Data-Flow Transformations using Taylor Expansion Diagrams

M. Ciesielski, S. Askar, D. Gomez-Prado,
University of Massachusetts
Department of Electrical & Computer Engineering
Amherst , MA 01003, USA
{*ciesiel, saskar, dgomezpr*}@*ecs.umass.edu*

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

Abstract: An original technique to transform functional representation of the design into a structural representation in form of a data flow graph (DFG) is described. A canonical, word-level data structure, Taylor Expansion Diagram (TED), is used as a vehicle to effect this transformation. The problem is formulated as that of applying a sequence of decomposition cuts to a TED that transforms it into a DFG optimized for a particular objective. A systematic approach to arrive at such a decomposition is described. Experimental results show that such constructed DFG provides a better starting point for architectural synthesis than those extracted directly from HDL specifications.

I. Introduction

Although considerable progress has been made in behavioral and high-level synthesis over the past two decades, new solutions are needed for advanced algorithm-oriented designs, such as multi-media and signal processing applications. Traditional high-level synthesis tools are effective at capturing the HDL specification by extracting a data flow graph (DFG) and mapping it directly into an architecture using structural optimization techniques, such as scheduling, resource allocation and binding [1]. However, these tools are less effective at the higher levels of abstraction. Most of these systems rely on a fixed data flow graph (DFG), extracted from the initial HDL specification, and provide limited means for modifying the initial data flow structure. In an attempt to explore other solutions the user often needs to rewrite the original specification, from which another DFG is derived and synthesized. This approach seriously limits the scope of the ensuing architectural synthesis and the quality of final hardware implementation.

Several attempts have been made to provide optimizing transformations in high-level synthesis [2], [3], [4], [5], [6], [7], [8]. Behavioral transformations have been also used in other areas, such as optimizing compilers [9] and logic synthesis [1]. With the exception for a few specialized systems for DSP code generation, such as SPIRAL [7], 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 [10], Spark [11] 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. To the best of our knowledge no systematic method for DFG modification for the purpose of behavioral optimization and synthesis has been presented to date.

In contrast, this paper presents a *systematic* method to perform behavioral transformations, based on a *canonical* representation of the computation, called Taylor Expansion Diagram (TED) [12]. Being a canonical representation, TED can capture an entire *class* of structural solutions, rather than a single DFG. Through a unique decomposition procedure proposed in this paper, this functional representation is converted into a structural representations (DFG), optimized for a particular design objective. This approach provides means for fast design space exploration directly from behavioral specifications.

A. Canonical TED Representation

Taylor Expansion Diagram [12] 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, $f(x, y, ...)$, can be represented using Taylor series expansion, w.r.t. variable x as follows:

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

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, .., 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

978-3-9810801-2-4/DATE07 © 2007 EDAA

expression represented by the graph is computed 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.

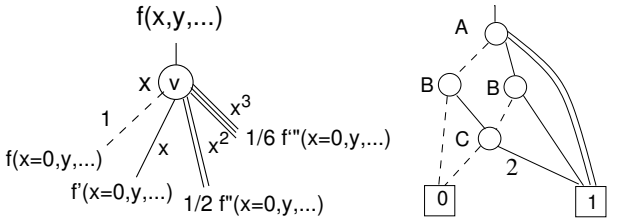

Fig. 1. TED: a) Decomposition principle; b) TED example for $F = A^2 + AB + 2AC + 2BC$

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).

In summary, 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 2(a) is computed as a sum of two paths from root to terminal node 1: $F = A{\cdot}B + A{\cdot}B^0{\cdot}C = A{\cdot}B + A{\cdot}C$, or equivalently $F = A{\cdot}(B + C)$. The latter form illustrates the fact that TED encodes useful factorization of the expression. This can be seen by noting that subexpression A, corresponding to the linear edge spanning out of variable A, is common to the two paths, $A \cdot B$ and $A \cdot C$, and can be factored out from the expression. This factorization is manifested in the graph by the presence of a subexpression $(B + C)$, rooted at node B, which can be extracted from the graph. This is an important feature of the TED representation, explored in factorization and common subexpression extraction [13].

In this context TED can serve as a canonical view of the computation, regardless of the specific data flow or architectural implementation. Several structural, data flow solutions can be derived from such a representation using the transformation process described here, guided by the desired objective (area, latency or power), resulting in an efficient hardware implementations.

