# High Performance DSP Software Using Data-Flow Graph Transformations

Vojin Živojnović Sebastian Ritz Heinrich Meyr
Integrated Systems for Signal Processing, IS2
Aachen University of Technology
Templergraben 55, 52056-Aachen, Germany

## Abstract

This paper presents an overview of transformations for DSP programs given in form of coarse-grain data-flow graphs. The goal is to produce a functionally equivalent data-flow graph with improved characteristics, regarding modeling and/or implementation of DSP software. Retiming, unfolding, vectorization, clustering as well as node/arc set extensions are discussed. As an example, an application of the presented transformations to the design of a satellite receiver is presented.

## 1 Introduction

In the last years a large number of block-diagram oriented DSP software synthesis tools appeared (eg. [1,2]). These tools offer a graphical, DSP-oriented programming environment resulting in higher modeling efficiency compared to standard sequential or concurrent programming. Additionally, by partly restricting the modeling freedom (no global variables, no aliased pointers, restricted control constructs etc.), and especially through extensive use of basic blocks, highly efficient implementations can be obtained.

Common for all software design tools and compilers is the use of the graph-theoretic model for the characterization of time and data dependencies among functional units. Especially for block-diagram oriented tools, the graph-theoretic model is the natural one. During the code synthesis most of these tools transform the original data-flow graph (block-diagram), into various functionally equivalent graphs. The transformed graphs are better suited for mapping on the software/hardware model or for further analysis.

The goal of this paper is to give an overview of data-flow graph transformations which yield to high-performance DSP software and hardware/software solutions. Beside the well known retiming transformation, we treat unfolding, vectorization, clustering and node/arc-set extension/reduction. Those transformations are selected which are suited for coarse-grain, probably multirate software design, and those improving single-processor software performance (speed and program/data memory utilization).

## 2 Background and Notation

There are three equivalent computational models which can be used to describe DSP programs with deterministic activation behavior. In 1966 Karp and Miller [3] introduced computation graphs in order to represent the execution of programs evaluating arithmetic expressions in parallel. Later on, in 1971, Commoner et al. [4] introduced a restricted version of the computation graph model named marked graph. Marked graphs can model concurrency and synchronization, but cannot model decisions or conditional executions. In 1987 Lee and Messerschmitt [5] introduced the synchronous data-flow graphs in order to model dependencies among functional units in DSP programs.

Computation graphs are the most general model. If the computation graph is restricted so that the threshold of the computation node is equal to the number of samples (tokens) the node consumes at each activation, the synchronous data-flow graph is obtained. If additionally all the rates and the threshold are equal one, the marked graph model is obtained. Although very often misleading, and surely highly redundant, all three models together with their accompanying theory contribute to the knowledge about models of DSP programs. In the sequel we shall reference our computational model simply as the graph.

We suppose that the DSP program is represented as a graph $G = \{V, E\}$. The nodes $v \in V$ model the functional behavior (processing operations). The arcs $e \in E$ are modeling the connections between processing elements and contain $d(e)$ initial samples (delays).

1058-6393/95 $4.00 © 1995 IEEE

A single delay represents a phase shift of one sample (iteration) in the data flow sequence. A node can be activated only if there are enough samples on all its incoming arcs. If some activation sequence can be found such that at some point no node can be activated, the graph is not a live graph. In the case of DSP software modeling only live graphs are of interest.

## 3 Graph Transformations

In the sequel we shall give an overview of the most important graph transformations leading to graphs with improved characteristics regarding implementation on programmable architectures. The presented transformations preserve the functional equivalence between the original and transformed graph, they are valid for any functionality of the processing nodes and are specially well suited for coarse-grain computational models. For every transformation a simple example is given.

### *Retiming*

Retiming $r$ is a transformation $r : G \rightarrow G_r$ of the original graph $G = \{V, E(d)\}$ to the retimed graph $G_r = \{V, E(d_r)\}$, where the new delay $d_r(e)$ of arc $e(u, v)$ connecting nodes $u$ and $v$ is defined by the equation

$$d_r(e) = d(e) + r(u) - r(v) \qquad (1)$$

and $r(v) : V \rightarrow Z$ is a node to integer function. Note that there is a sign difference in the retiming definition compared to [6]. Also, we shall suppose that the source nodes are always firable.

