = \begin{bmatrix}
\cos(0) & \cos(0) & \cos(0) & \cos(0) \\
\cos(\frac{\pi}{8}) & \cos(\frac{3\pi}{8}) & \cos(\frac{5\pi}{8}) & \cos(\frac{7\pi}{8}) \\
\cos(\frac{\pi}{4}) & \cos(\frac{3\pi}{4}) & \cos(\frac{5\pi}{4}) & \cos(\frac{7\pi}{4}) \\
\cos(\frac{3\pi}{8}) & \cos(\frac{5\pi}{8}) & \cos(\frac{7\pi}{8}) & \cos(\frac{9\pi}{8})
\end{bmatrix}
\] (2)

By recognizing the dependence between the cosine terms and replacing coefficients with the same numeric value by a unique symbolic variable, the above matrix can be represented in the following form with four distinct coefficients, \( A, B, C, \) and \( D \) (for simplicity, neglect the fact that \( A = \cos(0) = 1 \)).

\[
M = \begin{bmatrix}
B & C & -C & -B \\
D & -D & -D & D \\
C & -B & B & -C
\end{bmatrix}
\] (3)

Subsequently, it is possible to factorize the expressions and reduce the number of the ADD/MPY operations to 6 MPY and 8 ADDs, as shown by the equations 4. This 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.

\[
y_0 = A \cdot ((x_0 + x_3) + (x_1 + x_2)) \\
y_1 = B \cdot (x_0 - x_3) + C \cdot (x_1 - x_2) \\
y_2 = D \cdot ((x_0 + x_3) - (x_1 + x_2)) \\
y_3 = C \cdot (x_0 - x_3) - B \cdot (x_1 - x_2)
\] (4)

Even though this approach seems obvious and is commonly used by those who work in signal processing field, there are no systematic methods to perform this simplification automatically. Neither C compilers (like GCC) nor synthesis tools (like Synopsys’ BC, Mentor’s RTL Precision, or Synplicity’s Synplify DSP) offer this useful feature.

Different approaches are used to answer this problem, such as the use of dominator trees and code motion speculation during the scheduling step in high level synthesis [1]. Others perform factorization by using mathematic properties of the DSP transform [25] [24]. In the SPIRAL system [24] the optimization tool is made around a mathematical framework that finds best implementation by using manipulation and breakdown rules on the given transform until the decomposition appears with the butterfly pattern. This tool works with parameters and keywords as input and is not able to perform the generation of optimized code on specifications other than transforms known to the tool.
3.2 TED-based Factorization Algorithm

We now describe how TED can be used as a vehicle to perform efficient factorization of algebraic expressions, and in particular of expressions representing linear DSP transforms.

The key to achieve optimum factorization is to represent constants as TED nodes and keep them on top of the diagram. This is in contrast to a traditional TED representation, where constants are represented as labels on the graph edges. In such ordered graph, useful factorizations are manifested in the graph by the presence of one or more nodes with multiple incoming (parent) edges. We refer to those edges as reference edges. The fact that a node is referred by several parent edges simply means that the subfunction rooted at that node is shared by several expressions, each represented by a path from the TED root to the terminal node 1.

We illustrate our factorization algorithm with an example of the DCT2 transform of size 4 (DCT2-4), described by the symbolic matrix 3. The matrix  for the DCT2-4 example has four distinct coefficients, \( A, B, C, D \) treated as symbolic variables and placed on top of the TED. Figure 3(a) shows the TED graph of DCT2-4, using a default, lexicographical variable ordering, with the constant nodes on top.

We introduce the following definition:

**Definition** Candidate node is a TED node with more than one reference edge. Factorization node is a candidate TED node such that the expression rooted at the node is not shared by any other expression, except for a single literal. Factorization node is trivial if its expression involves a single variable (the node itself).

It should be clear that only nontrivial factorization nodes can contribute to useful factorizations. By definition, the function rooted at a factorization node will only include nodes that are not referred by other nodes, with the exception for the single bottom variable nodes.
In our DCT2-4 example in Figure 3(a) several nodes qualify as factorization nodes: the node with variable $x_0$, associated with expression $(x_0 - x_3)$; and the node with variable $x_1$, associated with expression $(x_1 - x_2)$. We can now extract these expressions from the graph and substitute them by new variables.

