Optimizing Data Flow Graphs to Minimize Hardware Implementation

D. Gomez-Prado, Q. Ren, M. Ciesielski
ECE Dept., University of Massachusetts
Amherst , MA 01003, USA
{*dgomezpr, qren, ciesiel*}*@ecs.umass.edu*

J. Guillot, E. Boutillon
LESTER, Université de Bretagne Sud
56321 Lorient, France
{*jguillot, emmanuel.boutillon*}*@univ-ubs.fr*

Abstract - This paper describes an efficient graph-based method to optimize data-flow expressions for best hardware implementation. The method is based on factorization, common subexpression elimination (CSE) and decomposition of algebraic expressions performed on a canonical representation, Taylor Expansion Diagram. The method is generic, applicable to arbitrary algebraic expressions and does not require specific knowledge of the application domain. Experimental results show that the DFGs generated from such optimized expressions are better suited for high level synthesis, and the final, scheduled implementations are characterized, on average, by 15.5% lower latency and 7.6% better area than those obtained using traditional CSE and algebraic decomposition.

1 Introduction

Many computations encountered in high-level design specifications are represented as polynomial expressions. They are used in computer graphics designs and Digital Signal Processing (DSP) applications, where designs are specified as algorithms written in C/C++. To deal with such abstract descriptions designers need efficient optimization tools to optimize the initial specification code, prior to architectural (high-level) synthesis. Unfortunately, conventional compilers do not provide sufficient support for this task. On the other hand, architectural optimization techniques, such as scheduling, resource allocation and binding, employed by high-level synthesis tools, do not address the front-end, algorithmic optimization [1]. These tools rely on a representation that is derived by a direct translation of the original design specifications, leaving a possible modification of that specification to the designer. As a result, the scope of the ensuing architectural optimization is seriously limited.

This paper introduces a systematic method to perform optimization of the initial design specification using a canonical, graph-based representation, called Taylor Expansion Diagram (TED) [2]. TEDs have already been applied to functional optimization, such as factorization and common subexpression elimination (CSE). However, so far their scope was limited to linear expressions, such as linear DSP transforms and to the simplification of arithmetic expressions, without considering final scheduled implementation [3, 4].

This paper describes how TEDs can be extended to handle the optimization of *nonlinear* polynomial expressions, using novel factorization and decomposition algorithms, to generate optimized data flow graphs (DFG), better suited for high-level synthesis. The optimization involves minimization of the latency and of the hardware cost of arithmetic operations in the final, scheduled implementations, and not just the minimization of the number of arithmetic operations, as done in all previous work. At the same time, expressions with constant multiplications are replaced by shifters and adders to further minimize the hardware cost. The proposed method have been implemented in a software tool, TDS, available online [5].

Experimental results show that the DFGs generated from the optimized expressions have smaller latency than those obtained using traditional algebraic techniques; they also require, on average, less area than those provided by currently available methods and tools.

2 Previous Work

Research in the optimization of the initial design specifications for hardware designs falls in several categories.

HDL Compilers. Several attempts have been made to provide optimizing transformations in high-level synthesis, HDL compilers [6, 7], and logic synthesis [8]. These methods rely on the application of basic algebraic properties, applied by term rewriting rules to manipulate the algebraic expressions. In general, they do not offer systematic way to optimize the initial design specification or to derive optimum data flow graphs for high-level synthesis. While high-level synthesis systems, such as Cyber [9] and Spark [10], apply methods of code optimization, they do not rely on any canonical representation that would guarantee even local optimality of the transformations.

Domain Specific Systems. Several systems have been developed for domain-specific applications, such as discrete signal transforms. SPIRAL [11] generates optimized implementation of linear signal processing transforms, such as DFT, DCT, DWT, etc. These signal transforms are characterized by highly structured form with known efficient factorizations and radix-2 decomposition. SPIRAL uses these properties to obtain solutions in a concise form and applies dynamic programming to find the best implementation. Those tools are very efficient in the DSP domain but are not useful in the general case.

Kernel-based Decomposition. Algebraic methods have been used in logic optimization to reduce the num-

ber of literals in Boolean logic expressions. Kernel-based decomposition, employed by logic synthesis, has been recently adapted to optimize polynomial expressions of linear DSP transforms and non-linear filters [12]. While this method provides a systematic approach to polynomial optimization, the polynomial representation is not canonical, which seriously reduces the scope of optimization.