Retiming is a transformation which redistributes the delays (markings) on arcs of a graph without affecting functionality. It changes the intra-iteration precedence constraints to inter-iteration precedence constraints and vice versa.

Retiming was introduced by Leiserson et al. [6] in 1983 for the design of VLSI circuits. Applications of the retiming transformation in DSP software design range from multi- and single-processor scheduling [7,8,9], vectorization [10] to program code compaction [11].

On the following example we shall show how data memory compaction can be obtained using retiming. On Fig. 1a) and 1b) the original graph and its retimed version are given. If we suppose static buffer management, the original graph needs 13 memory locations for the signal buffers, whereas the retimed graph, obtained by the firing sequence $a^4 f^4 b^3 c^2 d$, needs only 7. This is a reduction in data memory utilization of almost 50%.

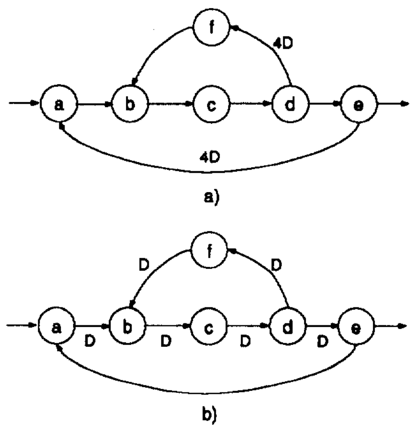

Figure 1: a) Original graph, b) Retimed graph.

If we are interested in obtaining a distribution of delays according to some linear criterion, we can formulate the problem as an integer linear programming problem. It is well known that in the general case this class of problems is NP-hard [12]. However, the specific form of constraints (1) makes it possible to solve the problem in polynomial time [6]. The incidence matrix is totally unimodular and the solution $r$ is guaranteed to be an all integer solution even without additional integer constraints [12].

In the multirate case the incidence matrix is not totally unimodular and $r$ has to be additionally constrained to an all integer vector resulting in an integer programming problem [13]. However, we can easily show that the multirate retiming problem with linear constraints can be solved in polynomial time too. The original graph can be transformed to an equivalent unit-rate graph [14], the retiming transformation applied, and the retimed graph transformed back to the multirate form. Because all three transformations are obtained in polynomial time, we can conclude that multirate retiming with a linear criterion function is not NP-hard.

The proposed procedure for multirate retiming, although of polynomial complexity, can be highly time consuming. The dimension of the equivalent unit-rate graph depends on the rate changes in the multirate graph. For a trivial, connected graph with two nodes, one arc and rates 100 and 99, the equivalent unit-rate graph consists of 199 nodes and 9900 arcs. If some retiming algorithm is of $O(|V|^2|E|)$ complexity, the retiming of the unit-rate equivalent would need $\sim 4 \times 10^8$ more time. A polynomial time multirate retiming algorithm which does not rely on the unit-rate equivalent graph is an interesting topic of further research.

*Unfolding - Blocking*

Unfolding [7] with factor $N$ is a transformation in which the transformed graph explicitly describes the precedence constraints in $N$ consecutive iterations. On Fig. 2b) the 2-unfolded ($N$=2) graph of the graph on Fig. 2a) is presented. In the case of multi-

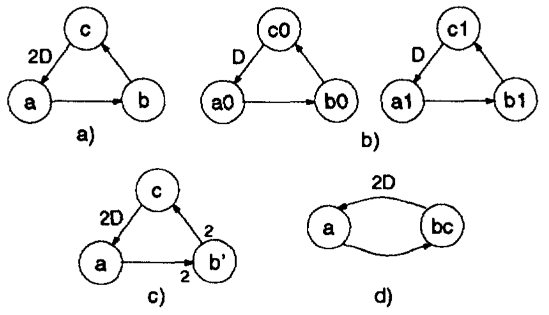

Figure 2: Graph transformations.

processor scheduling the unfolding transformation can always produce a graph which can be scheduled rate-optimally, i.e. reaching the iteration bound. The optimal unfolding factor $N$, which equals the least common multiple of all cycle delay counts, is the one which guarantees that every initial sample is processed during one iteration of the transformed graph. Unfolding increases the granularity of the program representation by the unfolding factor. As a consequence additional concurrency is recovered and more compact multiprocessor schedules can be obtained. The main disadvantage of unfolding is the increase in the size of the code necessary for executing the unfolded program.