B. Motivation

Consider a simple computation, $F = A \cdot B + A \cdot C$, where variables A, B, C are word level signals. The TED representation for this computation is shown in Figure 2(a). Figure 2(b) and (c) show two architectural solutions that can be obtained using state-of-the art commercial synthesis tools. The solution shown in Figure 2(b) minimizes latency ($L = 2$) and requires two multipliers and one adder, while the one in Figure 2(c) minimizes the number of operators at a cost of increased latency ($L = 3$). The two solutions correspond to different scheduling of the *same DFG*. However, another, better solution can be obtained by transforming the original specification $F = A \cdot B + A \cdot C$ into $F = A \cdot (B + C)$. The *modified* expression corresponds to a *different DFG*, which uses only one adder and one multiplier, and which can be scheduled in two control steps ($L = 2$), as shown in Figure 2(d). The existing commercial tools do not perform such transformations; they only derive a single, *fixed* data flow graph (DFG) for the given computation and have no capability to transform it into another DFG (better from a given objective point of view) prior to the architectural synthesis. In order to obtain an optimum solution (or to explore other solutions), the user must rewrite the original specification, from which another fixed DFG is derived and subjected to the architectural optimization.

To address this problem we propose a systematic way to transform the initial design specification into a data flow graph, optimized for a given objective. Figure 2 shows the initial HDL specification represented as a canonical TED along with all the solutions that can be obtained from this canonical representation. In particular, Figure 2(d) shows the solution with minimum latency subject to the imposed resource constraint.

II. Behavioral Transformations

A. System Flow

The basic idea behind the proposed system is to transform the *functional* TED representation of the design into a *structural* DFG representation. A partially scheduled DFG is obtained from TED by performing successive decomposition of the

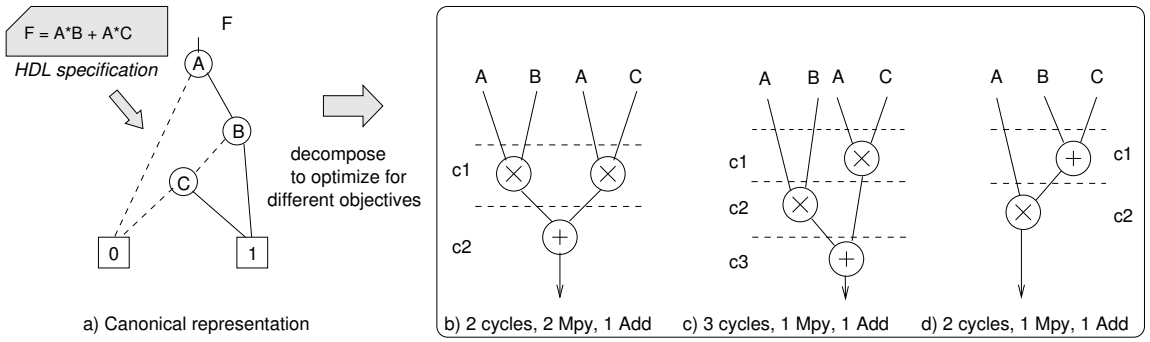

Fig. 2. Behavioral transformations: a) Canonical TED representation; b,c,d) Functionally-equivalent DFGs.

TED by means of cuts. The cut-based decomposition is guided in such a way as to optimize the DFG for a given objective. Input to the system is an algorithmic description of the design in C or behavioral HDL. The system flow, shown in Figure 3, is composed of the following steps. High-level synthesis system, GAUT [14], is used for front-end compilation and architectural synthesis, while the selection of best DFG is performed by cut-based TED decomposition.

1) **Compilation**: Compile the design to generate functional representation (TED). Minimize TED using variable ordering [15]. This phase is intended to minimize the number of ADD/MULT operations in the resulting DFG.
2) **TED Factorization and Extraction**: Generate hierarchical TED representation by performing TED factorization and common subexpression extraction using techniques described in [13]. Hierarchical TED is a graph, whose nodes represent TEDs of local functions.
3) **Transformation of TED into DFG**: Perform decomposition of local TEDs using sequence of cuts to create a partially scheduled DFG. The decomposition process is guided by the desired objective, such as latency, operators area, etc.

This paper concentrates on step 3 of the flow, using latency as a primary objective.

