CONSIDERABLE PROGRESS has been made during the past two decades in behavioral and high-level synthesis (HLS), making it possible to synthesize designs specified using standard programming languages such as C, C++, and System C. HLS tools automatically generate an RTL specification of the circuit from a bit-accurate algorithm description for a given target technology and the application constraints (latency, throughput, resource utilization, and so on). The algorithmic description used as input to HLS does not require explicit timing information and thus provides a higher level of abstraction than the RTL model. Thanks to HLS, the designer can faster and more easily explore different architectural solutions.

However, the optimization algorithms used by HLS, such as scheduling, resource binding, and allocation, operate on a fixed dataflow 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 depend on that initial specification. To explore other solutions, the user needs to rewrite the original specification, from which another DFG is derived and used for synthesis.

We propose a novel method in which the initial functional specification is first transformed into a canonical form and then converted into a DFG optimized for a particular application. The goal is to generate a DFG that, when given as input to a standard HLS tool, will produce a hardware implementation optimized for latency, hardware cost, or both. The optimizing transformations are based on a canonical, graph-based representation called a Taylor expansion diagram (TED). DFGs generated with this method are better suited for HLS than those extracted directly from the initial specification, or obtained using structural DFG transformations and algebraic decomposition methods (see the sidebar, “Dataflow Graph Transformations: Related Work”).

We have implemented the optimizing transformations in an experimental software system, called TDS (TED-based decomposition system), intended for dataflow- and computation-intensive designs used in computer graphics and digital signal processing (DSP) applications. The system is available online.

Introduction to TEDs

A Taylor expansion diagram, or TED, is a compact, graph-based data structure that provides an efficient way to represent word-level computation in a canonical, factored form. TEDs are particularly suitable for computations modeled as polynomial expressions.

Figure 1 illustrates the idea of design space exploration based on functional TED representation. In the
Dataflow Graph Transformations: Related Work

Automatic transformation of design specifications 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 high-level synthesis tools such as Spark, Cyber, and Catapult C. 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 and decomposition methods, successfully used in logic synthesis, have also been used to optimize polynomial expressions of linear digital signal processing (DSP) transforms and nonlinear filters. However, the polynomial representation that these methods employ is not canonical, which seriously reduces the scope of optimization.

Several systems have been developed for domain-specific applications such as digital filters and discrete signal transforms. These systems employ optimizations that rely on knowledge of the transforms’ mathematical properties. In general, however, they cannot handle computations described by nonlinear polynomials, such as those found in computer graphics and nonlinear filter designs.

The method described in the main text transforms an initial design specification of a generic nonlinear computation into an optimized dataflow graph. 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.

References

Traditional synthesis flow, a single DFG is extracted from the initial (functional) specification and used as input to HLS to generate the final implementation (the hardware architecture). Optimization of the resulting architectural design obtained from a fixed DFG is thus limited to local modifications on the register-transfer level. Figure 1 shows the set of such solutions as a cone associated with a given DFG. To improve 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 as height tree reduction and balancing, which are limited in scope and allow exploration of only a small fraction of the entire solution space.

In contrast, transformations of the design specification performed on the functional level can produce a set of functionally equivalent DFGs. The designer can select one such DFG, on the basis of a given objective and design constraints, to generate the final implementation.
which corresponds to equation (8). Solution (a) represents a
multivariate polynomial expression, \( f(x, y, \ldots) \),
and that equation (8) can be represented using Taylor series
expansion around the origin \( x = 0 \):

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

where \( f' (0, y, \ldots), f'' (0, y, \ldots) \), and so on, are the
successive derivatives of \( f \) with respect to \( x \), evaluated
at \( x = 0 \). The expression’s individual terms are then
decomposed iteratively with respect to the remaining
variables on which they depend \((y, \ldots, \text{etc.)}, \) one
variable at a time.

The resulting decomposition is stored as a
directed, acyclic graph, a TED. Each node of the
graph is labeled with the name of the variable at
the current decomposition level and represents the
expression rooted at this node. The TED’s top node
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 \((p, w)\), where \( p \)
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 that represent linear multi-
virate polynomials. Nonlinear expressions can be
easily converted into linear ones, by transforming
each occurrence of a nonlinear term \( x^p \) into a pro-
duct \( x_1 \cdot \ldots \cdot x_p \), where \( x_i = x \). A linear TED contains
only two types of edges: the additive edges (labeled
with power \( \wedge 0 \)), which are represented in the graph
as dotted edges; and the linear edges (labeled
with \( \wedge 1 \)), which are represented as solid lines. Linear
edges are also called multiplicative edges, because 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 expres-
sion for each path is obtained as a product of the
dge expressions, each being a product of the vari-
able in its respective power and the corresponding
edge weight. Only nonempty terms, corresponding
to edges with nonzero weights, are stored in the graph.
First decomposition level, 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 $(^0 1)$, leading to node $B$. Let us denote the term $F = B + C$ by $G(B, C)$.