In this paper we show how TEDs can be extended to offer an alternative solution not only to the generic problem of the optimization of non-linear polynomials but also to the efficient generation of DFGs, better suited for high-level synthesis.

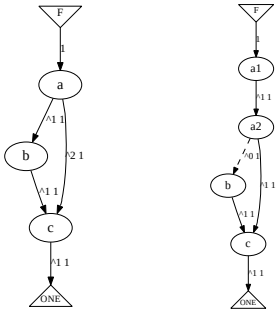
3 Polynomial Representation using TED

TED is a graph-based representation for multi-variate polynomials [2, 13] obtained from Taylor expansion:

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

The expression is decomposed iteratively, one variable at a time, in a predetermined order. The resulting decomposition is stored as a directed acyclic graph whose nodes represent the terms of the expansion. Each TED *node* is labeled with the name of the decomposing variable. Each *edge* is labeled with a pair $(^\wedge p, w)$, where $^\wedge p$ represents the power of the variable and w represents the edge weight. The resulting reduced, normalized graph is canonical for a fixed variable order.

An example of a TED for expression $F = a^2c + a \cdot b \cdot c$ is shown in Fig. 1(a). The two terms of the expression, $a^2 \cdot c$ and $a \cdot b \cdot c$ can be traced as paths from the root to terminal 1 (ONE). The label $(^\wedge 2, 1)$ on the edge from node a to node c denotes quadratic term a^2 with weight = 1. The remaining edges are linear, each labeled with $(^\wedge 1, 1)$.



(a) (b)

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

TED Linearization: It has been shown that the TED structure allows for efficient factorization and decomposition of expressions modeled as linear multi-variate polynomials [3, 4]. For example, a TED for expression $F = ab + ac$, for variable ordering (a, b, c) naturally represents the polynomial in its factored form, $a(b + c)$. Unfortunately, this efficiency is missing when considering optimization involving non-linear expressions. For example, in the TED for function $F = a^2c + abc$ in Figure 1(a), node a should be factored out, resulting in a more compact form $F = a(a + b)c$, but in its current form, the TED in Fig. 1(a) does not allow for such a factorization.

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

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

$$F(x) = f_0 + x \cdot f_1 + x^2 \cdot f_2 \cdots + x^n f_n \quad (2)$$
$$= f_0 + x_1(f_1 + x_2(\cdot f_2 \cdots + x_n \cdot f_n)) \quad (3)$$

By applying this rule, function $F = a^2c + abc$ can be viewed as $F = a_1a_2c + a_1bc$, which reduces to $F = a_1(a_2 + b)c = a(a + b)c$, see Figure 1(b).

TED linearization can be performed systematically by iteratively splitting the high-order TED nodes until each node has degree 1 and contains two children: one associated with a multiplicative (solid) edge, and the other with an additive (dotted) edge. The resulting linear TED is also canonical. A linearized TED for the expression $F = a^2c + abc$ is shown in Figure 1(b). In the remainder of this paper, we only consider linear TEDs. Although TED linearization has been known since the early TED stages, it has been used for purposes other than functional optimization. For example, a binary Taylor expansion diagram, BTD, [14] was proposed as a means to improve the efficiency of the internal TED data structure. Other, non-canonical TED-like forms have been used for the purpose of functional test generation for RTL designs [15].

4 TED Decomposition

The principal goal of factorization is to minimize the number of arithmetic operations (additions and multiplications) in the expression. An example of *factorization* is the transformation of the expression $F = ac + bc$ into $F = (a + b)c$, which reduces the number of multiplications from two to one. If a sub-expression appears more than once in the expression, it can be extracted and replaced by a new variable. This process is known as *common subexpression elimination* (CSE). A simplification of an expression by means of factorization or CSE is commonly referred to as *decomposition*.

Decomposition operations can be performed directly on the TED graph, taking advantage of its canonical representation. In fact, TED encodes the expression in a compact, factored form. The goal of TED decomposition described in this work is to find a factored form that will produce DFG with minimum hardware cost of the final, scheduled implementation. This is different than a straightforward minimization of the number of operations in the unsched-

uled DFG, which has been the subject of the known previous work [3, 4, 12].

The TED decomposition method described here extends the work of the original cut-based decomposition of Askar [4], which was based on the identification and selection of admissible cut sequences. The cut-based method was applicable only to TED graphs characterized by the presence of simple cuts: additive and multiplicative edges whose removal would separate the graph into two disjoint subgraphs, and hence was limited only to the disjoint decomposition. Many TEDs, such as the one shown in Figure 2, do not have a disjoint decomposition property.