If the unfolding factor is less than the optimal one, applying the retiming transformation can decrease the iteration period. The joint application of unfolding and retiming for unit-time graphs was treated in [15].

Blocking [14] is unfolding with a subsequent transformation to an acyclic precedence graph. It has been shown in [14] that blocking cannot guarantee rate-optimal multiprocessor schedules.

*Vectorization*

Vectorization is a transformation which increases the number of samples which are consumed and produced by each activation of a node, enabling a speedup of the node processing time (e.g. FFT based FIR filtering)[16]. On Fig. 2c) vectorization of node $b$ with vectorization factor $q = 2$ is done. In this way a new node $b'$ is obtained which can be activated only if two or more samples are presented at the input.

For acyclic graphs vectorization does not change the liveness. In the case of cyclic graphs the vectorization factors which guarantee liveness depend on the delay distribution in the cycles of the graph. In [10] it was shown that retiming can lead to efficient vectorization by concentrating the delays on link arcs of a graph.

*Clustering*

Clustering is a transformation in which a set of nodes is transformed into a single node. The graph on Fig. 2d) is obtained by clustering nodes $b$ and $c$ of the graph from Fig. 2a) to a new node denoted $bc$. It is obvious that clustering can change liveness. E.g. clustering nodes $a$ and $c$ would produce a terminating graph. However, by retiming nodes $a$ and $b$ once and then clustering $a$ and $c$ produces a live graph.

Clustering was applied for multiprocessor mapping [17,8], as well as for the reduction of the context-switching overhead [18] in single-processor code generation.

*Arc/Node Set Extension*

The role of the arcs in a graph is twofold. They describe flow of data and precedence constraints in the same time. Adding additional arcs to the graph without changing the functionality of the blocks introduces additional precedence constraints. A good example is the strongly connecting transformation which was already used in hardware design [6,19].

Strongly connecting is a transformation which converts a graph into a strongly connected graph. All source nodes are connected to the input connecting node and all sink nodes are connected to the output connecting node. A connecting arc from the output to the input connecting node is inserted which contains $N$ delays. If any path from a source to a sink node of the original graph is delay-free, i.e. has no latency, then $N > 0$ preserves liveness. It is easy to see that $N$ limits the processing node independent latency between output and input in case of retiming. Fig. 3 gives an example of the strongly connecting transformation. The inserted connecting nodes and arc are denoted dashed.

## 4 Implementation of Retimed Programs

In the previous section it was shown that retiming is a highly useful transformation which can be applied jointly with other transformations in order to obtain optimal results. Therefore, we shall discuss more deeply the implementation of retimed programs.

In general, the software implementation of a retimed program consists of two parts: the initial program

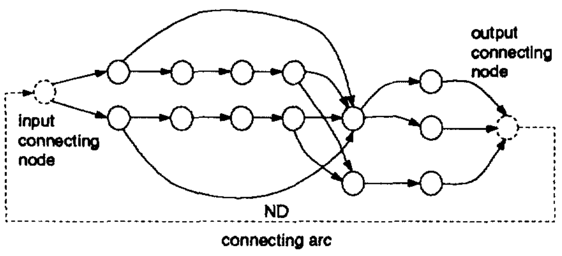

Figure 3: Strongly connecting transformation.

computing the new initial states (retimed delays) and the main program doing the actual processing. If some elements of the retiming vector $r$ are negative the computation of the values in the delays can be highly complex and sometimes even impossible. As an example, consider the computation depicted on Fig. 4a. There is no solution for the retiming when $r < 0$ for both nodes.