The TED-based factorization algorithm is summarized by the flow chart shown in Figure 4.

![Flow chart of the TED-based factorization algorithm](image)

**Figure 4: Flow chart of the TED-based factorization algorithm**

The algorithm is composed of the following steps:

1. Construct the TED with constants as top variables in the TED. The initial ordering of the remaining variables is used to minimize the TED size [26]. (In case of DSP transforms lexicographical ordering is sufficient). Ordering of TED in subsequent iterations depends on the outcome of Steps 2 and 3.

   (a) If there are any new nodes generated in Step 2, place them on top, below the constant nodes.

   (b) If there are bottom variables in Step 3 that were not extracted, place them on top, below the constant nodes.

2. Find factorization nodes; if none exist, go to Step 3. Otherwise select one node (arbitrary, or with largest factorization gain), extract the expression rooted at that node and substitute it with a new variable. **Go to Step 1** to create a new TED, with the new node placed on top.

3. At this point here are no more factorization nodes for the current ordering of variables. The variable(s) at the bottom that do not contribute to factorization need(s) are pushed to the top.
to expose new variables for factorization. **Go to Step 1** to create new TED with the bottom variables placed at the top.

**Stop:** The procedure terminates when there are no more factorization nodes and there are no nodes that have not been pushed up.

Let us illustrate this algorithm with our DCT2-4 example, for the TED constructed in Step 1 with the ordering \{A, B, C, D, x_0, x_1, x_2, x_3\}, shown in Figure 3(a).

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, \(S_1 = (x_0 - x_3)\). The new variable \(S_1\) is 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 \(S_2 = (x_0 + x_3)\). Once the subexpression is extracted, variable \(S_2\) is also pushed up. The next iterations of the algorithm leads to substitutions \(S_3 = (x_1 - x_2)\) and \(S_4 = (x_1 + x_2)\), 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: 
\[
\begin{align*}
y_0 &= A \cdot (S_2 + S_4), \\
y_1 &= B \cdot S_1 + C \cdot S_3, \\
y_2 &= D \cdot (S_2 - S_4), \\
y_3 &= C \cdot S_1 - B \cdot S_3,
\end{align*}
\]
where \(S_1 = (x_0 - x_3), \ S_2 = (x_0 + x_3), \ S_3 = (x_1 - x_2)\) and \(S_4 = (x_1 + x_2)\).

The computation of such optimized expressions requires 5 multiplications and 8 additions, a significant reduction from the 16 multiplications and 12 additions of the initial expressions.

Another application of this algorithm for DFT of size 8 is shown in Figure 5 and Figure 6. This results in an optimum factorization, and the reduction of TED from the initial 101 nodes to final 29 nodes.

![Figure 5: Initial TED of DFT8.](image)

In conclusion, TED-based factorization can be used for factorization of generic expressions; the factorization algorithm does not require any specific knowledge of the nature and structure of the expression (DSP transformation) to obtain an efficient factorization.
4 Extraction

TO BE COMPLETED

Explain the extraction algorithm, which allows to extract/factor out those forms which cannot be handled by dynamic factorization described in the previous section.

Example: \( f = a(b + c) \) with variable order \( a, b, c \). Factorization algorithm cannot factor out \( a \) for this order.

The combination of the two algorithms makes it possible to obtain minimal factored forms.

5 Linearization of TEDs

As discussed in Section 3, TEDs provide an efficient representation for verification of polynomials of arbitrary degree. TEDs are also efficient in factorization and decomposition of expressions that can be modeled as multi-variate polynomials (linear DSP transforms being a case in point). Unfortunately, this efficiency is missing when considering optimization involving non-linear expressions. As an example, consider TED of a nonlinear function \( F = a^2 + a \cdot b \), shown in Figure 7(a), with node \( a \) having both edges, linear and non-linear (recall that the degree of an edge is denoted as the first label assigned to the edge). The TED optimization algorithms designed to minimize the number of nodes or the number of operators (particularly multipliers), discussed earlier, cannot further reduce the diagram: the graph is already minimized in terms of the number of edges and nodes. Furthermore the nonlinear representation of edges does not allow to factor out common variable \( a \) which would minimize the number of multipliers. Clearly, expression \( F = a \cdot (a + b) \), which requires only one multiplier and one adder, is a desired optimal representation for function \( F \).
Figure 7: (a) original TED for $F = a^2 + a \cdot b$; (b) linerized TED representing function $F = a_1 \cdot (a_2 + b)$.