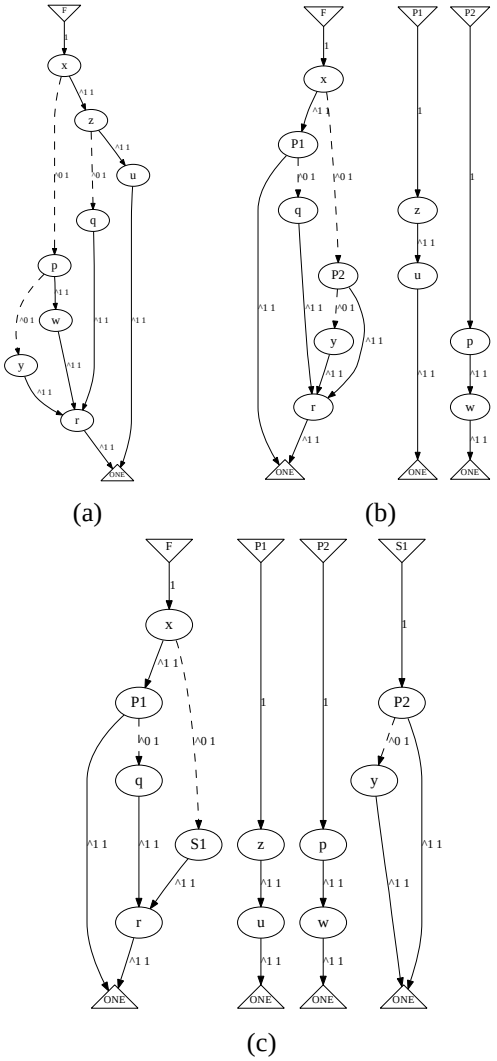


Figure 2. Complex TED decomposition for $F = x \cdot (z \cdot u + q \cdot r) + (p \cdot w + y) \cdot r$: (a) Original TED; (b) Simplified TED after product term substitutions, $P_1 = z \cdot u$ and $P_2 = p \cdot w$; (c) Simplified TED after sum term substitution, $S_1 = P_2 + y$.

The decomposition developed in this work applies to an arbitrary TED graph (linearized, if necessary), with both disjoint and non-disjoint decomposition. It applies a series of transformations of sum terms ($\sum v_i$) and product terms (Πv_i), represented by simple TED patterns, into *irreducible TED subgraphs*. Each irreducible subgraph is then replaced by a single node in a global, hierarchical TED, followed by disjunctive and conjunctive decomposition of the hierarchical TED. Disjunctive TED decomposition tries to identify additive edges, called *split edges*, whose removal decomposes the TED into two disjoint subgraphs. Conjunctive decomposition tries to identify the *dominators*. Dominator is a TED node with a property that all the paths from the root to terminal node 1 pass through this node. By construction, such a node defines a disjoint conjunctive decomposition. The resulting expression is simply a product of the subgraph above and below the dominator node. For example, node a_2 in the TED in Fig. 1(b) is a dominator, which decomposes the expression F conjunctively into $F = F_1 \cdot F_2$, where $F_1 = a_1$ and $F_2 = (a_2 + b)c$. Similarly, node c is a dominator in F and F_2. If neither disjunctive nor conjunctive decomposition exists in the graph, then the fundamental Taylor series decomposition is applied to the graph, resulting in non-disjoint decomposition.

The TED decomposition is illustrated with the example in Figure 2 for function $F = x \cdot (z \cdot u + q \cdot r) + (p \cdot w + y) \cdot r$. This TED does not have a single split-edge that would separate the graph disjunctively into two disjoint subgraphs; neither does it have a dominator that would allow it to decompose it conjunctively into disjoint subgraphs (note that r is not a dominator in this graph). Nevertheless, this function can be represented as a disjunction of two expressions $F_1 + F_2$, with $F_1 = x \cdot (z \cdot u + q \cdot r)$ and $F_2 = (p \cdot w + y) \cdot r$, sharing a common subgraph rooted at node r. Such a non-disjoint decomposition is accomplished in a systematic way on a TED as follows.