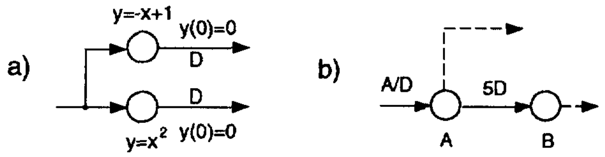

Figure 4: Implementation of retiming: Example graphs.

In order to discuss the problem let us suppose some retiming $r$ which transfers the delay vector from state $d$ to state $d_r$. Then every retiming vector $r$ with elements $r(v) - k, \forall v \in V$, and $k \in Z$, results in the same delay vector $d_r$. Setting $k$ to the minimum over all elements of $r$ converts every retiming to a nonnegative one.

The new initial samples can be computed at compile-time or at run-time. In the former case we suppose that the input samples are known at compile-time which guarantees that the new states can be easily computed. In this case latency is introduced, i.e. the program first produces the response to the input samples used for retiming, and then to the actual, run-time input.

Problems could arise when the program has to read the input samples from a fixed-rate I/O interface, as an A/D converter, at run-time. In this case the initial program and the transition to the main program should be able to process the input samples from the I/O fast enough in order to guarantee that no sample is lost.

On Fig. 4b a simple graph is given. Suppose that the necessary retiming is $r(A) = 2$ and $r(B) = 7$ and suppose that the execution times are $t(A) = 1$ and $t(B) = 5$ time units. The time needed for retiming will be 37 time units, which is more then 9 time units which are needed for one iteration of the actual processing. However, the first 5 samples of $B$ can be processed off-line, so per new input sample only 9 time units are necessary. We state that the time necessary for on-line retiming is always less or equal to the actual processing time for one iteration. After activating all the nodes having sufficient delays on the input, all the arcs having non-zero number of delays can be cut off. The new graph has obviously less nodes than the original one and needs less computation time.

It is well known that hardware retiming introduces latency in the output sequence. In the case of run-time software retiming no latency is introduced. This difference makes retiming very attractive for software design.

In the case of iterative computations, processing nodes with $f(0) = 0$ and a high stability margin, the initial states of the retimed graph can be filled with zeros, independently of the initial state of the original graph and the initial program can be dropped.

## 5 An Example

On Fig. 5 a block-diagram of the INMARSAT satellite receiver is given. Fig. 6 is the corresponding graph

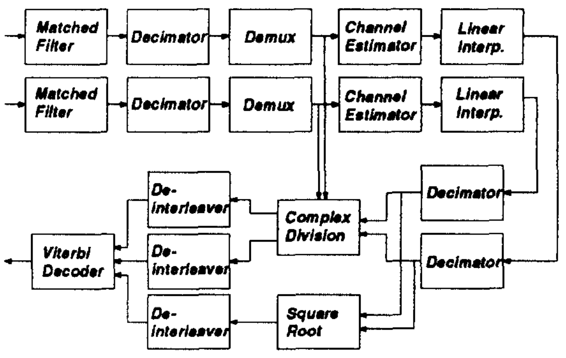

Figure 5: Block-diagram of the INMARSAT satellite receiver.

model on which strongly connecting was applied. During the simulation non-causal processing nodes have been used. In order to make them causal additional delays are introduced using retiming.

Retiming should minimize the total delay count under constraints on some of the arcs (denoted as $\underline{\geq x}$ on Fig. 6. Using the general integer linear programming algorithm [12], the graph on Fig. 7 is obtained. The computation of the presented example with 19 variables and 45 constraints took less than a second on a workstation.

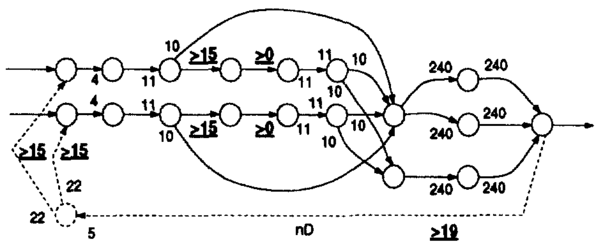

Figure 6: Multirate graph model of the receiver.