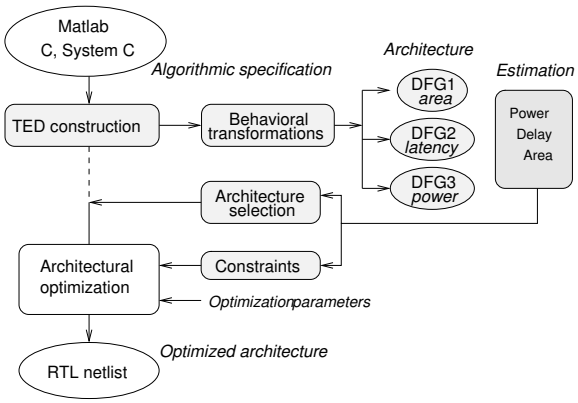

Fig. 3. System flow: behavioral transformation and DFG generation provided on top of architectural synthesis

B. Transforming TED into DFG

The proposed TED-DFG transformation is achieved by performing iterative cut-based TED decomposition. It is based on: (a) identifying additive and multiplicative *cuts* in the graph that will separate the expression encoded in the TED into smaller subexpressions, and (b) selecting a cut sequence that will produce a DFG with desired properties (min latency, balanced structure, etc.). Each cut partitions the original TED into subexpressions, which are subsequently partitioned into smaller TEDs by applying new cuts.

An *additive cut*, denoted by A_i, partitions the expression disjunctively into two sub-expressions. Additive cuts are shown as vertical bars in the diagrams. A *multiplicative cut*, denoted by M_i, partitions the expression conjunctively, and represents the multiplication of two expressions, above and below the cut. These cuts are shown as horizontal bars in the TED. Figure 4 shows a TED for function $P = (G + H) + F(I + J)$. (To simplify the explanation, TED is represented with a single terminal 1, with all the edges leading to terminal node 0 removed.) Cut $A3$ in the figure partitions function P disjunctively into $(G+H)$ and $F(I+J)$. This is shown as the top level decomposition in Figure 5. Cut $M1$ is an example of a multiplicative cut. It breakes the right TED subgraph, $Q = F(I + J)$, generated by applying cut $A3$, into two expressions: F and $(I+J)$ to be composed conjunctively (refer to Figure 5).

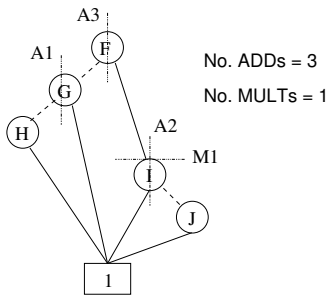

Fig. 4. Cuts in TED for $P = G + H + F(I + J)$.

Each time an additive or multiplicative cut is applied to a TED, a hardware operator (ADD or MULT) is introduced in the DFG to perform the required operation on the two subexpressions. This way, a functional TED representation (composed of algebraic *operations*) is eventually transformed into a structural representation (composed of hardware *operators*), a data flow graph (DFG).

C. Decomposition Algorithm

The additive (multiplicative) cut is called *admissible* if it partitions the TED expression into exactly two subexpressions in an additive (multiplicative) way, and it does not increase the number of operations determined by the original TED representation. Otherwise the cut is called *inadmissible*. Note that admissibility is a dynamic property; a cut may be inadmissible for a $TED(k)$ in step k, and become admissible for one of the subgraphs of $TED(j)$, at some step $j > k$. For example, cut $M1$ in Figure 4 is admissible only after the cut $A3$ has been applied. Subsequent application of $M1$ makes cut $A2$ admissible, etc. Finding an *admissible cut sequence* is one of the main tasks of this work. A cut-based decomposition is illustrated in Figure 5 for an admissible sequence $(A3, M1, A2, A1)$.

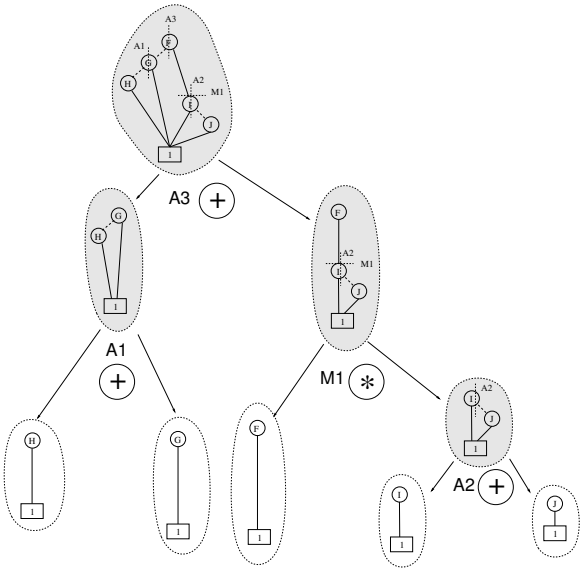

Fig. 5. Decomposition based on cut sequence $(A3, M1, A2, A1)$ on TED for P in Figure 4.