First, a series of nodes connected only by multiplicative edges, representing a product term, is represented by an irreducible TED and replaced with a single variable P_I. In this example, the following irreducible TEDs are identified and replaced by new variables: $P_1 = z \cdot u$ and $P_2 = p \cdot w$. The resulting hierarchical TED is shown in Figure 2(b).

Next, the sum terms are identified in the TED and substituted by new variables. A sum term appears in the TED graph as a set of variables, incident to the edges with a common node, and linked together by one or more additive edges. Such patterns can be readily identified by traversing the graph in a bottom-up fashion and creating, for each node v, a list of nodes reachable from v by a multiplicative edge. The procedure starts at terminal node 1 and traverses all the nodes in the graph bottom-up, in a reverse variable order. In our example, the set of nodes reachable from terminal node 1 is $\{P_1, r\}$. Since these nodes are not linked by an additive edge, they do not form a sum term in the expression. The list of nodes reachable from node r is $\{q, y, P_2\}$, of which $\{P_2, y\}$ are linked by an additive edge. Hence, they correspond to a sum-term $(P_2 + y)$. Such a term is substituted by a new variable $S1$ and represented as an irreducible TED. No other irreducible TED subgraph can be extracted. The resulting hierarchical TED, with the sum term $(P_2 + y)$ replaced by variable S_1, is shown in Figure 2(c).

This procedure is repeated iteratively until the top level TED is reduced to the simplest, irreducible form. The re-

sulting TED is then subjected to the final decomposition using the fundamental Taylor expansion principle. 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 following expressions are derived for the global TED in Figure 2(c) (Here $f(v)$ refers to a function of an irreducible TED rooted at node v): $F = f(x) = f(S_1) + x \cdot f(P_1)$, where $f(S_1) = S_1 \cdot f(r)$, $f(r) = r$, $f(P_1) = P_1 + f(q)$, $f(q) = q \cdot r$, $P_1 = z \cdot u$, $P_2 = p \cdot w$, and $S_1 = (P_2 + y)$.

5 DFG Optimization

The recursive TED decomposition procedure described in the previous section produces a simplified algebraic expression in factored form. By imposing additional rules regarding the ordering of variables in the expression, such a form can be made unique. We refer to such a form as Normal Factored Form (NFF).

Definition 1 *The factored form expression associated with a TED is called a Normal Factored Form (NFF) for that TED if there is one-to-one mapping between the operations in the factored form and the TED, and if the ordering of variables in the expression is compatible with that of the TED.*

The normal factored form for the TED in Figure 1(b) is $a_1(a_2 + b)c$. Although several other factored forms can be derived from this TED, such as: $c(a_2 + b)a_1$, etc., only $a_1(a_2 + b)c$ satisfies the condition for NFF. Specifically, there is exactly one addition $(a_2 + b)$, corresponding to the additive edge (a_2, b), and two multiplications associated with the dominator nodes a_2 and c. Furthermore, the ordering of variables in the expression is compatible with that of the TED. An important feature of the NFF is that it is unique for a TED with fixed variable order.

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

The proof comes directly from the construction of the TED decomposition algorithm, described in Section 4, where each split edge defines a disjunctive decomposition and a dominator defines a conjunctive decomposition.

It should be emphasized that the NFF of the decomposed TED depends only on the structure of the initial TED, which in turn depends on the ordering of its variables. Hence, variable ordering plays a central role in deriving decompositions that will lead to efficient hardware implementations. Several variable ordering algorithms have been developed, including static ordering and dynamic re-ordering schemes, similar to those in BDDs. However, the significant difference between variable ordering for BDDs and for TEDs is that ordering for linearized TEDs is driven by the complexity of the NFF and the structure of the resulting DFGs, rather than by the number of TED nodes.

DFG Generation: Once a TED has been decomposed, a structural Data Flow Graph (DFG) representation of the expression is constructed from its Normal Factored Form. Each irreducible TED is first transformed into a simple DFG using the basic property of the NFF: each additive edge in the TED maps into an addition operation and each multiplicative edge maps into a multiplication operation in the resulting DFG. All the DFGs are then composed together to form the final DFG.

DFG construction for the expression $F = x \cdot (z \cdot u + q \cdot r) + (p \cdot w + y) \cdot r$ from its Normal Factored Form is shown in Figure 2(c). The five multiplications in this NFF correspond to the three nontrivial multiplicative edges in the top TED graph and two nontrivial multiplicative edges in the subgraphs for P_1 and P_2 (S_1 does not have nontrivial multiplications). Similarly, there are three additions corresponding to the three additive edges.