Second decomposition level, variable $B$: The term $G(B = 0, C) = C$ corresponds to an additive edge, labeled $(^0 0, 1)$, incident to node $C$. The term $G'(B = 0, C) = 1$ is a linear edge, labeled $(^0 1)$, connected to terminal node 1.

Third decomposition level, variable $C$: $C(0) = 0$ is an empty additive term (not shown). The last term, $C = 1$, is a linear edge leading to terminal node 1.

Figure 3a shows the resulting TED. The expression encoded in the graph is computed as a sum of two paths from the TED root to terminal node 1: $A \cdot B$ and $A \cdot B^0 \cdot C = A \cdot C$—that is, $AB + AC$. In fact, the TED encodes such an expression in factored form, $F = A(B + C)$, because 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 3b. This ordering does not expose any particular factored form, and hence is not useful for factorization.

In summary, a TED is a graphical representation of finite multivariate polynomials that maps word-level (integer) inputs into word-level outputs. A TED is reduced and normalized similarly to binary decision diagrams (BDDs) and binary moment diagrams (BMDs). Finally, the reduced, normalized, and ordered TED is canonical for a given variable order. However, despite an apparent similarity between linear TEDs, BDDs, and BMDs, the three representations are different. A 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 $a(b + 1)$, whereas a BDD for $ab + a$ reduces to $a$. A BMD for the same function will be the same as a TED only if the inputs $a, b$ are single-bit variables; otherwise the BMD must represent each input in terms of its component bits. A detailed description of the TED representation with its application to verification and high-level synthesis is available in work by Ciesielski, Kalla, and Askar, and by Gomez-Prado et al.

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 subexpression 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) through factorization and CSE is commonly called decomposition.
Decomposition of algebraic expressions can be performed directly on the TED graph, because 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 or hardware cost of the scheduled, synthesized design.

Here we describe two methods for TED decomposition, each applicable to a different class of designs. One is based on factorization and CSE performed on a static TED, with a fixed variable order. This method applies 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 systematically. This method is more suitable for structured linear DSP transforms with common computing patterns.

Static TED decomposition

Static decomposition consists of hierarchical, cut-based decomposition and subgraph extraction, followed by transformation of the decomposed TED into a DFG. Gomez-Prado et al.\(^5\) and Ciesiel\'ski et al.\(^6\) provide details of the complete TED decomposition procedure.

Cut-based decomposition is based on identifying a sequence of cuts in the TED structure to decompose the TED into disjoint subgraphs.\(^5\) An additive cut, denoted by \(A_i\), applied to an additive edge, partitions the graph into two subgraphs, 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 the property that all the paths from the TED root to terminal node 1 must pass through it. 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 4a shows the hierarchical TED decomposition for an expression \(P = G + H + F(I + J)\) for the \((A_3, A_1, M_1, A_2)\) cut sequence (a); DFG obtained for this cut sequence (b); and DFG for the \((A_1, A_3, M_1, A_2)\) cut sequence (c).

Decomposition of algebraic expressions can be performed directly on the TED graph, because 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 or hardware cost of the scheduled, synthesized design.

Here we describe two methods for TED decomposition, each applicable to a different class of designs. One is based on factorization and CSE performed on a static TED, with a fixed variable order. This method applies 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 systematically. This method is more suitable for structured linear DSP transforms with common computing patterns.

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

An important property of this decomposition is that different cut sequences generate different DFGs, from which the designer can select the one with the required property (such as minimum latency or resource utilization). Figures 4b and 4c 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 4a. The two DFGs differ in their structure (tree versus serial) and hence will produce different hardware architectures.

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

Figure 5 illustrates a nondisjoint decomposition for the expression \( F = ac + bc + ad + bd + ab \), encoded in the TED in Figure 5a 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 nondisjoint subgraphs, \( F_1 = ab \) and \( F_2 = (c + d)(a + b) \), with subgraph \( b \) being common to both partitions, as shown in Figure 5b. Such a decomposition is accomplished by explicit extraction of common subgraphs. An \( M \) cut then decomposes expression \( F_2 \) conjunctively into \( (c + d) \) and \( (a + b) \); Figure 5c shows the resulting decomposition.