Figure 6 shows two DFGs derived from two different cut sequences, $(A3, A1, M1, A2)$ and $(A1, A3, M1, A2)$, applied to the TED in Figure 4.

The following lemma is an obvious consequence of the cut-based procedure described above.

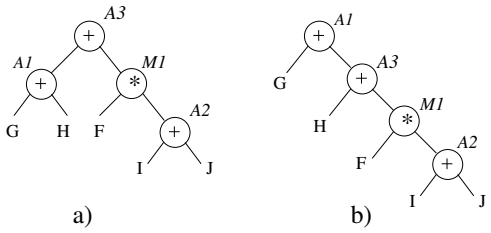

Fig. 6. Data flow graphs obtained from different cut sequences for TED of P in Figure 4: a) DFG for sequences $(A3, A1, M1, A2)$, $(A3, M1, A1, A2)$, or $(A3, M1, A2, A1)$; b) DFG for sequence $(A1, A3, M1, A2)$.

Lemma: Each admissible cut sequence generates a unique DFG.

Proof: Each sequence, being admissible, produces a legal DFG. Furthermore only one such DFG can be obtained for a given sequence since different DFGs mean different partial order of DFG nodes (operators), hence different cut sequence.

The resulting DFGs may differ in terms of critical path, balance structure, etc., which translates into physical characteristics of the designs (latency, power dissipation due to spurious computation, etc.), synthesized from these DFGs. An ordering of cuts is sought that will optimize the DFG structure for a given design objective (tree balance, longest path, etc).

We use a Branch & Bound algorithm to search for admissible cut sequences that optimize a particular design objective. Here we concentrate on minimizing latency, i.e., the critical path of the expected DFG. The algorithm uses cut admissibility for branching, and latency of the current solution as the bounding condition.

III. Results

The TED-to-DFG transformation procedure has been implemented as part of an experimental tool *TEDify*. [16], available on the web. The generated DFGs have been compared to those obtained by the GAUT high-level synthesis system [14] for a number of DSP designs.

Figure 7 shows the DFG extracted by GAUT from the original VHDL or C description of an FIR16 filter. This DFG is then used in subsequent architectural optimization. Figure 8 shows the DFG obtained by our TEDify tool: a TED was extracted from the same specification and was subjected to a cut-based TED decomposition to produce a DFG with minimum latency.

The reduction in the number of multiplier operations, from 16 required by the original specification extracted by GAUT to 8 produced by TEDify, has been accomplished by the first two steps of the flow (1: TED variable ordering; and 2: TED factorization and expression extraction), while the number of adders remained the same. The latency of a DFG produced in