Fortunately, TEDs can be readily transformed into a form that can support this kind of factorization by the process called TED linearization. The linearization procedure involves reconstituting a TED so that each variable appears only in linear term. Conceptually, the linearized TED represents an expression in which each variable $x^k$, for $k > 1$, is written as a product of $k$ variables, i.e., $x = x_1 \cdot x_2 \cdots x_k$, with the understanding that $x_i = x_j$ for all values of $i, j \leq k$. In our example, function $F = a^2 + a \cdot b$ can be rewritten as $F = a_1 \cdot a_2 + a_1 \cdot b$, which reduces to $F = a_1 \cdot (a_2 + b)$, or equivalently to $F = a \cdot (a + b)$, after factoring out $a$ (or, equivalently, extracting $(a + b)$). In practice, such a transformation is performed directly on a TED, without a need to rewrite the expression, by the algorithm discussed next. Factorization and common subexpression elimination can then performed on the transformed TED using all the previously developed TED factorization, decomposition and extraction techniques.

5.1 Linearization Algorithm

A non-linear polynomial can be transformed into a linear one by splitting variables with degree $k > 1$ into $k$ variables. For example, introducing variables $x_1, x_2, \cdots, x_k$ transforms the non-linear expression in Equation 5 into its linear form, shown in Equation 7. Note that this form makes it possible to factorize the expression to a well known Horner form, with minimum number of multiplications, see eq (7).

$$F(x) = f_0 + x \cdot f_1 + \cdots + x^n \cdot f_n \quad (5)$$
$$= f_0 + x_1(f_1 + \cdots + x^{n-1} \cdot f_n) \quad (6)$$
$$= f_0 + x_1(f_1 + x_2(f_2 + \cdots + x_{n-1}(f_{n-1} + x_n \cdot f_n)) \cdots) \quad (7)$$

A non-linear TED can be transformed into a linear TED by recursively splitting the nodes. Figure 8 (a) shows a non-linear TED with node $x$ in power $n$, with $n+1$ children functions: $f_0, f_1, \cdots, f_n$. 

13
Without loss of generality, child weights are assumed to be 1. Labels on edges depict their degree. The splitting is performed recursively; first the node $x^n$ is split into two nodes, $x_1$ and $x^{n-1}$, which reduces the degree of $x$ from $n$ to $n - 1$. The zero-degree child $f_0$ of node $x$ remains at the top node, $x_1$, and the first-degree child becomes a zero-degree child of node $x^{n-1}$, etc. The degrees of all other children of node $x$, $(f_1 \cdots f_n)$, are reduced by 1. Following these steps, non-linear TED in Figure 8 (a) is transformed into the TED in Figure 8 (b), with maximal degree of node $x$ reduced by 1. Recursively applying the above steps to node $x^{n-1}$ leads to a linearized TED shown in Figure 8(c). Notice that this TED represents the polynomial in Horner form.

During these steps, newly introduced nodes are subject to canonicity check to ensure the canonicity property. When there are more than one nodes sharing exact same children, only one of them is chosen to represent the function, others are deleted from the memory.

**Algorithm 1**

A non-linear TED node $x$ linearized TED nodes $x_1, x_2 \cdots x_n$ $i = 1$

**degree**($x$) > 1 Unlink zero-degree child of node $x$

Decrease degrees of all other children of node $x$ by 1

$x =$ CheckTedCanonicity($x$)

Introduce node $x_i$

Set the zero-degree child of $x_i$ to the unlinked zero-degree child of node $x$

Set the first-degree child of $x_i$ to node $x$

$x_i =$ CheckTedCanonicity($x_i$) TED node linearization

### 5.2 Experimental Results for Non-linear Polynomials

Qian: Show tables of results.

Compare to non-linear polynomials from Farzan Fallah. Comment on results in terms of the number of multipliers, etc.
6 TEDify System

TO BE COMPLETED
Describe the system ...

7 Experimental Results

TO BE COMPLETED
Summarize and discuss the results in tables.

References