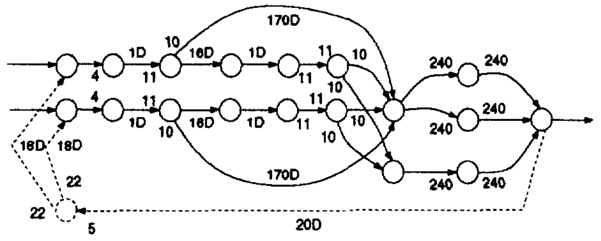

Figure 7: Retimed graph with minimum delay count.

## 6 Conclusions

In the paper an overview of data-flow graph transformation for DSP software design is presented. The selection was made according to the applicability on coarse-grain DSP programs for mapping on programmable single- and multi-processor architectures. Special attention was devoted to the retiming transformation which is the most powerful transformation useful especially if applied in connection to other transformations. At the end, an application of the multirate retiming transformation for the design of a satellite receiver is presented.

## References

[1] J. Buck, S. Ha, E. Lee, and D. Messerschmitt, "Ptolemy: A platform for heterogenous simulation and prototyping," in *Proc. 1991 European Simulation Conf.*, (Copenhagen, Denmark), June 1991.

[2] S. Ritz, M. Pankert, V. Živojnović, and H. Meyr, "High level software synthesis for the design of communication systems," *IEEE J. on Sel. Areas in Comm.*, vol. 11, pp. 348–358, April 1993.

[3] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing," *SIAM J.*, vol. 14, pp. 1390–1411, Nov. 1966.

[4] F. Commoner and A. Holt, "Marked directed graphs," *J. Comput. Syst. Sci.*, vol. 5, pp. 511–523, 1971.

[5] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. on Computers*, vol. C-36, No. 1, pp. 24–35, January 1987.

[6] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming," in *Proc. CalTech Conf. VLSI*, 1983.

[7] K. Parhi and D. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *IEEE Transactions on Computers*, vol. 40, pp. 178–195, February 1991.

[8] P. D. Hoang and J. M. Rabaey, "Scheduling of DSP programs onto multiprocessors for maximum throughput," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 41, pp. 2225–2235, June 1993.

[9] V. Živojnović, H. Körner, and H. Meyr, "Multiprocessor scheduling with A Priori node assignment," in *Proc. of VLSI'94 - La Jolla*, Oct. 1994. accepted for presentation.

[10] V. Živojnović, S. Ritz, and H. Meyr, "Retiming of DSP programs for optimum vectorization," in *Proceedings of the ICASSP'94 - Adelaide*, 1994.

[11] S. Bhattacharyya, *Compiling Dataflow Programs for Digital Signal Processing.* PhD thesis, University of California, Berkeley, 1994.

[12] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization.* Prentice-Hall, Inc., 1982.

[13] V. Živojnović, S. Ritz, and H. Meyr, "Optimizing DSP programs using the multirate retiming transformation," in *Proceedings of the EUSIPCO'94 - Edinburgh*, 1994.

[14] E. A. Lee, *A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors.* PhD thesis, University of California, Berkeley, 1986.

[15] L. Chao and E. Sha, "Unfolding and retiming dataflow DSP programs for RISC multiprocessor scheduling," in *Proc. of the ICASSP-92*, (San Francisco), pp. V(565–568), 1992.

[16] S. Ritz, M. Pankert, V. Živojnović, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," in *Intl. Conf. on Application-Specific Array Processors*, pp. 285–296, Prentice Hall, IEEE Computer Society, 1993.

[17] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, 1989.

[18] M. Willems, "Hierarchization of dedicated subdiagrams," tech. rep., Aachen University of Technology, 1994.

[19] S. Malik et al., "Retiming and resynthesis: Optimizing sequential networks with combinatorial techniques," *IEEE Trans. on CAD*, vol. 10, pp. 74–84, Jan. 1991.