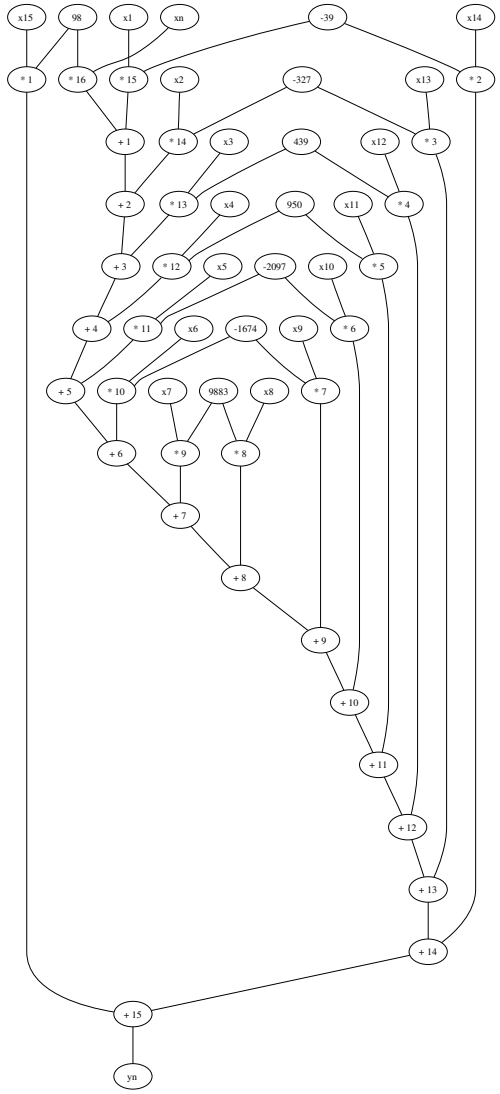

Fig. 7. DFG for FIR16 produced by GAUT

step 3 of the flow was reduced from 16 control steps in the specification produced by GAUT to 5 control steps generated by TEDify.

Table I shows the pre-synthesis results; it compares the DFGs (in terms of the number of operators and control steps) produced by GAUT and TEDify for a number of DSP designs (written in C or behavioral VHDL). In most cases the latency was reduced, sometimes drastically.

Table II shows the results for FIR-16 and IIR filter designs synthesized by GAUT under resource constraints for the original DFG, extracted by GAUT and for the DFG computed by TEDify. The resource limits imposed on the number of ADD and MULT operators are shown in columns 2 and 3 of the table. The next two columns show the latency (in ns) of the final architecture for the two DFGs (produced by GAUT

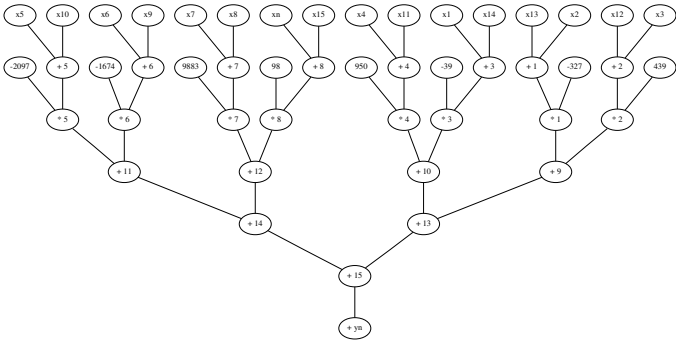

Fig. 8. DFG for FIR16 produced by TEDify

Design	DFG: GAUT			DFG: TEDify		
	# Operations		Latency	# Operations		Latency
	ADD	MULT	c-steps	ADD	MULT	c-steps
IIR	4	4	5	4	0	3
FIR16	15	16	16	15	8	5
Prodmat	48	64	4	48	64	3
Ellip	20	0	11	36	0	7
DCT	48	64	4	48	64	3

TABLE I

PRE-SYNTHESIS RESULTS: COMPARISON OF DFG EXTRACTED FROM THE INITIAL SPECIFICATION BY GAUT AND DFG PRODUCED BY TEDIFY

and TEDify, respectively), synthesized by GAUT. The delay of ADD was set to 10 ns and MULT to 20 ns, with the clock cycle set to 10 ns (these parameters are user-defined and can be altered).

Benchmark	# of Operators		Latency(ns)	
	ADD	MULT	GAUT	TEDify
FIR16	1	1	330	210
	1	2	180	150
	1	4	170	150
	1	8	170	150
	2	1	330	210
	2	2	180	130
	2	4	170	90
	2	8	170	90
	4	1	330	210
	4	2	180	130
	4	4	170	90
	4	8	170	70
IIR	1	1	90	40
	1	2	90	40
	2	1	60	30
	2	2	60	30

TABLE II

POST-SYNTHESIS RESULTS FOR RESOURCE-CONSTRAINED SYNTHESIS: COMPARISON OF DESIGN PARAMETERS FOR THE DFG EXTRACTED BY GAUT AND DFG GENERATED BY TEDIFY

Figure 9 shows the Gantt chart for the FIR-16 filter, synthesized by GAUT using the original DFG (extracted by GAUT), with the number of resources limited to one ADD and one MULT. (see row 1 of Table II). Figure 10 shows the Gantt chart for the same design under the same resource constraint, synthesized by GAUT using the DFG generated by TEDify. We can see the difference in latency (210 ns vs 330 ns) and in the number of registers needed by the two solutions.