It should be emphasized, however, that unlike Normal Factored Form the DFG representation is not unique. While the number of operators remains fixed, the DFG can be further restructured and balanced to minimize its latency. Traditional methods known from logic synthesis can be used for this purpose [8].

These two steps, variable ordering and DFG balancing, are at the core of the optimization techniques employed in this work. The actual delay of the operators and their arrival times are considered during such a restructuring in order to minimize the latency of the final implementation.

Replacing Constant Multipliers by Shifters: It is well known that multiplications by integers can be implemented more efficiently in hardware by converting them into a sequence of shifts and additions/subtractions. Standard techniques are available to perform such a transformation based on Canonical Signed Digit (CSD) representation. However, these methods do not address common subexpression elimination or shifter factorization.

We now present a systematic way to transform integer multiplications into shifters using the TED structure. This is done by introducing a special *left shift* variable into a TED, while maintaining its canonicity. The modified TED can then be optimized using all the known TED simplification methods.

First, each integer constant C is represented in CSD format as $C = \sum_i (k_i \cdot 2^i)$, where $k_i \in (-1, 0, 1)$. By introducing a new variable L to replace constant 2, C can be represented as $\sum_i (k_i \cdot 2^i) = \sum_i (k_i \cdot L^i)$. The term L^i in this expression can be interpreted as *left shift* by i bits. The next step is to generate the TED with the shift variables, linearize it, and perform the TED decomposition. Finally, in the DFG generated by the TED decomposition, the terms involving shift variables, L^k, are replaced by actual shifters (by k bits). The final DFG representation is minimal in terms of the hardware cost of its operators.

An example in Figure 3 illustrates this procedure for the expression $F = 7a + 6b$. The original TED for this expression is shown in Figure 3(a), and its DFG in Figure 3(b). The expression is then transformed into an expression with a shift variable L: $F = (L^3 - 1)a + (L^3 - L^1)b =$

$L^3(a+b) - (a + L \cdot b)$, shown in Figure 3(c). The nonlinear term, L^3, is then linearized and the TED ordered, as shown in Figure 3(d). The TED is then decomposed into the DFG, shown in Figure 3(e). After replacing variables L_i by L, the DFG in Figure 3(f) is obtained. Finally, all constant multiplications with inputs L^k are replaced by k-bit shifters, as shown in Figure 3(g). The optimized expression corresponding to this DFG is $F = ((a + b) << 2 - b) << 1 - a$, where "$<< k$" refers to left shift by k bits This implementation requires only three adders/subtracters and two shifters, a considerable gain compared to the two multiplications and one addition of the original expression $F = 6a + 7b$.

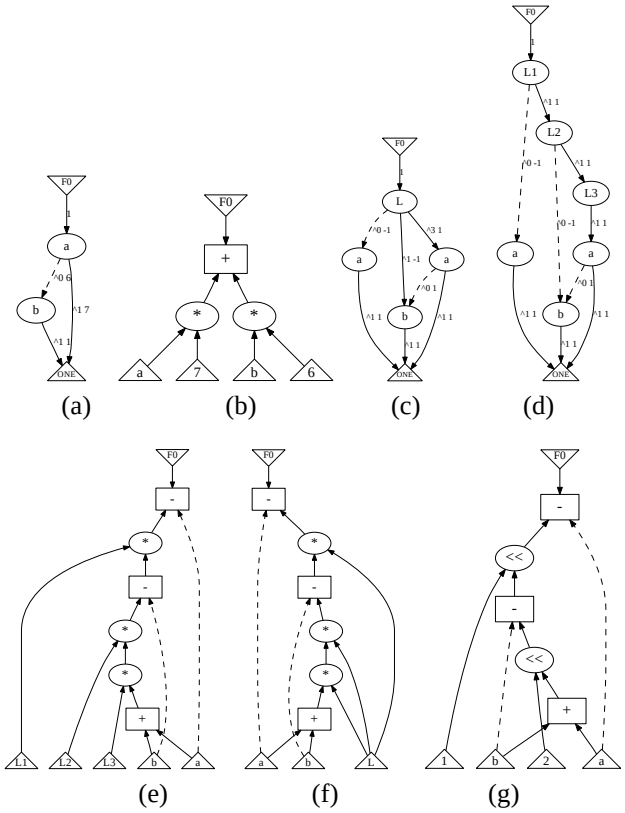


Figure 3. Replacing constant multiplications by shift operations for expression $F_0 = 7a + 6b$: (a) Original TED; (b) Initial DFG; (c) TED after introducing a *shift* variable L; (d) Linearized TED; (e) DFG derived from the linearized TED; (f) DFG after combining variables L_i into L; (g) Final DFG after replacing multipliers by shifters.

