

TDS: Behavioral Transformation System

D. Gomez-Prado, Q. Ren, M. Ciesielski
ECE Dept., University of Massachusetts
Amherst, MA,

J. Guillot, E. Boutillon
Lab-STICC, Université de Bretagne Sud
Lorient, France

Abstract

TDS is an experimental software system to perform high-level transformations of designs specified at algorithmic and behavioral levels. The system transforms the initial design specifications into a data flow graph (DFG) optimized for final hardware implementation. The optimizing transformations are based on a canonical representation, Taylor Expansion Diagram (TED), followed by structural transformations of the resulting DFG network. The system is intended for data-flow and computation-intensive designs used in computer graphics and digital signal processing applications.

1. Introduction

The goal of high-level synthesis (HLS) is to transform an initial design specification into hardware architecture using a series of compilation and optimization steps. In doing so, HLS tools rely on a data flow graph (DFG) representation extracted directly from the original design specification. In order to explore another solution the user must rewrite the specification, from which another DFG is derived and subjected to architectural optimization.

TDS system offers a systematic way to transform the initial specification into a DFG that addresses the final hardware implementation and explores a larger space of architectural solutions. It uses a functional, canonical representation of the computation, called Taylor Expansion Diagram (TED) [1]. TED is a compact, graph-based data structure that provides an efficient way to represent word-level computation in a canonical, factored form [2]. It is particularly suitable for algorithm-oriented, data-intensive applications, where computations are modeled as polynomial expressions.

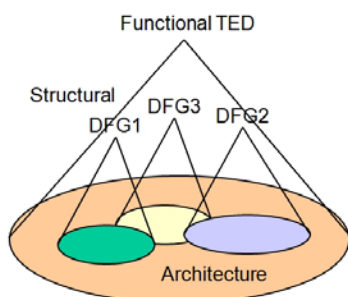


Figure 1: Architectural space exploration from TED.

The idea of a TED-enabled design space exploration is shown in Fig. 1. The result of traditional synthesis can be viewed in this diagram as a set of architectural solutions derived from a single data flow graph. Optimizations at this level are restricted to RTL transformations between

different architectures that are derived from the same DFG. Similarly, transformations between different DFGs, if at all possible, are limited to simple structural transformation, such as tree high reduction, graph balancing, etc., which can explore only a small fraction of the design space.

In contrast, transformation of the initial specification on the *functional* level can result in a larger set of structural DFGs, leading to better final implementations. This is accomplished in the TDS system by representing the design functionality in a canonical TED and transforming it into a DFG with the desired property. With this approach it is also possible to transform one DFG into another one by raising the level of abstraction to the functional TED space, from which a different DFG can be generated. Such an approach provides a much larger scope for architectural optimization and has biggest impact on final hardware implementation.

2. TED Decomposition

Transformation of the functional TED representation into a structural DFG is accomplished using a series of functional decomposition and optimization steps, including: TED linearization, factorization, common subexpression elimination (CSE), TED decomposition and DFG construction. An important feature of this process is that it addresses actual design objectives, such as latency and resource utilization, in the scheduled DFG. The following examples illustrate some of the capabilities of TDS.

a) $F=(a+b)(c-d)-ac-bc+ad+bd$. TED decomposition recognizes that F trivially reduces to 0. Surprisingly, very few, if any, HLS tools are able to find this simplification.

b) Hadamard Transform. The Hadamard transform is a multidimensional Digital Fourier Transform (DFT) of size $n=2^k$, used in communication algorithms. The TED-based decomposition is able to find, without any prior knowledge of the underlying structure, the optimal factorization of this transform. The initial computational complexity is $O(n^2)$ while the size of the optimized computational graph is $O(n\log(n))$.

c) Quintic spline: Quintic Spline is a function used in computer graphics designs. It is expressed as:

$$Q5 = zu^5 + 4au^4 + 6bu^3v^2 + 6cu^2v^3 + 4dvv^4 + qv^5.$$

The decomposition of its TED leads to a minimum-latency DFG shown in Fig. 2. Using a high-level synthesis tool, GAUT [4], the DFG generated by the TED decomposition results in a 39% reduction in minimum latency, and 50% reduction of area, compared to the DFG extracted directly from the initial specification. More details and examples of TED decomposition are given in [2] and [3].

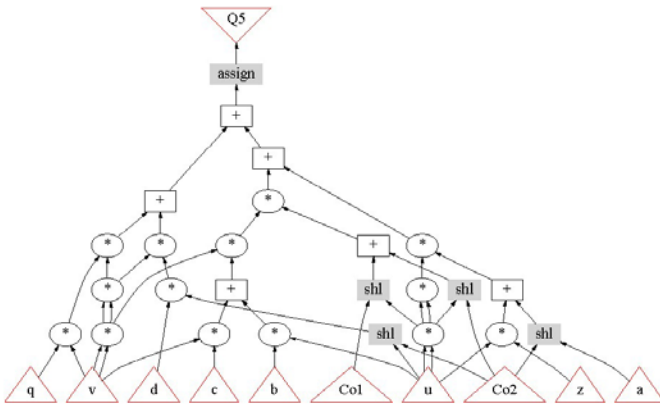


Figure 2: Optimized DFG for *quintic spline* design.