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

We should point out that the expression obtained from the TED decomposition depends on the TED’s structure (and its variable order). Such an expression is called the normal factored form (NFF) for a given TED. (We will formally define NFF and prove its properties later in the article, in the context of DFG generation.)

Dynamic TED factorization

An alternative approach to TED decomposition is based on a dynamic factorization and CSE. We can illustrate this approach with an example of the discrete cosine transform (DCT) used frequently in multimedia applications. A type 2 DCT is defined as

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

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. Equation 1 shows Matrix \( M \) for \( N = 4 \).

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

\[
= \begin{bmatrix}
B & C & -C & -B \\
D & -D & -D & D \\
C & -B & B & -C
\end{bmatrix}
\]
High-Level Synthesis

Equation 2. We can achieve this simplification 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 replacing them with new variables.

\[
\begin{align*}
y_0 &= A \cdot [(x_0 + x_3) + (x_1 + x_2)] \\
y_1 &= B \cdot (x_0 - x_3) + C \cdot (x_1 - x_2) \\
y_2 &= D \cdot [(x_0 + x_3) - (x_1 + x_2)] \\
y_3 &= C \cdot (x_0 - x_3) - B \cdot (x_1 - x_2)
\end{align*}
\]

Figure 6a shows the initial TED representation for the DCT matrix in Equation 1. The subsequent parts of the figure show the transformation of the TED that produces the factorization in Equation 2.

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 at the top part of the TED graph. This is in contrast to a traditional TED representation, in which constants are represented as labels on the graph edges. In the case of DCT, coefficients \(A, B, C, D\) are treated as symbolic variables and placed at the top part of the TED, as in Figure 6a.

The dynamic CSE algorithm obtains candidate expressions for factorization in such a TED by identifying the nodes with multiple incoming (parent) edges. The subexpressions rooted at such nodes are extracted from the graph and replaced with new variables. As a rule, the 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 6a 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 replaced with new variables \(S_1 = (x_0 - x_3)\) and \(S_2 = (x_0 + x_3)\) as they correspond to the nodes located lowest in the TED. The algorithm then pushes up variables \(S_1\) and \(S_2\), below the constant nodes, and reorders the TED to expose new candidates for extraction. Figure 6b shows the result. The next step is to extract new candidate expressions, \((x_1 - x_2)\) and \((x_1 + x_2)\), and replace them with variables \(S_3\) and \(S_4\), resulting in the TED shown in Figure 6c. At this point, there are no more nontrivial expressions to be extracted, and the algorithm terminates. (Guillot has described the details of the dynamic CSE algorithm.) Consequently, the dynamic TED-based CSE results in the following simplified expressions:

\[
\begin{align*}
y_0 &= A \cdot (S_2 + S_4) \\
y_1 &= B \cdot S_1 + C \cdot S_3 \\
y_2 &= D \cdot (S_2 - S_1) \\
y_3 &= C \cdot S_1 - B \cdot S_3
\end{align*}
\]
Considering that $A = 1$, the computation of such optimized expressions requires only five multiplications and eight 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.

**DFG generation and optimization**

The recursive TED decomposition procedures produce simplified algebraic expressions in factored form. By imposing additional rules regarding variable ordering in the expression, we can make such a form unique.

**Definition.** The factored form expression associated with a given TED is called the normal factored form (NFF) if the order of variables in the expressions is compatible with the order of variables in the TED.

**Theorem.** The NFF 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 $F = F_1$ or $F_2$, where OR is the 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 ($F_1$ or $F_2$ versus $F_2$ or $F_1$) 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.

The TED in Figures 5a–c illustrates the uniqueness of the NFF for the variable order $[c, d, a, b]$. On the top decomposition level, shown in Figure 5b, the TED is split into two nondisjoint 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 $F_{11} \cdot F_{12}$, where $F_{11} = (c + d)$ and $F_{12} = (a + b)$, each with a 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 algorithms, 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 (as soon as possible) scheduling 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 takes place in the background during the TED decomposition.