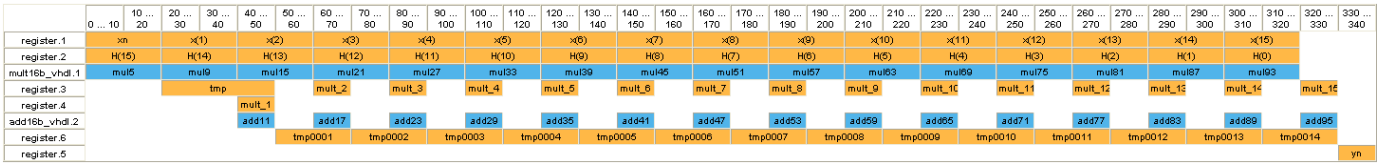

Fig. 9. Gantt chart for FIR16 design synthesized with the original DFG extracted by GAUT

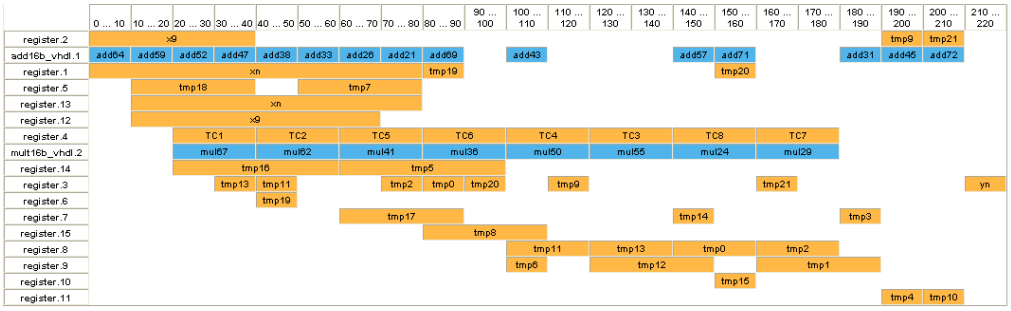

Fig. 10. Gantt chart for FIR16 design synthesized with DFG generated by TEDify

IV. Conclusions

We are currently using TED ordering which minimizes the number of nodes of the TED, but other objectives that will more directly address the number of operators (especially the multipliers) should be investigated. The correlation between the number of multiplicative edges and multipliers would suggest that ordering which minimizes the number of those edges, rather than the number of TED nodes, would be more appropriate for resource-limited applications. This objective should also drive the factorization and extraction algorithms, which are an important part of the TED decomposition scheme, prior to cut-based transformation onto DFG. Those and other practical issues are currently under investigation.

Furthermore, the area cost should be properly qualified by distinguishing multipliers with arbitrary operands from those that use constants, especially if the constants are powers of 2, in which case the multiplier can be replaced by a less expensive shifter.

Finally, there are natural limitations inherent to the TED representation. Currently TEDs can handle designs with adders, subtractors, multipliers, shifters and Boolean logic, but cannot efficiently handle other operators. For this reason the use of TEDs is currently limited to untimed algorithmic descriptions of DSP designs.

V. Acknowledgment

This work has been supported by a grant from the National Science Foundation under award No. CCR-0204146.

References

[1] G. DeMicheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[2] L. Guerra, M. Potkonjak, and J. Rabaey, "High level Synthesis for Reconfigurable Datapath Structures," *Proc. Intl. Conf. on Computer-Aided Design*, 1993.

[3] V. Chaiyakul, D. Gajski, and R. Ramachandran, "High-level transformations for minimizing syntactic variances," *Design Automation Conf.*, pp. 413–418, 1993.

[4] A. Chandrakasan, M. Potkonjak, R. Mehra, Rabaey J., and Brodersen R., "Optimizing Power Using Transformations," *IEEE Transactions on Computer Aided Design*, pp. 12–31, 1995.

[5] M. Potkonjak, S. Dey, and R. Roy, "Synthesis for testability using transformations," *ASP-DAC'95*, pp. 485–490, 1995.

[6] M. Srivastava and M. Potkonjak, "Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughtput," *IEEE Transactions on VLSI Systems*, 1995.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

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

[14] Université de Bretagne Sud LESTER, "GAUT, Architectural Synthesis Tool," http://lester.univ-ubs.fr:8080, 2004.

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

[16] http://tango.ecs.umass.edu/TED/Doc/html