3. TDS system

The overall TDS system flow is shown in Fig. 3. The left part of the figure shows traditional high-level synthesis flow, using high-level synthesis tool GAUT [4], which extracts a data flow graph from the initial C/C++ specification. The flow on the right transforms the extracted DFG into a hybrid network composed of functional blocks, represented with TEDs, and other operators, represented as black boxes. Constant multiplications are replaced by shifters and adders to further minimize the hardware cost. The entire global DFG network is further restructured to minimize the design latency, subject to the imposed resource constraints. The resulting DFG network is then passed back to GAUT for high-level synthesis.

Fig. 4 shows a list of interactive commands available for the manipulation of TEDs and generation and optimization of the DFG networks. The system is available online [5].

<code>Ids 01> help</code>	
<code>- balance</code>	Balances the DFG or Netlist to minimize latency.
<code>- bblldown</code>	Moves down the given variable one position.
<code>- bblup</code>	Moves up the given variable one position.
<code>- bottom</code>	Moves the given variable to the bottom.
<code>- bottomdcse</code>	CSE Dynamic, with pre-movement of candidates to bottom.
<code>- bottomscse</code>	CSE Static, with pre-movement of candidates to bottom.
<code>- candidate</code>	Shows the candidate expression for CSE.
<code>- dcse</code>	CSE Dynamic, extract all candidates, move it, look for more candidates.
<code>- decompose</code>	Decomposes the TED in its Normal Factor Form.
<code>- dfactor</code>	Dynamic factorization.
<code>- dfg2nl</code>	Constructs the Netlist from the current DFG.
<code>- dfg2ted</code>	Flattens a DFG into TED.
<code>- dfgarea</code>	Balances the DFG to minimize the area.
<code>- dfgschedule</code>	Performs the scheduling of the DFG.
<code>- erase</code>	Erases a primary output from the TED.
<code>- extract</code>	Extracts a list of primary outputs from the TED or Netlist.
<code>- flip</code>	Flips the order of a linearized variable.
<code>- lcse</code>	CSE for linearized TED.
<code>- linearize</code>	Transforms a non linear TED into a linear one.
<code>- nl2ted</code>	Extracts from a Netlist all parts representable by TED.
<code>- poly</code>	Constructs a TED from a polynomial expression.
<code>- print</code>	Prints out TED information: statistic, normal factor form, etc.
<code>- printnl</code>	Prints out statistics of the number of operators in the Netlist.
<code>- purge</code>	Purges the TED, DFG and/or Netlist.
<code>- read</code>	Reads a script, a cdfg or an archived TED.
<code>- reloc</code>	Relocates the given variable to the desired position.
<code>- remapshift</code>	Remaps the shifters to <<.
<code>- reorder</code>	Reorders the variables in the TED.
<code>- scse</code>	CSE Static, extract a candidate, move it, look for more candidates.
<code>- shifter</code>	Replaces constant multipliers by shifters.
<code>- show</code>	Shows the TED, DFG or Netlist graph.
<code>- sift</code>	Heuristically optimize the level of the variable.
<code>- sub</code>	Substitutes an arithmetic expression by a variable
<code>- top</code>	Moves the given variable to the top.
<code>- tr</code>	Constructs a TED from a set of predetermined transform.
<code>- unlinearizedfg</code>	Un-linearizes a DFG linearized in TED.
<code>- vars</code>	Presets the order of variables, before any TED is constructed.
<code>- write</code>	Writes the existing Netlist/TED into *.lcdfg/ted1 file.

Figure 4: List of interactive commands of TDS

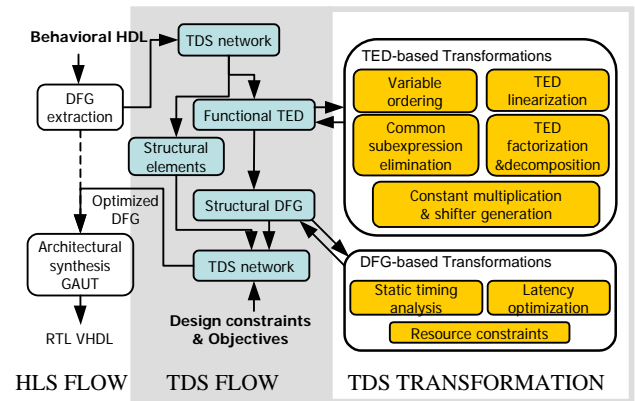


Figure 3: TDS system flow.

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

References

- [1] 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.
- [2] M. Ciesielski, S. Askar, D. Gomez-Prado, J. Guillot, and E. Boutillon, "Data-Flow Transformations using Taylor Expansion Diagrams", in *DATE*, 2007, pp. 455–460.
- [3] D. Gomez-Prado, M. Ciesielski, Q. Ren, J. Guillot, and E. Boutillon, "Optimization Data Flow Graphs to Minimize Hardware Implementations", in *DATE*, 2009.
- [4] GAUT, Université de Bretagne Sud Lab-STICC, <http://www-labsticc.univubs.fr/www-gaut/>, 2008.
- [5] TDS system, University of Massachusetts, Amherst, <http://www.ecs.umass.edu/ece/labs/vlsicad/tds.html>, 2008