6 Experimental Results

The TED decomposition described in this paper was implemented as part of a prototype system TDS [5]. The design, written in C, is first compiled by a high-level synthesis system, GAUT, [16] [17]. to produce an initial data flow netlist. TDS transforms this netlist into a set of TEDs and performs all the TED- and DFG-related optimizations including: variable ordering, TED linearization, factorization, decomposition, replacement of constant multiplications by shifters, DFG construction, DFG balancing, etc. The optimized DFG is passed back to GAUT, which produces synthesizable VHDL code for final logic synthesis.

The results shown in the tables are reported for the following delay parameters: multiplier=18ns, adder/sub=8ns, shifter=9ns; clock period=10ns. Table 1 compares the implementation of a Savitzky-Golay (SG) filter using: 1) the original design; 2) the design produced by CSE decomposition system of [12]; and 3) produced by TDS. The table reports the number of arithmetic operations (adders, multipliers, shifters, subtractors) for each DFG solution; the actual number of resources used for a given latency; and the implementation area using GAUT (datapath only) and Synopsys DC Compiler (datapath, steering and control logic) synthesis tools. The results for circuits that cannot be synthesized for a given latency are marked with '–' (overconstrained).

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

Table 1. SG Filter implementations synthesized with the GAUT and Synopsys DC.

The minimum latency for the DFG extracted from the original SG design, without any modification, is 160 ns. The DFG solution produced by both CSE and TDS has minimum latency of 120 ns. However, the TDS implementation requires smaller area than both the original and CSE synthesized solution, as measured by both synthesis tools. In fact, all entries in the table show a tight correlation between the synthesis results of Synopsys DC and GAUT, which allows us to limit the results of other experiments to those produced by GAUT only.

Table 2 presents a similar comparison for designs from different domains (filters, digital transforms, computer graphic algorithms, etc.), synthesized with GAUT. A closer look at the Quintic Spline design shows that the CSE solution has smallest number of operations in its DFG and the latency of 140 ns. The DFG obtained by TDS produced the implementation with 110 ns, i.e., 21% faster, even though it had more DFG operations. And for the minimum latency of 140 ns, obtained by CSE, it produced implementation with area 22% smaller than CSE. Similar behavior can be seen for all the remaining designs. In all cases the latency of DFGs produced by TDS was smaller; and with the exception for Quartic Spline design, all of them have also smaller hardware area for the minimum latency produced by CSE.

Table 3 summarizes the implementation results for these benchmarks. We can see that the implementations obtained by TDS have latency smaller on average by 15.5% and 27.2% w.r.t. the CSE and original design.And for the reference latency (defined as the minimum latency

obtained by the other two methods), the TDS implementations have, on average, 7.6% and 36.3% smaller area w.r.t. the CSE and original design, respectively.