However, unlike the NFF, the DFG representation is not unique. Although 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 with logarithmic trees, one can also apply standard logic synthesis methods, such as collapsing and re-decomposition. Furthermore, multiplications by constants can be replaced with 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 TED decomposition is that it gives insight into the final DFG structure. One can generate different DFG solutions 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.
TDS system

We have implemented the TED decomposition as part of a prototype system, TDS, shown in Figure 7. The input to the system is the design specified in C/C++ or given in the form of a DSP matrix. The left part of the figure shows the traditional HLS flow, which extracts a DFG from the initial specification and performs standard HLS operations: scheduling, allocation, and resource binding. The right part of the figure shows the actual TDS optimization flow. It transforms the DFG extracted from the initial specification into a hybrid network composed of functional blocks, represented with TEDs, and other operators that cannot be expressed as TEDs, which the system treats as black boxes. TDS optimizes the resulting hybrid network and transforms it into a final DFG using TED- and DFG-related optimizations. TDS then balances the entire DFG network 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, linearization, replacement of constant multiplications by shifters, DFG construction, balancing, and so on. The decomposition quality is controlled by choosing initial TED ordering and reordering of component TEDs.

Experimental results

We tested the TDS system on several practical designs from computer graphics and DSP applications, with a primary goal of minimizing the DFG latency.
Table 1 compares the implementation of a quintic spline filter using
- the original design written in C;
- the design produced by a kernel-based decomposition (KBD) system;10 and
- the design produced by TDS.

We used GAUT to synthesize the DFGs generated by these systems. In Table 1, the first row under the column headings reports the number of arithmetic operations (adders, multipliers, shifters, subtractors) in the unscheduled DFG. The remaining rows give the number of resources and datapath area for a given latency in a scheduled DFG, as reported by GAUT. Boldface indicates the minimum attainable latency for each of the three solutions. A dash indicates that a circuit cannot be synthesized at a given latency.

The DFG latency obtained by TDS is 110 ns, 21.4% shorter than that obtained by KBD, which is 140 ns. Although the KBD solution has the smallest number of operations in its unscheduled DFG at a latency of 140 ns (KBD's minimum), the design synthesized with this DFG requires more resources than the one obtained by TDS. Specifically, the design area of the TDS implementation is 22% smaller than that of the KBD solution. We observed similar behavior for all the tested designs.

Table 2 summarizes the implementation results for these benchmark designs. The implementations obtained by TDS have latency that on average is 15.5% shorter than the latency of implementations produced by KBD, and 27.2% shorter than the latency of the original design DFGs. The area results reported in Table 2 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 quartic design obtained with the DFG produced by TDS requires more area than the one produced by KBD as measured at the reference latency; this is because our main objective was to minimize the DFG latency. In particular, for this design TDS generated a DFG with a minimum latency of 110 ns, compared to 130 ns for KBD. The TDS solution is characterized by more parallelism, which for the 130-ns reference latency required more resources than the KBD solution.

**OUR NEW APPROACH** for transforming the initial specification of dataflow designs, based on TED decomposition, obtains solutions with minimum latency or minimum resource cost in a scheduled dataflow graph. The designs obtained with DFGs generated
by our approach are characterized by lower latency and better resource utilization than those obtained by a straightforward minimization of the number of arithmetic operations in the algebraic expression.

Although currently the TDS prototype system is integrated with GAUT, an academic HLS tool, we believe that it could be successfully used as a precompilation step with 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 the high-level synthesis. Finally, the TED model implicitly assumes infinite-precision arithmetic. Future work should address the issue of the finite precision to make the system applicable to real DSP applications.

Acknowledgments
This work was supported in part by the US National Science Foundation, award no. CCF-0702506, and in part by France’s Centre national de la recherche scientifique (CNRS), contract Projet International de Coopération Scientifique (PICS) 3053.

References

Maciej Ciesielski is a professor in the Electrical and Computer Engineering Department at the University of Massachusetts, Amherst. His research interests include electronic design automation, logic and high-level synthesis, and formal verification. He has a PhD in electrical engineering from the University of Massachusetts, Amherst.
Rochester and holds a doctorate Honoris Causa from the Université de Bretagne Sud, France.

Jeremie Guillot is currently a postdoctoral researcher at LIRMM (Montpellier, France) where he works on the adaptability of MPSoC architecture. He has a PhD in microelectronics from the Université de Bretagne Sud, France.

Daniel Gomez-Prado is a PhD candidate at the University of Massachusetts, Amherst. His research interests include high-level synthesis, functional verification, and design space exploration. He has an MS in electrical and computer engineering from the University of Massachusetts, Amherst.

Emmanuel Boutillon is a professor at Lab-STICC, Université de Bretagne Sud, France. His current research interests are the interactions between algorithm and architecture in wireless communications. He has a PhD in electronics from Telecom Paris Tech, France.

Direct questions and comments about this article to Maciej Ciesielski, Dept. of Electrical and Computer Engineering, 309B Knowles Engineering Building, Amherst, MA 01003; ciesiel@ecs.umass.edu.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/csdl.