Design		Original design		CSE solution		TDS solution	
	Latency (ns)	+,×,≪,−	Area	+,×,≪,−	Area	+,×,≪,−	Area
Cosine wavelet	DFG →	9,12,9,14		10,10,4,5		9,10,12,7	
	L=110	–	–	–	–	3,2,4,1	447
	L=120	–	–	2,4,1,1	402	3,3,2,1	364
	L=130	–	–	2,4,1,1	402	2,3,2,1	356
	L=140	–	–	2,3,1,1	319	2,3,2,1	273
	L=150	–	–	1,3,1,1	311	2,2,2,1	273
	L=160	–	–	2,2,1,1	236	1,2,2,1	265
	L=170	–	–	1,2,1,1	228	2,2,1,1	236
	L=180	2,5,1,1	476	1,2,1,1	228	2,2,1,1	236
Chroma	DFG →	8,12,0,2		10,6,7,8		7,13,0,9	
	L=100	–	–	–	–	2,5,0,3	455
	L=110	2,4,0,2	364	2,3,3,2	413	2,4,0,2	364
Chebyshev polys	DFG →	3,15,5,0		7,7,4,1		6,7,10,6	
	L=100	–	–	–	–	2,3,2,1	347
	L=110	–	–	–	–	2,2,2,1	264
	L=120	–	–	1,3,1,1	302	1,2,2,1	256
	L=130	–	–	1,2,1,1	219	1,2,1,2	227
	L=140	–	–	1,2,1,1	219	1,2,1,1	219
	L=150	–	–	1,2,1,1	219	1,2,1,1	219
	L=160	–	–	1,2,1,1	219	1,2,1,1	219
	L=170	1,3,1,0	265	1,1,1,1	136	1,1,1,1	136
Quintic Spline	DFG →	5,28,2,0		5,13,3,0		6,14,4,0	
	L=110	–	–	–	–	1,5,1,0	460
	L=120	–	–	–	–	2,4,2,0	422
	L=130	–	–	–	–	1,4,1,0	377
	L=140	–	–	1,4,1,0	377	1,3,1,0	294
	L=150	–	–	1,3,1,0	294	1,3,1,0	294
	L=160	–	–	1,3,1,0	211	1,3,1,0	294
	L=170	–	–	1,2,1,0	211	1,3,1,0	294
	L=180	1,5,1,0	460	1,2,1,0	211	1,2,1,0	211
Quartic Spline	DFG →	4,21,2,0		5,11,4,0		5,13,4,0	
	L=100	–	–	–	–	2,5,1,0	468
	L=110	–	–	–	–	1,5,1,0	460
	L=120	–	–	–	–	2,4,1,0	385
	L=130	–	–	1,3,1,0	294	1,4,1,0	377
	L=140	–	–	1,3,1,0	294	1,3,1,0	294
	L=150	–	–	1,2,1,0	211	1,3,1,0	294
	L=160	1,5,0,0	423	1,2,1,0	211	1,3,1,0	294
VCI 4x4	DFG →	11,12,0,0		11,0,8,9		9,2,4,6	
	L=70	4,7,0,0	613	–	–	4,2,4,4	406
	L=80	4,6,0,0	530	–	–	4,2,2,2	302
	L=90	3,4,0,0	356	–	–	2,2,2,2	286
	L=100	3,4,0,0	356	2,0,4,2	208	2,1,2,2	203

Table 2. Comparison of minimum achivable latency and area for different designs. The area reported is for GAUT.

Design	TDS vs			
	Original		CSE	
	Latency (%)	Area (%)	Latency (%)	Area (%)
SG Filter	25.00	27.62	0.00	20.73
Cosine	38.88	50.42	8.33	9.45
Chrome	9.09	0.00	9.09	11.86
Chebyshev	41.17	48.68	16.66	15.23
Quintic	38.88	54.13	21.42	22.02
Quartic	37.50	30.50	23.07	-28.23
VCI 4x4	0.00	42.98	30.00	2.40
Average	27.22	36.33	15.51	7.64

Table 3. Percentage improvement of TDS vs Original and CSE on achievable latency; and area at the minimum achievable latency.

Conclusions: As shown by our results, a simple-minded minimization of the *number* of arithmetic operations in an algebraic expression, advocated in previous work, does not necessarily translate into a minimum hardware cost or a minimum latency in a scheduled DFG, and hence it does not guarantee a good hardware implementation. In contrast, the optimization method presented here is better suited for hardware implementations. It can discover solutions that have lower latency, unmatched by other methods; and and for the minimum latency obtained by those methods, require on average less area.

Acknowledgments This work has been supported by a grant from the National Science Foundation, award No. CCR-0702506.

References

[1] S. Gupta, M. Reshadi, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Dynamic Common Sub-expression Elimination during Scheduling in High-level Synthesis", in *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*, New York, NY, USA, 2002, pp. 261–266, ACM Press.

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

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

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

[5] University of Massachusetts, Amherst, "TDS - TED-based behavioral transformation system", http://www.ecs.umass.edu/ece/labs/vlsicad/tds.html

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

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

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] A. Hooshmand, S. Shamshiri, M. Alisafaee, B. Alizadeh, P. Lotfi-Kamran, M. Naderi, and Z. Navabi, "Binary Taylor Diagrams: an Efficient Implementation of Taylor Expansion Diagrams", in *ISCAS (1)*. 2005, pp. 424–427, IEEE.

[15] Bijan Alizadeh, "Word level Functional Coverage Computation", in *ASP-DAC*, Fumiyasu Hirose, Ed. 2006, pp. 7–12, IEEE.

[16] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, *High Level Synthesis from Algorithm to Digital Circuits*, Springer, 2008.

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